Knowledge Representation, Reasoning, and the Design of Intelligent Agents

Michael Gelfond Yulia Kahl

January 28, 2013

Contents

Preface 9

1 Logic-Based Approach to Agent Design **11**
1.1 Modeling an Intelligent Agent 12
1.2 Simple Family Knowledge Base — an Example 14
1.3 A Historical Comment 16
1.3.1 The Axiomatic Method 17
1.3.2 Logic and Design of Intelligent Agents 18
Summary 19
References and Further Reading 20
Exercises 20

2 Answer Set Prolog (ASP) **23**
2.1 Syntax 23
2.2 Semantics 28
2.2.1 Informal Semantics 28
2.2.2 Formal Semantics 34
2.3 A Note on Translation from Natural Language 41
2.4 Properties of ASP Programs 43
Summary 47
References and Further Reading 47
Exercises 48

3 Roots of ASP **53**
3.1 First-Order Logic (FOL) 53
3.2 Non-Monotonic Logics 57
3.2.1 Circumscription 57
3.2.2 Autoepistemic Logic 59
3.2.3 Reiter's Default Theories 60
3.3 ASP and Negation in Logic Programming 62

3.3.1 Clark's Completion . . . 63
3.3.2 Well-Founded Semantics . . . 66
Summary . . . 69
References and Further Reading . . . 70
Exercises . . . 72

4 Creating a Knowledge Base **75**
4.1 Reasoning about Family . . . 76
4.1.1 Basic Family Relationships . . . 76
4.1.2 Defining Orphans . . . 79
4.1.3 Defining Ancestors . . . 81
4.2 Reasoning about Electrical Circuits . . . 84
4.3 Hierarchical Information and Inheritance . . . 88
Summary . . . 94
References and Further Reading . . . 95
Exercises . . . 95

5 Representing Defaults **99**
5.1 A General Strategy for Representing Defaults . . . 99
5.1.1 Uncaring John . . . 100
5.1.2 Cowardly Students . . . 103
5.1.3 A Special Case . . . 105
5.2 Knowledge Bases with Null Values . . . 105
5.2.1 Course Catalog . . . 105
5.3 Simple Priorities between Defaults . . . 107
5.4 Inheritance Hierarchies with Defaults . . . 111
5.4.1 Submarines Revisited . . . 111
5.4.2 Membership Revisited . . . 112
5.4.3 The Specificity Principle . . . 113
5.5 (*) Indirect Exceptions to Defaults . . . 115
Summary . . . 120
References and Further Reading . . . 120
Exercises . . . 121

6 Answer Set Programming **127**
6.1 Computing Hamiltonian Paths . . . 129
6.2 Solving Puzzles . . . 135
6.2.1 Sudoku Puzzle . . . 135
6.2.2 Mystery Puzzle . . . 140
Summary . . . 143

References and Further Reading . . . 144
Exercises . . . 144

7 Algorithms for Computing Answer Sets **147**
7.1 Finding Models of Propositional Formulas . . . 147
7.2 Finding Answer Sets of Logic Programs . . . 152
7.2.1 The First Solver . . . 153
7.2.2 The Second Solver . . . 160
7.2.3 Finding Answer Sets of Disjunctive Programs . . . 164
7.2.4 Answering Queries . . . 164
Summary . . . 165
Exercises . . . 166

8 Modeling Dynamic Domains **167**
8.1 The Blocks World — A Historic Example . . . 168
8.2 A General Solution . . . 177
8.3 $\mathcal{AL}$ Syntax . . . 182
8.4 $\mathcal{AL}$ Semantics — The Transition Relation . . . 183
8.4.1 States . . . 183
8.4.2 Transitions . . . 187
8.5 Examples . . . 190
8.5.1 The Briefcase Domain . . . 190
8.5.2 The Blocks World Revisited . . . 195
8.5.3 Blocks World with Concurrent Actions . . . 198
8.6 Non-determinism in $\mathcal{AL}$ . . . 200
8.7 Temporal Projection . . . 202
Summary . . . 203
References and Further Reading . . . 204
Exercises . . . 204

9 Planning Agents **209**
9.1 Classical Planning with a Given Horizon . . . 209
9.2 Examples of Classical Planning . . . 212
9.2.1 Planning in the Blocks World . . . 212
9.2.2 Igniting the Burner . . . 217
9.2.3 Missionaries and Cannibals . . . 222
9.3 Heuristics . . . 229
9.4 Concurrent Planning . . . 233
9.5 (*) Finding Minimal Plans . . . 234
Summary . . . 237

References and Further Reading . . . 239
Exercises . . . 239

10 Diagnostic Agents **243**
10.1 Recording the History of a Domain . . . 245
10.2 Defining Explanations . . . 247
10.3 Computing Explanations . . . 251
10.4 (*) Finding Minimal Explanations . . . 259
10.5 Importance of New Predicates *hpd* and *obs* . . . 260
Summary . . . 261
Exercises . . . 262

11 Probabilistic Reasoning **265**
11.1 Classical Probabilistic Models . . . 266
11.2 The Jungle Story in P-log . . . 268
11.3 Syntax and Semantics of P-log . . . 271
11.4 Representing Knowledge in P-log . . . 282
11.4.1 The Monty Hall Problem . . . 282
11.4.2 Death of a Rat? . . . 286
11.4.3 The Spider Bite . . . 287
11.4.4 The Bayesian Squirrel . . . 291
11.5 (*) P-log + CR-Prolog and the Wandering Robot . . . 294
Summary . . . 297
Exercises . . . 298

12 The Prolog Programming Language **301**
12.1 The Prolog Interpreter . . . 302
12.1.1 Unification . . . 302
12.1.2 SLD Resolution . . . 306
12.1.3 Searching for SLD Derivation — Prolog Control . . . 310
12.1.4 SLDNF Resolution . . . 312
12.2 Programming in Prolog . . . 314
12.2.1 The Basics of Prolog Programming . . . 314
12.2.2 Parts Inventory Program . . . 321
12.2.3 (*) Finding Derivatives of Polynomials . . . 325
Summary . . . 335
References and Further Reading . . . 335
Exercises . . . 335

A ASP Solver Quick-Start **337**

B Approximating STUDENT **341**
B.1 ASPIDE 341
B.2 QSystem 344

Preface

This is a book about knowledge representation and reasoning (KRR) — a comparatively new branch of science which serves as the foundation of Artificial Intelligence, Declarative Programming, and the design of knowledge-intensive software systems. Our main goal is to show how a a software system can be given knowledge about the world and itself, and how this knowledge can be used to solve non-trivial computational problems. There are several approaches to KRR which both compete and complement each other. The approaches differ primarily by the languages used to represent the knowledge and by corresponding computational methods. This book is based on a knowledge representation language called Answer Set Prolog (ASP) and the answer set programming paradigm — a comparatively recent branch of KRR with a well-developed theory, efficient reasoning systems, methodology of use, and a growing number of applications.

The text can be used for classes in Knowledge Representation, Declarative Programming, and Artificial Intelligence for advanced undergraduate or graduate students in computer science and related disciplines including software engineering, logic, cognitive science, etc. It will also be useful to serious researchers in these fields who would like to learn more about the answer sets programming paradigm and its use in KRR. Finally, we hope it will be of interest to anyone with a sense of wonder about the amazing ability of humans to derive volumes of knowledge from a collection of basic facts. Knowledge representation and reasoning, located at the intersection of mathematics, science and the humanities, provides us with mathematical and computational models of human thought and gives some clues to the understanding of this ability. The reader is not required to know logic or to have previous experience with computational systems. However, some understanding of mathematical method of thinking will be of substantial help.

We have attempted to maintain a proper balance between mathematical analysis of the subject and practical design of intelligent agents — soft-

ware systems capable of using knowledge about their environment to perform intelligent tasks. Beginning with simple question-answering agents, we progress to more and more complex ones and explain how common problems of knowledge representation and reasoning such as commonsense (default) reasoning, planning, diagnostics and probabilistic reasoning are solved with ASP and its extensions. The precise mathematical definitions of basic concepts are always accompanied by informal discussions and by examples of their use for modeling various computational tasks performed by humans. Readers are encouraged to run programs to test their agent's ability to perform these tasks using available, state-of-the-art ASP reasoning systems.

Of course the worth of a particular KRR theory is tested by the ability of software agents built on the basis of this theory to behave intelligently. If, given a certain amount of knowledge, the agent exhibits behavior that we believe reasonable for a human with exactly the same knowledge, we deem the theory to be a step in the right direction.

We hope that serious readers will learn to appreciate the interplay between mathematical modeling of a phenomenon and system design and better understand the view of programming as *refinement of specifications*. Even though the book does not contain large practical projects, we believe that the skills learned will be of substantial use in the marketplace.

We meant for this book to be a foundation for those interested in KRR. Our desire to limit the material to one semester forced us to skip many topics closely related to our approach. This includes various constraint and abductive logic programming languages and algorithms, descriptions of important ASP reasoning methods, the methodology of transforming ASP programs to improve their efficiency, and a large body of useful mathematical knowledge. The book does not cover other important KR topics such as descriptive logics and their use for the Semantic Web and other applications, natural language processing, etc. To help those interested in building on the foundation presented and learning more about these other topics, we have included a section on references and further reading in each chapter.

Chapter 1

Logic-Based Approach to Agent Design

The goal of artificial intelligence is to learn how to build software components of intelligent agents capable of reasoning and acting in a changing environment. To exhibit intelligent behavior, an agent should have a mathematical model of its environment and its own capabilities and goals, as well as algorithms for achieving these goals. Our aim is to discover such models and algorithms, and to learn how to use them to build practical intelligent systems. Why is this important? There are philosophical, scientific, and practical reasons. Scientists are getting closer to understanding ancient enigmas like the origins and the physical and chemical structure of the universe, the basic laws of development of living organisms, etc., but still know comparatively little about the enigma of thinking. Now, however, we have the computer — a new tool which gives us the ability to test our theories of thought by designing intelligent software agents. In the short time that this tool has been applied to the study of reasoning, it has yielded greater understanding of cognitive processes and continues to produce new insights on a regular basis, giving us much hope for the future. On the software engineering front, mathematical models of intelligent agents and the corresponding reasoning algorithms help develop the paradigm of declarative programming which may lead to a simpler and more reliable programming methodology. And, of course, knowledge-intensive software systems, including decision support systems, intelligent search engines, and robots, are of great practical value. In addition, attempts to solve the problems of AI illuminate connections between different areas of computer science and between CS and other areas of science including mathematical logic, philosophy and

linguistics.

1.1 Modeling an Intelligent Agent

In this book when we talk about an **agent**, we mean an entity which observes and acts upon an environment and directs its activity towards achieving goals. Note that this definition allows us to view even the simplest programs as agents. A program usually gets information from the outside world, performs an appointed reasoning task, and acts on the outside world by printing the output, making the next chess move, starting a car, or giving advice. If the reasoning tasks an agent performs are complex and lead to nontrivial behavior, we call it intelligent. If the agent readily adapts its behavior to changes in its environment, it is called adaptive. If it performs tasks independent of human control, we call it autonomous. An agent can possess some or all of these qualities in varying degrees. For example, consider a program in charge of control of a large system for paper production. Among its many complex tasks, it can make decisions based on temperature readings or thickness measurements and adjust the speed of a conveyer belt and alert the operator. Clearly this program is supposed to observe, think, and act on the environment to achieve certain goals. It is intelligent, as well as adaptive in response to its sensors. And it is certainly autonomous to a certain degree, although its decisions can be overridden by an operator. Of course the scope of the program and hence its ability for intelligent behavior are very limited.

As common in many AI and Computer Science texts, we ignore engineering tasks related to the agent's physical interaction with the world and concentrate on modeling the agent and designing software components responsible for its decision making.

A mathematical model of an intelligent agent normally consists of

- a *language(s)* for representing the agent's knowledge;
- *reasoning algorithms* which use this knowledge to perform intelligent tasks, including planning, diagnostics, and learning. Most such algorithms are based on sophisticated search and are capable of solving problems of non-polynomial complexity;
- an *agent architecture* which is the structure combining different submodels of an agent (normally related to different reasoning tasks) in one coherent whole.

In this book we consider the following typical agent architecture. An agent's memory contains knowledge about the world, as well as that entity's capabilities and goals. The agent:

1. observes the world, checks that its observations are consistent with its expectations, and updates its knowledge base;

2. selects an appropriate goal G;

3. searches for a plan (a sequence of actions) to achieve G;

4. executes some initial part of the plan, updates the knowledge base, and goes back to step (1).

Sometimes, we refer to these four steps as the **agent loop**. In step one, notice that the agent does not assume that it is the sole manipulator of its environment. This allows for incorporation of external events, as well as failure analysis; for example, the agent can compare its expected world model to the one it observes and attempt to *explain* the possible discrepancies between the two. In step two, goal selection can be implemented in many different ways, including prioritization, real-time user input, or random selection. Step three, planning, is an art in itself, requiring reasoning skills, efficient search and, possibly, weights on actions. And last, in step four, performing an action can involve hardware robots or, in other domains, "softbots," capable, say, of cleaning up directories or searching the Web. There is a myriad of issues involved in this simple outline.

This architecture is simple. A more sophisticated architecture may involve a complex structuring of various intelligent tasks and communication between modules performing these tasks in parallel, powerful learning and vision components, the ability to communicate and cooperate or compete with other agents, etc. In this book, however, we will limit ourselves to a sequential architecture presented in the agent loop and show that even this simple approach can lead to important insights and systems.

To implement this architecture we need to meet a number of difficult challenges. For example, how can we create a knowledge base for our agent? How can we make it function in a changing environment and maintain focus on its current goal without losing the ability to change goals in response to important events in its environment? How can we make it capable of explaining some of these events and use the explanations to fill gaps in its knowledge base? How can we get it to use its knowledge to do intelligent planning?

To answer these and other related questions, we chose to use the **logic-based approach**. The main idea is as follows. To make machines smart, we need to teach them how to reason and how to learn. Since most teaching comes from instruction, and most learning comes from the same, we need an efficient means of communication. As with any problem in computer science, the choice of language has a large impact on the elegance and efficiency of the solution. Languages differ according to the type of information their designers want to communicate to computers. There are two basic types: algorithmic and declarative. **Algorithmic languages** describe sequences of actions for a computer to perform. **Declarative languages** describe properties of objects in a given domain and relations between them. The **logic-based** approach proposes to

- use a declarative language to describe the domain,
- express various tasks (which may include requests to find plans or explanations of unexpected observations) as queries to the resulting program, and
- use an inference engine, i.e. a collection of reasoning algorithms, to answer these queries.

1.2 Simple Family Knowledge Base — an Example

To illustrate the logic-based approach used in this book, we will teach our computer basic facts about families. For simplicity we consider a small family consisting of three people — John, Alice, and their son, Sam. The domain will be structured in terms of binary relations $father$, $mother$, and $gender$. In these terms the family can be described by statements:

$$
\begin{array}{l}
father(john, sam). \\
mother(alice, sam). \\
gender(john, male). \\
gender(sam, male). \\
gender(alice, female).
\end{array}
$$

read as "John is the father of Sam", etc. (We are keeping things simple here and assuming that first names are enough to uniquely identify a person. Of course, if Sam had a father also named Sam, we would need to give each his own unique identifier.)

These basic relations can be used to teach a computer new notions. For instance, the following two rules can be viewed as a definition of relation

parent. We say that X is a parent of Y if X is a father of Y or X is a mother of Y.

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

Note that identifiers starting with capital letters denote variables, while those starting with lower-case letters denote names of objects (e.g. sam, john) and relations (e.g. father, parent).

The next rule defines the meaning of relation "X is a child of Y."

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

The above program is written in a variant of a declarative language called **Answer Set Prolog (ASP)**. Replacing symbol $\leftarrow$ by `:-` will turn the above statements into an executable program. We will discuss the full syntax and semantics of the language in Chapter 2.

To make sure that the program allows the computer to "understand" the material, we will test it by asking a number of questions. For example, "Is Sam a child of John?" or "Who are Sam's parents?" If the answers are satisfactory, the program learned. (Notice that this is exactly the method we use to check human understanding.) In what follows we will refer to an agent answering our questions as STUDENT. We can view STUDENT as a *theoretical* question-answering system. Doing this will allow us to avoid the discussion of details of actual systems, which we will later use to automate STUDENT's reasoning.

Since STUDENT is not capable of understanding even basic English sentences, we express the questions in its own language. To ask if Sam is a child of John, we type

$$?\ child(sam, john)$$

STUDENT will use the available knowledge and its reasoning mechanism to answer this question in the affirmative.

To ask "Who are Sam's parents?" we will use variables. Statement

$$?\ parent(X, sam)$$

is read as "Find X such that X is a parent of Sam." This time the reasoning mechanism of STUDENT will look for X satisfying the above query and return the names of Sam's parents: John and Alice.

Note that the real syntax for queries varies slightly with the implementation used. Here we simply put a question mark to distinguish the query from a fact.

The system still does not know that Sam is Alice's son. This is not surprising since it does not know the meaning of the word. For practice you can define relation $son(X, Y)$ — X is a son of Y.

The family example exhibits typical features of logic-based programming. The knowledge about the domain is stated in precise mathematical language and various search problems are expressed as queries, which are answered by the reasoning mechanism of STUDENT. The program is **elaboration tolerant**, which means that small changes in specifications do not cause global program changes. For instance, we expanded the original program several times to accommodate new relationships between family members. In each case, the new definitions were natural, and did not require changes to the rest of the program. It is equally easy to modify existing definitions. Suppose, for instance, that we would like to incorporate facts from a Spanish-language database which has statements $padre(a, b)$ instead of $father(a, b)$. These statements can be simply added to our knowledge base. The Spanish-English translation will be given by the rule

$$father(X, Y) \leftarrow padre(X, Y)$$

which modifies our previous definition of $father$. Other statements from the Spanish database can be incorporated in the same manner. Note that all relations based on $father$ remain unchanged.

The intelligence of this system can be significantly improved. Its vocabulary can be expanded. It can be supplied with a natural language interface allowing the user to state questions in English, etc. It can be taught to plan with the information it has been given as well as to find explanations for possible discrepancies between its knowledge and its observations of the real world. In this book we will discuss some of these enhancements. We will also address models of intelligent agents whose knowledge contains probabilistic information about the agent's domain. Usually such agents are studied within the probabilistic approach to AI, which is frequently viewed as an alternative to the logic-based approach. We do not share this view. Instead *we view probabilistic reasoning as commonsense reasoning about degrees of belief of a rational agent.* We will illustrate this view and its ramifications toward the end of the book.

1.3 A Historical Comment

The roots of the logic-based approach to agent design are very deep. An interested reader is encouraged to look for a serious discussion of these roots

in the history of logic. The goal of this section is to give a brief introduction to the development of the basic ideas which formed the foundations of this approach. More information on the history of the subject and various approaches to combining logic and artificial intelligence will be found Chapter 3.

1.3.1 The Axiomatic Method

The need to structure existing mathematical knowledge and to improve its reliability and coherence led mathematicians of ancient Greece to the development of the axiomatic method. The classical exposition of this method is given in Euclid's *Elements* in which all geometric knowledge available to Euclid is logically derived from a small collection of *axioms* — geometric propositions accepted without proof. For more than two thousand years the book served as a model of rigorous argument. It has been used to teach geometry from the time of its publication to the late 19th or early 20th century and long remained the second most read book in Europe after the Bible. Increased interest in the enigma of thinking, the 18th century shift of emphasis from geometry to calculus with its notion of continuum, inclusion of infinity and infinite sets as one of the main subjects of mathematics and the centuries long quest for the development of reliable reasoning methods for these newly created mathematical notions lead to substantial progress in the development of logic and the axiomatic method. At the beginning of the 20th century, logicians developed the general idea of *formal language* and used it to axiomatize set theory. Basic mathematical notions such as natural and real number, function, geometric figure, etc., were defined in terms of sets and their membership relations. As a result (almost) all of the mathematical knowledge of the early 20th century could be viewed as logical consequences of a collection of axioms which fit on a medium-size black board. This exceptional scientific achievement demonstrated the high degree of maturity of logic and is somewhat similar to the development of Mendeleev's periodic table in chemistry or the understanding of the structure of DNA in biology. Another important achievement of logic was the creation of the mathematical notion of a correct mathematical argument — the notion of proof. All these notions not only deepened our understanding of mathematical reasoning, but also had a strong influence on mathematics.

1.3.2 Logic and Design of Intelligent Agents

For a long time the influence of the axiomatic method outside of mathematics has been much weaker. Most likely, Gottfried Wilhelm Leibniz was the first to suggest that the method can have a much broader applicability, to add our understanding of other areas of science and to even substantially change human behavior. In his *The Art of Discovery* written in 1685, Leibniz says:

> The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate [calculemus], without further ado, to see who is right.

He was hoping that

> humanity would have a new kind of instrument increasing the power of reason far more than any optical instrument has ever aided the power of vision.

The idea, often referred to as the *Leibniz Dream*, greatly contributed to the development of Computing Science.

In the 20th century, Alfred Tarski and others, partly influenced by the Leibniz Dream, developed a research program whose goal was to investigate if the axiomatic method can be successfully applied outside of mathematics. In the 1950s John McCarthy came up with a program of applying this method to artificial intelligence, which gave birth to what is now called the **logic-based approach to AI.** The original idea was to supply computer programs with a substantial amount of knowledge about their problem domain represented in the language of mathematical logic and to use logical inference to decide what actions are appropriate to achieve their goals. The idea served as the foundation of declarative logical languages. For instance, logic programming language Prolog, developed in the late 1970s by Robert Kowalski, Alain Colmerauer, Philippe Roussel, and others, allows a programmer to supply the program with knowledge about its domain represented by so called definite clauses — a small subset of the language of classical mathematical logic. A computation problem is then reduced to proving that objects in the domain have a given property. This is achieved by an inference mechanism called SLD resolution. The language is Turing complete, which means that it can be viewed as a universal programming language. The simplicity of the class of definite clauses and the effectiveness

of its inference mechanism allows for efficient implementations. Unfortunately, Prolog has some non-declarative features. For instance, a simple modification of a program which preserves its logical equivalence may cause the program to go into an infinite loop. There are also some other substantial limitations preventing Prolog from being used as a full-scale knowledge representation language suitable for the design and implementation of intelligent agents, but we will not discuss them in this chapter.

Another interesting declarative language, Datalog, significantly expands the more traditional query answering languages of relational databases. It can be viewed as a subset of Prolog, but it is limited to representing domains with a finite number of objects. The inference mechanisms of Datalog, however, are quite different from that of Prolog and tailored toward query answering. In recent years there has been a substantial resurgence of Datalog leading to more research and more practical applications of the language.

Summary

Inspired by the *Leibnitz Dream* of applying the axiomatic method to understanding methods of correct reasoning, researchers have been applying the logic-based approach to the design of intelligent agents. This approach consists of using a declarative language for representation, defining reasoning tasks as queries to a program, and computing the results using an inference engine. One such branch of research involves Answer Set Prolog, a simple yet powerful language useful for creating formal representations of various kinds of knowledge. When coupled with an inference engine, these representations can yield answers to various non-trivial queries; in fact, complex reasoning tasks such as planning and diagnostics, as well as other important parts of the agent architecture, can be reduced to querying knowledge bases. It is important to note that the separation of knowledge representation and the reasoning algorithm allows for a high degree of elaboration tolerance and clarity. We can use a query-answering system, generically termed STUDENT, to illustrate the knowledge gained by the computer given a specific representation. Based on what the system knows is true, false, or unknown, we can evaluate the worth of such a representation, elicit unspoken assumptions, and learn about our own view of a domain.

References and Further Reading

The ideas of the logic-based approach to Artificial Intelligence and declarative programming were advocated by Cordell Green, Robert Kowalski, and John McCarthy, among others [61], [55], [75]. The importance of elaboration tolerance for knowledge representation was stressed by John McCarthy (see, for instance, [76]). The simple agent architecture used in this book was introduced in [15] and [10]. It is similar to earlier architectures [59] and can be viewed as a special case of a more general belief-desire-intentions (BDI) architecture from [96]. A popular account of this architecture that includes historical references can be found in [114].

The first account of the axiomatic method is given in Euclid's *Elements* [43]. A new, simplified approach to the axiomatization of geometry was suggested by David Gilbert in [56]. The discovery of paradoxes of set theory in the beginning of the 20th century lead to further advances in axiomatic methods and attempts to axiomatize all of mathematics. (See, for instance, the famous formalization of a large part of mathematics by Alfred Whitehead and Bertrand Russell [113].) Later work demonstrated that full axiomatization of set theory, if at all possible, would require substantial new advances in our understanding of the axiomatic method. The possibility of applying the axiomatic method to realms outside of mathematics was suggested by Gottfried Wilhelm Leibniz [63] and later expanded by Alfred Tarski [107] and others.

Valuable information about the early history of Prolog and Logic Programming can be found in [62], [29], and [28]. For more information on Datalog see, for instance, [3], [112], and [115].

Exercises

1. Compare and contrast the words *intelligent* and *rational*.

2. How can intelligent systems improve our ability to reason about a specific question?

3. Do some independent reading on Leibniz and explain why some people might consider him to be the first computer scientist.

4. In an address titled *Under the Spell of Leibniz's Dream*, Edsger Dijkstra says

> I think it absolutely astounding that he [Leibniz] foresaw how "the symbols would direct the reasoning," for how strongly they would do so was one of the most delightful discoveries of my professional life.

Describe one way in which, in computer science, symbols direct reasoning.

5. Read John McCarthy's 1959 paper titled *Programs with Common Sense*.

   (a) How does he define a program with common sense?

   (b) Compare and contrast STUDENT and the advice taker.

Chapter 2

Answer Set Prolog (ASP)

Answer Set Prolog is a declarative language; thus, an ASP program is a collection of statements describing objects of a domain and relations between them. Its semantics defines the notion of an **answer set** — a possible set of beliefs of an agent associated with the program.[1] The valid consequences of the program are the statements that are true in all such sets of beliefs. A variety of tasks can be reduced to finding answer sets, or subsets of answer sets, or computing the consequences of an ASP program.

2.1 Syntax

Whenever we define a formal language, we start with its alphabet. In logic, this alphabet is usually called a signature. Formally, a **signature**[2] is a four-tuple $\Sigma = \langle \mathcal{O}, \mathcal{F}, \mathcal{P}, \mathcal{V} \rangle$ of (disjoint) sets. These sets contain the names of the Objects, Functions, Predicates and Variables used in the program. (Predicate is just a term logicians use instead of the word relation. We will use both words intechangeably.) Each function and predicate name is associated with its **arity** — a non-negative integer indicating the number of parameters. For simplicity we assume that functions always have at least one parameter. Normally, the arity will be determined from the context. Elements of $\mathcal{O}, \mathcal{F}$, and $\mathcal{P}$ are often referred to as *object, function*, and *predicate*

[1] Historically, belief sets — under the name of *stable models* — were defined on a special class of logic programs written in the syntax of programming language Prolog. Once the definition was extended to apply to the broader class of programs defined below, the term *answer set* was adopted.

[2] In some books the definition of a signature slightly differs from the one used here in that it does not contain $\mathcal{O}$; instead, object constants are identified with function constants of arity 0.

constants, respectively. Often the word *constant* in this context will be replaced by the word *symbol*. In the family relations program from Chapter 1, our signature Σ_f is:

$$\mathcal{O} = \{john, sam, alice, male, female\}$$
$$\mathcal{F} = \{\ \}$$
$$\mathcal{P} = \{father, mother, parent, child, gender\}$$
$$\mathcal{V} = \{X, Y\}$$

Whenever necessary we will assume that our signatures contain standard names for non-negative integers, and for functions and relations of arithmetic, e.g. $+, *, \leq$, etc.

Sometimes it is convenient to expand the notion of signature by including in it another collection of symbols called **sorts**. Sorts are normally used to restrict parameters of predicates and parameters and values of functions.[3] For this purpose every object constant and every parameter of a predicate constant is assigned a sort, similarly for parameters and values of functions. The resulting five-tuple will be called a **sorted signature**. For instance, signature Σ_f above can be turned into a sorted signature Σ_s by introducing a sort, $person$, and by assigning it to object constants of the signature and to parameters of its predicate symbols. Sometimes these assignments will be written as $person = \{john, sam, alice\}$, $father(person, person)$, etc.

Object and function constants are used to construct terms. Terms not containing variables usually name objects of the domain. For instance, object constant *sam* is a name of a person, $abs(-2)$ where *abs* is a function symbol normally used for the absolute value, is a name for the number 2, etc. Here is the definition.

Terms (over signature Σ) 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 a term.

For simplicity arithmetic terms will be written in the standard mathematical notation; e.g., we will write $2 + 3$ instead of $+(2, 3)$. Terms containing no symbols for arithmetic functions and no variables are called **ground**. Here are some examples from our family program:

[3] This idea is familiar to users of procedural languages as parameters of procedures and functions are often associated with a type. Output types of parameters of functions are also common.

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

If a program contains natural numbers and arithmetic functions, then both $2 + 3$ and 5 are terms; 5 is a ground term while $2 + 3$ is not.

Let's extend our program signature to include the function symbol car. (Intuitively, we are assuming here that a person X has exactly one car, denoted by $car(X)$.) Now we can also make ground terms $car(john)$, $car(sam)$, and $car(alice)$ and non-ground terms $car(X)$ and $car(Y)$. So far our naming seems reasonable. There is, however, a complication — according to our definition $car(car(sam))$ is also a ground term but it does not seem to denote any reasonable object of our domain. To avoid this difficulty consider the sorted signature Σ_s defined as follows:

- Object constants of Σ_s are divided into two sorts, $person = \{john, sam, alice\}$ and $thing = \{car(X) : person(X)\}$[4].
- $\mathcal{F} = \{car\}$ where car is a function symbol which maps elements of sort $person$ into that of $thing$; e.g., car maps $john$ into $car(john)$.
- In addition to predicate symbols with sorted parameters like *father(person,person)*, *mother(person,person)*, etc., of Σ_f, we also add a new predicate symbol with parameters of two different sorts denoted by *owns(person,thing)*.

The definition of a term for a sorted signature is only slightly more complex than the one for an unsorted signature. For $f(t_1, \ldots, t_n)$ to be a term, we simply require sorts of the values of terms $t_1, \ldots, t_n$ to be compatible with that of the corresponding parameter sorts of f. So the terms of a sorted signature Σ_s are $john, X, car(sam), car(alice), car(X)$, etc. Note however that, since $car(sam)$ is of sort $thing$ and function symbol car requires sort $person$ as a parameter, $car(car(sam))$ is not a term of Σ_s.

Term and predicate symbols of a signature are used to define statements of our language. 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. If the signature is sorted, these terms should correspond

[4]Here $\{t(X) : p(X)\}$, where $t(X)$ is a term and $p(X)$ a condition, is the standard set-building notation read as *the set of all* $t(X)$ *such that* X *satisfies* p.

to the sorts assigned to the parameters of p. (If p has arity 0 then parentheses are omitted.) For example, $father(john, sam)$ and $father(john, X)$ are atoms of signature Σ_s, while $father(john, car(sam))$ is not. If the signature were to contain a zero-arity predicate symbol $light\_is\_on$ then $light\_is\_on$ would be an atom. If the ts do not contain variables, then $p(t_1, \ldots, t_n)$ says that objects denoted by $t_1, \ldots, t_n$ satisfy property p. Otherwise, $p(t_1, \ldots, t_n)$ denotes a condition on its variables. Other statements of the language can be built from atoms using various logical connectives.

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 and its negation are called **complementary**. The literal complementary to l will be denoted by $\bar{l}$. An atom $p(t_1, \ldots, t_n)$ is called **ground** if every term $t_1, \ldots, t_n$ is ground. Ground atoms and their negations are referred to as **ground literals**.

Now we have enough vocabulary to describe the syntax of an ASP program. Such programs serve as an agent's knowledge base, so we often use the words synonymously. A **program** Π of ASP consists of a signature Σ and a collection of **rules** of the form:

$$l_0 \; or \; \ldots \; or \; l_i \leftarrow l_{i+1}, \ldots, l_m, \; not \; l_{m+1}, \ldots, \; not \; l_n \tag{2.1}$$

where ls are literals of Σ. (To make ASP programs executable, we replace $\neg$ with `-`, $\leftarrow$ with `:-`, and *or* with `|`.)

For simplicity we assume that, unless otherwise stated, signatures of programs consist only of symbols used in their rules.

Symbol *not* is a new logical connective called **default negation**, (or **negation as failure**); *not* l is often read as "*it is not believed that* l *is true.*" Note that this does not imply that l is believed to be false. It is conceivable, in fact quite normal, for a rational reasoner to believe neither statement p nor its negation, $\neg p$. Clearly default negation *not* is different from classical $\neg$. While $\neg p$ states that p is false, *not* p is a statement about belief.

The disjunction *or* is also a new connective, sometimes called **epistemic disjunction**. The statement l_1 *or* l_2 is often read as "l_1 *is believed to be true or* l_2 *is believed to be true.*" It is also different from the classical disjunction $\vee$. While the statement $p \vee \neg p$ of propositional logic, called *the law of the exclusive middle*, is a tautology, the statement p *or* $\neg p$ is not. The former states that proposition p is either true or false, while the latter states that p is believed to be true or believed to be false. Since a rational reasoner can

remain undecided about the truth or falsity of propositions, this is certainly not a tautology.

The left-hand side of an ASP rule is called the **head** and the right-hand side is called the **body**. Literals, possibly preceded by default negation *not* are often called **extended literals**. The body of the rule can be viewed as a set of extended literals (sometimes referred to as the *premises* of the rule).

The head or the body can be empty. A rule with an empty head is often referred to as a **constraint** and written as

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

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

$$l_0\ or\ \ldots\ or\ l_i.$$

Following the Prolog convention, non-numeric object, function and predicate constants of Σ are denoted by identifiers starting with lower-case letters; variables are identifiers starting with capital letters. Variables of Π range over ground terms of Σ. A rule r with variables is viewed as the set of its **ground instantiations** — rules obtained from r by replacing r's variables by ground terms of Σ and by evaluating arithmetic terms (e.g. replacing 2+3 by 5). The set of ground instantiations of rules of Π is called the **grounding** of Π; program Π with variables can be viewed simply as a shorthand for its grounding. This means that it is enough to define the semantics of ground programs. For example, consider the program Π_1 with signature Σ where

$$\begin{array}{l}\mathcal{O} = \{a, b\}\\ \mathcal{F} = \{\ \}\\ \mathcal{P} = \{p, q\}\\ \mathcal{V} = \{X\}\end{array}$$

and rule

$$p(X) \leftarrow q(X).$$

Its rule can be converted to the two ground rules,

$$p(a) \leftarrow q(a).$$

$$p(b) \leftarrow q(b).$$

which constitute the grounding of Π_1, denoted by $gr(\Pi_1)$.

To proceed we also need to define what it means for a set of ground literals to satisfy a rule. We first define the notion for the parts that make up the rule, and then show how the parts combine to define the satisfiability of the rule.

Definition 2.1.1. *(Satisfiability) A set S of ground literals* **satisfies***:*

1. *l* **if** *$l \in S$;*
2. *not l* **if** *$l \notin S$;*
3. *l_1 or ... 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 r's body, it satisfies r's head.*

For example, let r be the rule

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

and let S be the set

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

Let's check if S satisfies r. First we check if the body of the rule is satisfied by S. The body consists of three extended literals: $q(b)$, $\neg t(c)$, and *not* $t(b)$. The first two are satisfied by clause (1) of the definition, and the last is satisfied by clause (2). By clause (4), we have that the body is satisfied. Since the body is satisfied, to satisfy the rule, S must satisfy the head (clause (5)). It does not, since neither $p(a)$ nor $p(b)$ is in S (clause(3)). Therefore, S does *not* satisfy r. There are many sets that do satisfy r including $\emptyset, \{p(a)\}, \{p(b)\}, \{q(b)\}, \{t(c)\}$, and $\{p(a), q(b), \neg t(c)\}$. For practice, check to see that this is so.

2.2 Semantics

First we will introduce the semantics of ASP informally to give a feeling for the nature of the reasoning involved. We will state the basic principles and give a number of examples. Then we will formally define the semantics by stating what it means for a program to entail a ground literal.

2.2.1 Informal Semantics

Informally, program Π can be viewed as a specification for answer sets — sets of beliefs that could be held by a rational reasoner associated with Π. Answer sets will be represented by collections of ground literals. In forming such sets the reasoner must be guided by the following informal principles:

1. Satisfy the rules of Π. In other words, believe in the head of a rule if you believe in its body.
2. Do not believe in contradictions.
3. Adhere to the "Rationality Principle" which says: "Believe nothing you are not forced to believe."

Let's look at some examples. Recall that in accordance with our assumption, the signatures of programs below consist only of symbols used in their rules.

Example 2.2.1.

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

Note that the second rule is a fact. Its body is empty. Clearly any set of literals satisfies an empty collection and hence, according to our first principle, we must believe $q(a)$. The same principle applied to the first rule forces us to believe $p(b)$. The resulting set $S_1 = \{q(a), p(b)\}$ is consistent and satisfies the rules of the program. Moreover, we had to believe in each of its elements. Therefore, it is an answer set of our program. Now consider set $S_2 = \{q(a), p(b), q(b)\}$. It is consistent, satisfies the rules of the program, but contains the literal $q(b)$ which we were not forced to believe in by our rules. Therefore, S_2 is not an answer set of the program. You might have noticed that, since we did not have any choices in the construction of S_1, it is the *only* answer set of the program.

Example 2.2.2. *(Classical Negation)*

$\neg p(b) \leftarrow \neg q(a).$	"Believe that $p(b)$ is false if you believe that $q(a)$ is false."
$\neg q(a).$	"Believe that $q(a)$ is false."

There is no difference in reasoning about negative literals. In this case, the only answer set of the program is $\{\neg p(b), \neg q(a)\}$.

Example 2.2.3. *(Epistemic Disjunction)*

$p(a)$ *or* $p(b).$ "Believe $p(a)$ or believe $p(b)$."

There are three sets satisfying this rule — $\{p(a)\}$, $\{p(b)\}$, and $\{p(a), p(b)\}$. It is easy to see though that only the first two are answer sets of the program. According to our Rationality Principle, it would be irrational to adopt the

third set as a possible set of beliefs — if we do we would clearly believe more than necessary.

Consider now the program

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

Here the first rule gives us two choices — believe $p(a)$ or believe $p(b)$. The next two rules force us to believe $q(a)$ regardless of this choice. This is a typical example of so called *reasoning by cases*. The program has two answer sets, $\{p(a), q(a)\}$ and $\{p(b), q(a)\}$.

Another important thing to notice about epistemic disjunction is that it is different from exclusive *or*. (Recall that if the disjunction between A and B is understood as exclusive, then A is true or B is true but not both. A regular disjunction, which is satisfied if at least one of its disjuncts is true, is sometimes called inclusive.)

Consider the following program:

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

The answer set of this program is $\{p(a), p(b)\}$. Note that if *or* were exclusive, the program would be contradictory.[5]

Of course the exclusive or of $p(a)$ and $p(b)$ can also be easily expressed in our language. This can be done by using two rules:

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

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

which naturally correspond to the definition of exclusive or. It is easy to check that, as expected, the program has two answer sets $\{p(a), \neg p(b)\}$ and $\{\neg p(a), p(b)\}$.

[5]One may ask why anyone would bother putting in the disjunction when $p(a)$ and $p(b)$ are known. Remember, however, that the facts could have been added (or learned) later. Also, in reality, the program may not be so straight-forward. The facts may be derived from other rules and seemingly-unrelated new information.

Example 2.2.4. *(Constraints)*

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

The first rule forces us to believe $p(a)$ or to believe $p(b)$. The second rule is a constraint that prohibits the reasoner's belief in $p(a)$. Therefore, the first possibility is eliminated, which leaves $\{p(b)\}$ as the only answer set of the program. In this example you can see that the constraint limits the sets of beliefs an agent can have, but does not serve to derive any new information. Later we will show that this is always the case.

Example 2.2.5. *(Default Negation)*
Sometimes agents can make conclusions based on the absence of information. For example, an agent might assume that with the absence of evidence to the contrary, a class has not been cancelled. Or, he might wish to assume that if a person does not know whether she is going to class the next day, then that day is not a holiday. Such reasoning is captured by default negation. Here are several examples.

$p(a) \leftarrow$ *not* $q(a)$.	"If $q(a)$ does not belong to your set of beliefs, then $p(a)$ must."

No rule of the program has $q(a)$ in its head and, hence, nothing forces the reasoner, which uses the program as its knowledge base, to believe $q(a)$. So, by the Rationality Principle, he does not. To satisfy the only rule of the program, the reasoner must believe $p(a)$; thus, $\{p(a)\}$ is the only answer set of the program.

Now consider the following program:

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

Clearly, $q(a)$ must be believed, i.e., must belong to every answer set of the program. This means that the body of the first rule is never satisfied and, hence, the first rule does not contribute to our construction. Since there is no rule in the program whose head contains $q(b)$, we cannot be forced to believe $q(b)$; the body of the second rule is satisfied and, hence, $p(b)$ must be believed. Thus, the only answer set of this program is $\{q(a), p(b)\}$.

If given a definition of an answer set of a program, one can easily define the notion of entailment:

Definition 2.2.1. *(ASP Entailment)*
A program Π **entails** *a literal* l *(*$\Pi \models l$*) if* l *belongs to all answer sets of* Π.

Often instead of saying that Π entails l we say that l is a **consequence** of Π.

This seemingly simple notion is really very novel and deserves careful study. To see its novelty let us recall that the entailment relation of classical logic which forms the basis for mathematical reasoning has the important property called **monotonicity**: addition of new axioms to a theory T of classical logic can not decrease the set of consequences of T. More formally, entailment relation $\models$ is called monotonic if for every A, B, and C if $A \models B$ then $A, C \models B$. This property guarantees that a mathematical theorem, once proven, stays proven. This is not the case for ASP entailment. Addition of new information to program Π may invalidate the previous conclusion. In other words for a non-monotonic entailment relation $\models$, $A \models B$ does not guarantee that $A, C \models B$. Clearly, program Π_1 consisting of the rule

$$p(a) \leftarrow \ not\ q(a)$$

entails $p(a)$ while program

$$\Pi_2 = \Pi_1 \cup \{q(a)\}$$

does not. Addition of $q(a)$ to the agent's knowledge base invalidates his previous conclusion. It forces the agent to stop believing in $p(a)$.

This feature seems to be typical for our commonsense reasoning, where our conclusions are often tentative. This quality of commonsense reasoning made a number of AI researchers doubt that the logical approach to AI would succeed. The discovery of **non-monotonic** logics in the 1980s dispelled these doubts. It is exactly this non-monotonic quality given to ASP by its unique connectives and entailment relation that makes it such a powerful knowledge representation language. Much more will be said about non-monotonicity in this book but for now we will return to our definitions.

We will use the notion of entailment to answer queries to program Π. By a **query** we will mean a conjunction or disjunction of literals. Queries not containing variables will be called ground.

Definition 2.2.2. *(Answer to a Query)*

- *The answer to a ground conjunctive query,* $l_1 \wedge \cdots \wedge l_n$*, is*
  - yes *if* $\Pi \models \{l_1, \ldots, l_n\}$,
  - no *if there is* i *such that* $\Pi \models \bar{l}_i$,
  - unknown *otherwise.*
- *The answer to a ground disjunctive query,* l_1 *or* $\ldots$ *or* l_n*, is*
  - yes *if there is* i *such that* $\Pi \models l_i$,
  - no *if* $\Pi \models \{\bar{l}_1, \ldots, \bar{l}_n\}$,
  - unknown *otherwise.*
- *An answer to a query* $q(X_1, \ldots, X_n)$*, where* $X_1, \ldots, X_n$ *is the list of variables occurring in* q*, is a sequence of ground terms* $t_1, \ldots, t_n$ *such that* $\Pi \models q(t_1, \ldots, t_n)$.

(Note that the actual reasoning system we are going to use to answer queries refers to them as *epistemic queries*, uses a comma instead of $\wedge$ for conjunction and does not directly support disjunctive queries.)

Example 2.2.6.
Consider again program Π_1 consisting of a rule

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

It has the answer set $\{p(a)\}$ and thus answers *yes* and *unknown* to queries $?p(a)$ and $?q(a)$, respectively. Query $?(p(a) \wedge q(a))$ is answered by *unknown*; $?(p(a)\ or\ q(a))$ is answered by *yes*. Query $?p(X)$ has exactly one answer $X = a$. Let's add one more rule to this program:

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

This rule is known as the **Closed World Assumption (CWA)** . It guarantees that answer sets of a program are complete with respect to the given predicate; i.e., every answer set must contain either $q(t)$ or $\neg q(t)$ for every ground term t from the signature of the program. The new program's answer set is $\{p(a), \neg q(a)\}$. This time queries $?p(a)$ and $?(p(a)\ or\ q(a))$ will still be answered by *yes* while the answers to queries $?q(a)$ and $?(p(a) \wedge q(a))$ will change to *no*.

2.2.2 Formal Semantics

We first refine the notion of consistency of a set of literals. Pairs of literals of the form $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called *contrary*. A set S of ground literals is called *consistent* if it contains no contrary literals.

All that is left is to precisely define the notion of an answer set. The definition consists of two parts. The first part of the definition is for programs without default negation. The second part explains how to remove default negation so that the first part of the definition can be applied.

Definition 2.2.3. *(Answer Sets, Part I)*
Let Π *be a program not containing default negation, i.e. consisting of rules of the form:*

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

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

- S *satisfies the rules of* Π*; and*
- S *is minimal; i.e., there is no proper subset of* S *which satisfies the rules of* Π*.*

Let's look at some examples, this time employing the formal definition and *observe that it captures our intuition.*

Example 2.2.7. *(Example 2.2.1, revisited)*
Let us now go back to Example 2.2.1 and check that our informal argument is compatible with Definition 2.2.3, namely that $S_1 = \{q(a), p(b)\}$ is the answer set of program Π_1 below:

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

S_1 satisfies fact $q(a)$ by clause (1) of the definition of satisfiability (Definition 2.1.1). Clause (4) of this definition guarantees that the (empty) body of the second rule is vacuously satisfied by S_1, and hence, by clause (5), the second rule is satisfied by S_1. Since the head $p(b)$ of the first rule is in S_1, S_1 also satisfies the first rule. Clearly S_1 is consistent and no proper subset of S_1 satisfies the rules of Π_1; therefore, S_1 is Π_1's answer set. Suppose now that S is an answer set of Π_1. It must satisfy the rules of Π_1 and, hence, S contains S_1. From minimality we can conclude that $S = S_1$; i.e. S_1 is the unique answer set of Π_1. Later we will show that any program with neither *or* nor *not* has at most one answer set.

Now that we have the answer set, we can see that, by the definition of entailment, this program entails $q(a)$ and $p(b)$. Below are the answers to some possible ground queries created from the program's signature:

? $q(a)$	*yes*
? $\neg q(a)$	*no*
? $p(b)$	*yes*
? $\neg p(b)$	*no*

We skip Example 2.2.2 (the reader is encouraged to work it out as an exercise) and consider another example.

Example 2.2.8.
Consider a program consisting of two rules:

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

Let $S_1 = \{\neg p(a)\}$. Clearly, S_1 is consistent. Since $p(b) \notin S_1$, by clause (1) of the definition satisfiability, the body of the first rule is not satisfied by S_1. Hence, by clause (5), S_1 satisfies the first rule. The body of the second rule is empty. Hence, by clause (4), it is vacuously satisfied by S_1. Thus, by clause (5), S_1 satisfies the second rule of the program. The only proper subset, $\emptyset$, of S_1 does not satisfy the second rule; therefore, S_1 is an answer set of our program. To show that S_1 is the only answer set, consider an arbitrary answer set S. Clearly, S must contain $\neg p(a)$. This means that by the minimality requirement $S = S_1$.

By the definition of entailment we can see that this program entails $\neg p(a)$. Below are the answers to some possible ground queries created from the program's signature:

? $p(a)$	*no*
? $\neg p(a)$	*yes*
? $p(b)$	*unknown*
? $\neg p(b)$	*unknown*

It may be worth noticing that this example clearly illustrates the difference between logic programming connective $\leftarrow$ and classical implication (denoted by $\supset$). Recall that $p(b) \supset p(a)$ is classically equivalent to its *contrapositive*, $\neg p(a) \supset \neg p(b)$. Hence, the classical theory consisting of $p(b) \supset p(a)$ and $\neg p(a)$ entails $\neg p(b)$. Our example shows that this is not the case if $\supset$ is replaced by $\leftarrow$. If we want the contrapositive of $p(a) \leftarrow p(b)$, we need to

explicitly add it to the program[6]. We suggest that the reader check that, as expected, the resulting program would have the answer set $\{\neg p(a), \neg p(b)\}$.

Example 2.2.9.

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

Let $S_1 = \{\neg p(a), p(b)\}$. S_1 is a consistent set of ground literals which clearly satisfies the rules of the program. To show that S_1 is an answer set we need to show that no proper subset of S_1 satisfies the program rules. To see that, it is enough to notice that $\neg p(a)$ must belong to every answer set of the program by clause (1) of the definition of satisfiability, and $p(b)$ is required by clause (5). Therefore, S_1 is minimal.

This time, the answer to query $?p(b)$ is *yes* and to $?\neg p(b)$ is *no*.

Example 2.2.10. *(Empty Answer Set)*

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

There are no facts, so we are not forced to include anything in S_1. Let's check if $S_1 = \emptyset$ is an answer set. S_1 satisfies the rule because it does not satisfy its body. Since $\emptyset$ has no proper subsets, it is the only answer set of the program. Note that an empty answer set is by no means the same as the absence of one.

Since neither $p(a) \in \emptyset$ nor $\neg p(a) \in \emptyset$, the truth of $p(a)$ is *unknown*, and similarly for $p(b)$.

Now we will look at several programs containing epistemic disjunction.

Example 2.2.11. *(Epistemic Disjunction, revisited)*
We will start with the first two programs from Example 2.2.3 from the previous section. Program

$$p(a) \text{ or } p(b).$$

has two answer sets, $\{p(a)\}$ and $\{p(b)\}$. Each contains a literal required by clause (3) of the definition of satisfiability. Note that $\{p(a), p(b)\}$ is not minimal, so it is not an answer set. Since entailment requires literals to be true in *all* answer sets of a program, this program does not entail $p(a)$ nor does it entail $p(b)$; i.e., their truth values, along with the truth values of their negative counterparts, are *unknown*.

[6] In further chapters you will see why we do not want a built-in contrapositive for our $\leftarrow$ connective.

Now consider program

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

In this case our definition clearly gives two answer sets: $\{p(a), q(a)\}$ and $\{p(b), q(a)\}$. This program entails $q(a)$. It entails no other literals (but does entail, say, the disjunction $p(a) \; or \; p(b)$).

Now let's look at a new example emphasizing the difference between epistemic and classical readings of disjunction.

Example 2.2.12. *($p(a) \; or \; \neg p(a)$ is Not a Tautology)*
Consider

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

The program has two answer sets: $S_1 = \{p(a), p(b)\}$ and $S_2 = \{\neg p(a), p(b)\}$. S_1 satisfies the first rule because it does not satisfy the rule's body, the second rule because it contains $p(a)$ and $p(b)$, and the third rule because it contains $p(a)$. S_2 satisfies the first rule because it contains both $\neg p(a)$ and $p(b)$, the second because it does not satisfy the body, and the third because it contains $\neg p(a)$. Because each literal is required to satisfy some rule, the sets are minimal.

But now let us look at the program

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

It is not difficult to check that $\emptyset$ is the only answer set of this program. This is not surprising since, as we mentioned in Section 2.1 when we discussed syntax, $p(a) \; or \; \neg p(a)$ is not a tautology. Without the presence of explicit disjunction, $p(a) \; or \; \neg p(a)$, the reasoner remains undecided about the truth value of $p(a)$. He neither believes that $p(a)$ is true nor that it is false. Thus $p(b)$ is not included in the answer set.

It may be strange that we cannot conclude $p(b)$. Indeed one may be tempted to reason as follows: Either $p(a)$ is true or $p(a)$ is false. Hence, $p(b)$ must be included in any answer set. But, of course, this reasoning is based on the wrong reading of epistemic *or* — it slides back to a classical reading of the disjunction. Since the reasoner may have no opinion on the truth values of $p(a)$, the argument fails.

Example 2.2.13. *(Constraints, revisited)*

$$p(a) \text{ or } p(b).$$
$$\leftarrow p(a).$$

The first rule is (minimally) satisfied by either $S_1 = \{p(a)\}$ or $S_2 = \{p(b)\}$. However, S_1 does not satisfy the second rule since it is not possible to satisfy an empty head if the body is satisfied. Therefore, S_2 is the only answer set of the program. Note that, although we cannot include $p(a)$ in any answer set, we do not have enough information to entail $\neg p(a)$. The answers to queries $p(a)$ and $p(b)$ are *unknown* and *yes*, respectively.

So far we have not had to address default negation. The second part of the definition of answer sets addresses this question.

Definition 2.2.4. *(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*.*

S *is an Answer Set of* Π *if* S *is an answer set of* Π^S*.*

We refer to Π^S as the **reduct** of Π with respect to S.

Example 2.2.14. *(Default Negation, revisited)*
Consider a program Π from Example 2.2.5 (see the table below). Let's confirm that $S = \{q(a), p(b)\}$ is the answer set of Π. Π has default negation; therefore, we use Part II of our definition of answer sets to compute Π^S.

	Π	Π^S
r_1	$p(a) \leftarrow not\ q(a).$	$(deleted)$
r_2	$p(b) \leftarrow not\ q(b).$	$p(b).$
r_3	$q(a).$	$q(a).$

1. We remove r_1 from Π because it has $not\ q(a)$ in its premise, while $q(a) \in S$.
2. Then, we remove the premise of r_2.

In this way, we eliminate all occurrences of default negation from Π. Clearly, S is the answer set of Π^S and, hence, of Π.

Note that, as mentioned in the definition, a reduct is always computed with respect to a candidate set of ground literals S. The algorithm for computing answer sets is not presented until Chapter 7; thus, we must still come up with candidate sets based on our intuition given by the informal semantics. (Of course, theoretically, we could test all possible sets of ground literals from the signature, but in practice this approach works only on small programs.) Meanwhile, the following proposition will be of some help for reasoning about answer sets.

Proposition 2.2.1. Let S be an answer set of a ground ASP program Π.

(a) S satisfies every rule $r \in \Pi$.

(b) If literal $l \in S$ then there is a rule r from Π such that the body of r is satisfied by S and l is the only literal in the head of r satisfied by S. (It is often said that **rule** r **supports literal** l.)

The first part of the proposition guarantees that answer sets of a program satisfy its rules; the second guarantees that every element of an answer set of a program is supported by at least one of its rules.

Here are some more examples of answer sets of programs with default negation:

Example 2.2.15.

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

This program has no answer set. This can be established by simply considering two available candidates, $S_1 = \emptyset$ and $S_2 = \{p(a)\}$. S_1 does not satisfy the above rule and hence, according to the first clause of Proposition 2.2.1, cannot be the program's answer set. The second clause of the proposition allows us to see that S_2 cannot be an answer set since $p(a)$ is not supported by any rule of the program.
Note that the absence of answer sets is not surprising. The rule, which tells the agent to believe $p(a)$ if he does not believe it, should naturally be rejected by a rational agent.

This is our first example of an ASP program without answer sets. We will refer to such programs as **inconsistent**. There are, of course, many other inconsistent programs, such as

$$p(a).$$
$$\neg p(a).$$

or

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

but inconsistency normally appears only when the program is erroneous, i.e. does not adequately represents the agent's knowledge. Later, however, we will show how various interesting reasoning tasks can be reduced to discovering the inconsistency of a program.

Example 2.2.16.

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

There are two candidate answer sets, $S_1 = \{p(a)\}$ and $S_2 = \emptyset$. Since the reduct of the program with respect to S_1 is $p(a)$, S_1 is an answer set. Clearly, S_2 does not satisfy rules of the program; thus, S_1 is the only answer set.

Example 2.2.17.

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

This program has two answer sets: $\{p(a)\}$ and $\{p(b)\}$. Note that $\emptyset$ is not an answer set for this program since it does not satisfy the program's rules. The set $\{p(a), p(b)\}$ is not an answer set of the program since its elements are not supported by the rules of the program.

Example 2.2.18.

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

The constraint eliminates $\{p(b)\}$, making $\{p(a)\}$ the only answer set of the program.

Example 2.2.19.

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

This program has no answer set. Indeed by Proposition 2.2.1 an answer set must contain $\neg p(a)$ and cannot contain $p(b)$. Hence, by the first rule of the program it should contain $p(a)$, which is impossible.

In the next example we consider a slightly more complex program.

Example 2.2.20. Let Π consist of the rules

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

After grounding the program will have the form

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

Even though the number of candidate answer sets for this program is large, the answer sets of the program can still be obtained in a relatively simple way. It is easy to see that by Proposition 2.2.1 an answer set S of the program must satisfy the first three rules. Thus, S contains $\{s(b), r(a), p(a)\}$ or $\{s(b), r(a), p(b)\}$. The program has no rule that can possibly support $s(a)$ and, hence, every answer set of the program satisfies *not* $s(a)$. This means that every answer set containing $\{p(a), r(a)\}$ must also contain $q(a)$. There is no rule that can support $r(b)$; thus, the body of the last rule cannot be satisfied and $q(b)$ cannot belong to any answer set of the program. This implies that the program may only have two answer sets, $\{s(b), r(a), p(a), q(a)\}$ and $\{s(b), r(a), p(b)\}$. Both of the candidates are indeed answer sets which can be easily checked using the definition.

So far our computation of answer sets has been done by hand. There is a large number of efficient software systems, called **ASP solvers** which automate this process. Appendix A contains instructions for downloading and using several such solvers. At this point we advise the readers to download one of these solvers, called DLV, and use it to compute several answer sets for programs from the above example. The use of other solvers will be discussed in the later chapters.

2.3 A Note on Translation from Natural Language

So far we have discussed the semantics of logical connectives of ASP. The question of translating fragments of natural language into ASP theories was

addressed only briefly. It is important to realize that, as in any translation, the translation from even simple English sentences to statements in ASP may be a non-trivial task. Consider the simple English sentence, "All professors are adults." The direct (literal) translation of this sentence seems to be

$$(1)\ adult(X) \leftarrow prof(X).$$

Used in conjunction with a list of professors, it will allow us to make conclusions about their adulthood; e.g., the program consisting of the above rule and the fact $prof(john)$ entails $adult(john)$. But what happens if we expand this program by $\neg adult(alice)$? Intuitively we should be able to conclude that $\neg prof(alice)$. The literal translation of the English statement does not allow us to do that. This happens because some information implicitly present in the English text is missing from our translation. One can argue, for instance, that the translation is missing something akin to classical logic's law of the exclusive middle with respect to professors and adults, and thus a good translation to ASP should include statements

$$(2)\ adult(X)\ or\ \neg adult(X)$$

$$(3)\ prof(X)\ or\ \neg prof(X)$$

where X ranges over some sort $person$. For simplicity we assume that there are only two persons, $john$ and $alice$. Let us denote the program consisting of the rules (1)–(3) and the sort $person$ by Π_1. One can easily check that Π_1 combined with the two facts above has one answer set

$$\{prof(john), adult(john), \neg prof(alice), \neg adult(alice)\}.$$

(We are not showing the sort $person$.) As is typical in natural language translation, this is not the only reasonable representation of our statement in ASP. One can simply translate our English statement into program Π_2 consisting of the above sort $person$ and two rules:

$$adult(X) \leftarrow prof(X)$$

$$\neg prof(X) \leftarrow \neg adult(X).$$

The first is a direct translation and the second is its contrapositive. Again the program, used together with $prof(john)$ and $\neg adult(alice)$, produces the expected answers. Note, however, that programs Π_1 and Π_2 are not equivalent; i.e., they do not have the same answer sets. Indeed, if we do not

include the sort, $\emptyset$ is the only answer set of Π_2 while Π_1 has answer sets including

$$\{prof(alice), adult(alice), prof(john), adult(john)\}$$
$$\{prof(alice), adult(alice), \neg prof(john), \neg adult(john)\}$$
$$\{\neg prof(alice), \neg adult(alice), prof(john), adult(john)\}$$
$$\{\neg prof(alice), \neg adult(alice), \neg prof(john), \neg adult(john)\}.$$

Which representation is better may depend on the context of the sentence, the purpose of our representation, and simply the taste of the translator. Any type of translation is an art and translation from English into ASP is no exception. One of the goals of this book is to help you to become better translators and to better understand the consequences of your translation decisions.

2.4 Properties of ASP Programs

In this section we will give several useful properties of ASP programs. (In the rest of this book, unless otherwise stated, we use the terms ASP program and logic program interchangeably.) We start with conditions that guarantee program consistency.

In what follows we consider programs consisting of rules of the form

$$p_0 \ or \ \ldots \ or \ p_i \leftarrow p_{i+1}, \ldots, p_m, \ not \ p_{m+1}, \ldots, \ not \ p_n \tag{2.2}$$

where ps are atoms and $i \geq 0$ (in other words, programs containing no classical negation $\neg$ and no constraints).

Definition 2.4.1. *(Level Mapping)*
Let program Π *consist of rules of form (2.2). A function* $||\ ||$ *from ground atoms of* Π *to natural numbers*[7] *is called a* **level mapping** *of* Π*. Level* $||D||$*, where* D *is a disjunction of atoms, is defined as the* minimum level of D's members.

Definition 2.4.2. *(Local Stratification)*
A program Π *consisting of rules of form (2.2) is called* **locally stratified** *if there is a level mapping* $||\ ||$ *of* Π *such that for every rule* $r \in \Pi$*, the following is true:*

[7]For simplicity we consider a special case of the more general original definition which allows arbitrary countable ordinals.

1. *for every* p_k *where* $i < k \leq m$, $||p_k|| \leq ||head(r)||$; *and*

2. *for every* p_k *where* $m < k \leq n$, $||p_k|| < ||head(r)||$.

It is easy to see that any program without classical and default negation is locally stratified. A function mapping all atoms of such a program into, say, 0 is a level mapping which satisfies the corresponding conditions. A function $||p(a)|| = ||r(a)|| = 1$ and $||q(a)|| = 0$ is a level mapping of a program consisting of rule

$$p(a) \leftarrow \ not\ q(a), r(a).$$

Clearly the mapping satisfies the conditions from the definition and, hence, the program is locally stratified.

Proposition 2.4.1. *(Properties of Locally Stratified Programs)*

- *A locally stratified program is consistent.*
- *A locally stratified program without disjunction has exactly one answer set.*
- *The above conditions hold for the union of a locally stratified program and any collection of closed world assumptions, i.e. rules of the form*

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

The proposition immediately implies the existence and uniqueness of answer sets of programs from Examples 2.2.5 and 2.2.6. It is, however, not applicable to the program from Example 2.2.15.

Let's consider two more examples of locally stratified programs.

Example 2.4.1.
Consider a program

$$p(0).$$
$$p(f(I)) \leftarrow p(I).$$

It is not difficult to see that the minimal set of ground literals satisfying rules of this program is an infinite set

$$S = \{p(0), p(f(0)), p(f(f(0))), \dots\}$$

The program contains no negations and, hence, is locally stratified. Therefore, it has no answer sets except S.

The next example is only slightly more sophisticated.

Example 2.4.2.
Consider the program Π

$$p(0).$$
$$p(f(I)) \leftarrow \ not\ p(I).$$

and a set

$$S = \{p(0), p(f(f(I))), p(f(f(f(f(I))))), \dots\}.$$

Using the definition one can easily prove that S is an answer set of Π. Clearly, Π is locally stratified. (Consider a level mapping assigning each atom $||p(t)||$ the number of fs occurring in t and check that it satisfies the corresponding conditions.) Therefore, S is the only answer set of the program.

Another approach to proving consistency and, sometimes, uniqueness of a logic program is based on the notion of the program's dependency graph. Let Π be a (not-necessarily) ground program consisting of rules of form (2.2). A **dependency graph** of Π is a collection of nodes labeled by predicate symbols from the signature of Π and a collection of arcs of the form $\langle p_1, p_2, s\rangle$ where s is $+$ (positive link) or $-$ (negative link). The graph contains $\langle p_1, p_2, +\rangle$ if there is a rule of Π containing an atom formed by p_1 in the head and an atom formed by p_2 in the body; it contains $\langle p_1, p_2, -\rangle$ if there is a rule of Π containing an atom formed by p_1 in the head and an extended literal of the form $not\ l$ where l is formed by p_2 in the body. Note that two nodes can be connected by both positive and negative links. A cycle in the graph is said to be negative if it contains at least one edge labeled by $-$. A program is called **stratified** if its dependency graph contains no negative cycles.

Proposition 2.4.2. *Let* Π *be a program consisting of rules of form (2.2).*

- *If the dependency graph of* Π *contains no cycles with an odd number of negative links then* Π *is consistent, i.e. has an answer set.*
- *A stratified program without disjunction has exactly one answer set.*

The proposition can be used to prove consistency of programs

$$p(X) \leftarrow \ not\ q(X).$$
$$q(X) \leftarrow \ not\ p(X).$$
$$r(0).$$

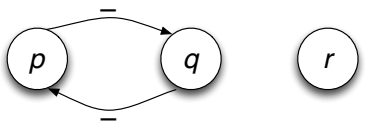


Figure 2.1: Dependency Graph

and

$$p(X) \leftarrow \ not\ q(X).$$
$$r(0).$$

The first program has a dependency graph with an even number of negative cycles — see Figure 2.1. The second is stratified, thus having exactly one answer set. Despite substantial differences in their definitions, there is a close relationship between stratified and locally stratified programs. In fact every stratified program is locally stratified.

The following proposition is useful for understanding the role of constraints.

Proposition 2.4.3. *Let* Π *be a logic program consisting of a collection* R *of rules with non-empty heads and a collection* C *of constraints. Then* S *is an answer set of* Π *iff* S *is an answer set of* R *which satisfies constraints from* C*.*

Last, we describe a procedure for removing negative literals and constraints from a program without changing its answer sets. For every predicate symbol p from the signature of Π:

1. Introduce a new predicate symbol p^+.

2. Replace every occurrence of a negative literal of the form $\neg p(\bar{t})$ in the program by a new, positive literal $p^+(\bar{t})$. We call this literal the *positive form* of $\neg p(\bar{t})$.

3. For every sequence of ground terms $\bar{t}$ which can serve as a parameter of p, expand Π by the axiom

$$\leftarrow p(\bar{t}), p^+(\bar{t})$$

4. Replace every constraint

$$\leftarrow body$$

of the program by rule

$$p \leftarrow body,\ not\ p$$

where p is a new atom.

It is not difficult to show that the new program Π^+, which contains neither classical negation nor constraints, has the following property:

Proposition 2.4.4. *Let S be a set of literals over signature Σ of Π. By S^+ we denote the set obtained by replacing negative literals of S by their positive forms. A consistent set S of literals from Σ is an answer set of Π iff S^+ is an answer set of Π^+.*

ASP program!properties of—)

Summary

In this chapter we introduced our main knowledge representation tool — Answer Set Prolog. The logical connectives and the semantics of the language capture the intuition behind a particular notion of beliefs of rational agents as described by the Rationality Principle.

We hope that the reader agrees that the language satisfies the following important principles of good design: it has a simple syntax, its logical connectives have reasonably clear intuitive meaning, and the definition of its semantics is mathematically simple and transparent. The entailment relation of the language is non-monotonic which makes ASP dramatically different from the language of classical logic. In the following chapters we show the importance of this property for knowledge representation.

The chapter continues with a brief discussion of the subtleties of the translation of natural language texts into Answer Set Prolog — a theme which permeates this book.

It concludes by describing several simple but important mathematical properties of the language. The first two give sufficient conditions guaranteeing existence and uniqueness of answer sets of the programs. The last one illustrates the role of constraints and the means of limiting an agent's beliefs without generating new ones.

References and Further Reading

The syntax and semantics of Answer Set Prolog was introduced in [50]. The notion of stratification and its relation to dependency graphs was introduced

in [5]. Local stratification first appeared in [94]. The properties of stratified and locally stratified programs were proven in these papers. There are many interesting extensions to these notions (see for instance [93]). Clause 1 of Proposition 2.4.2 is a simple special case of a more general result from [39]. There is a substantial number of generalizations of Answer Set Prolog which allow rules whose heads and bodies contain more-complex formulas (see, for instance, [40] and [87], [88]). Some of the other generalizations of the language will be described in later chapters. The Closed World Assumption and its importance for knowledge representation and reasoning was first discussed in [97]. (Representation of CWA is Answer Set Prolog is from [50]). The first efficient ASP system computing answer sets of non-disjunctive logic programs is described in [84] and [83]. For a description of the corresponding system for programs with disjunction, see [64] and [24].

Exercises

1. Given the following signature

$$\begin{aligned}\mathcal{O} &= \{a\}\\ \mathcal{F} &= \{f\}\\ \mathcal{P} &= \{p\}\\ \mathcal{V} &= \{X\}\end{aligned}$$

specify which of the following are terms, atoms, literals, or none.

(a) a	(f) $\neg f(a)$	(k) $\neg p(a)$
(b) $\neg a$	(g) $f(p)$	(l) $p(f(a))$
(c) X	(h) $f(X)$	(m) $p(\neg f(a))$
(d) f	(i) p	(n) $\neg p(f(a))$
(e) $f(a)$	(j) $p(a)$	(o) $p(X)$

2. Given program Π:

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

and set $S = \{p(c)\}$

   (a) Construct the program Π^S from the definition of answer set.

   (b) Check if S is an answer set of Π. Justify your answer.

3. (a) Compute the answer sets of the following program:

$$p \text{ or } q \text{ or } r.$$
$$\neg p \leftarrow \text{not } s.$$

   (b) How does the above program answer queries $?p$ and $?q$?

4. Compute the answer sets of the following program:

$$p(a) \leftarrow \text{not } p(b), \neg p(c).$$
$$p(b) \leftarrow \text{not } p(a), \neg p(c).$$
$$\neg p(X) \leftarrow \text{not } p(X).$$

5. Compute the answer sets of the following program:

$$p \leftarrow \text{not } q.$$
$$q \leftarrow \text{not } p.$$
$$r \leftarrow \text{not } s.$$
$$s \leftarrow \text{not } r.$$
$$\neg s \leftarrow q.$$

6. (a) Compute the answer sets of the following program. *Assume that a and b are the object constants of this program's signature.*

$$\neg s(a).$$
$$p(X) \leftarrow \text{not } q(X), \neg s(X).$$
$$q(X) \leftarrow \text{not } p(X).$$
$$r(X) \leftarrow p(X).$$
$$r(X) \leftarrow q(X).$$

   (b) How does the program answer queries $?s(a)$, $?r(a)$, $?s(b)$, and $?q(b)$?

7. (a) Compute the answer sets of the following program:

$$p(a) \text{ or } \neg p(b).$$
$$q(X) \leftarrow \neg p(X).$$
$$\neg q(X) \leftarrow \text{not } q(X).$$
$$r(X) \leftarrow \text{not } p(X).$$

   (b) How does the program answer queries $?q(a)$, $?r(a)$, $?q(b)$ and $?r(b)$?

8. (a) Compute the answer sets of the following program:

$$p(X) \; or \; q(X) \leftarrow \; not \; r(X).$$
$$\neg p(X) \leftarrow h(X), \; not \; r(X).$$
$$h(a).$$
$$h(b).$$
$$r(a).$$

   (b) How does the program answer queries
   $?p(b)$, $?q(b)$ and $?r(b)$?

9. (a) Compute the answer sets of the following program:

$$p(a) \leftarrow \; not \; p(b).$$
$$p(b) \leftarrow \; not \; p(a).$$
$$q(a).$$
$$\neg q(b) \leftarrow p(X), \; not \; r(X).$$

   (b) How does the program answer queries
   $?q(a)$, $?q(b)$, $?p(a)$ and $?r(b)$?

10. Translate the following story about dealing with high prices into ASP. Ignore the time factor. "Either the supply was increased or price controls were instituted. Instituting price controls leads to shortages. There are no shortages." Use zero-arity predicate symbols *increased_supply*, *price_controls* and *shortages*. Compute the answer sets of this program. How does your program answer queries

$$? \; increased\_supply$$

$$? \; increased\_supply \wedge \neg price\_controls$$

    Is that what you intended? (Note that the answer depends on your understanding of the disjunction in the first sentence of the story. Is it inclusive or exclusive?)

11. Consider the following story. "If Jim does not buy toys for his children, Jim's children will not receive toys for Christmas. If Jim's children do not write their Christmas letters, Jim will not buy them toys. Jim's children do receive toys for Christmas." Assume that the intended interpretation of this story implies that Jim's children wrote their Christmas letters.

(a) Translate the story into an ASP program and compute the answer set. Use disjunction to encode the law of the exclusive middle to allow the program to come to the proper conclusion.

(b) Translate the story into an ASP program and compute the answer set, this time making the contrapositive explicit for each statement.

Chapter 3

Roots of ASP

Answer Set Prolog is a comparatively new knowledge representation (KR) language with roots in older non-monotonic logics and the logic programming language Prolog. Early proponents of the logical approach to Artificial Intelligence believed that the classical logical formalism called *first-order logic* would serve as the basis for the application of the axiomatic method to the development of intelligent agents. In this chapter we briefly describe some important developments which forced them to question this belief and to work on the development of non-classical knowledge representation languages including ASP. To make the reading easier for people not familiar with mathematical logic, we give a very short introduction to one of its basic logical tools — first-order logic.

3.1 First-Order Logic (FOL)

First-order logic is a formal logical system which consists of a formal language, an entailment or consequence relation for this language, and a collection of inference rules which can be used to obtain these consequences. The language of FOL is parametrized with respect to a signature Σ. The notions of term and atom over Σ are the same as those defined in Section 2.1. The statements of FOL (called FOL formulas) are built from atoms using boolean logical connectives and quantifiers $\forall$ (for all) and $\exists$ (there exists). Atoms are formulas. If A and B are formulas and X is a variable then $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, $\neg A$, $\forall X\, A$, $\exists X\, A$ are formulas. An occurrence of a variable X in a formula A is called *bound* if it belongs to a sub-formula of A which has the form $\forall X\, F$ or $\exists X\, F$; otherwise it is free. Formulas without free occurrences of variables are called *sentences*. $\forall X\, p(X)$ is a sentence;

$p(X)$ is an FOL formula that is not a sentence. Sometimes, the latter are referred to as *conditions* on X. To simplify further presentation, we assume that for every variable X, all occurrences of X in a formula are either free or bound, exclusively. This will eliminate formulas like $(\forall X(p(X) \vee q(X))r(X))$ where the last occurrence of X is free and the first two are bound. A set of FOL sentences is often referred to as an FOL *theory*.

To illustrate, let's take our family example from Chapter 1. Its FOL signature, atoms, and terms are identical to the those given in Section 2.1. Here are some formulas made from the signature's symbols:

$$father(john, sam) \wedge (mother(alice, sam) \vee mother(alice, john))$$

$$\forall X, Y\ (parent(Y, X) \supset child(X, Y))$$

$$\neg \exists X\ gender(X, male)$$

$$gender(X, male)$$

The first three are sentences. The last one is a condition on X.

Note that so far we only talked about the syntax of FOL. Nothing was said about the semantics (meaning) of these formulas and their truth or falsity. The meaning of a first-order formula depends on the interpretation of its nonlogical symbols. Such an interpretation is normally defined by specifying a non-empty set U called the *universe* and a mapping I which maps

- every object constant c of Σ into an element $I(c)$ of U,
- every function constant f of arity n into a function $I(f) : U^n \to U$,
- every predicate constant p of arity n into a subset $I(p)$ of U^n.

Using the first two clauses of the above definition, function I can be naturally extended to arbitrary terms. Whenever convenient we will refer to an interpretation by its second element I.

To illustrate this notion let us consider a family example above, and define an interpretation, say I_1, of its symbols.

Suppose that our universe U_1 of I_1 consists of a group of four people and a group of two genders. Let us denote elements of U_1 by $d_1, \ldots, d_4, m, f$ where the ds stand for people and m and f for genders. (Note that these symbols do not belong to the signature of our FOL theory. Sometimes they are referred to as *meta-symbols*.) Mapping of object constants into elements of the universe can be viewed as naming these elements within the language. Making the value of $I_1(alice)$ to be d_1 simply means that in our interpretation,

the first person in the group has the name *alice*. Similarly for other constants. Let us assume that $I_1(john) = d_2$, $I_1(sam) = d_3$, $I_1(female) = f$, and $I_1(male) = m$. Let us also assume that our interpretation I_1 maps predicate *father* of arity 2 into the set $\{\langle d2, d3\rangle\}$. Intuitively this means the only people in the group which satisfy the father-child relation are d_2 and d_3. Similarly for *mother* and *gender*.

This is of course not the only possible interpretation. Interpretation I_2 may have the same universe and differ from I_1 only in its mapping of *father*: $I_2(father) = \{\langle d_4, d_3\rangle\}$. Other interpretations may have completely different universes and mappings.[1]

Now we can define the truth value of an FOL sentence in interpretation I with universe U. First we expand the signature Σ by a collection of constant symbols, one for each element of the universe U; let's say that for each $d \in U$ the constant symbol c_d is added. The interpretation is extended so that each new constant symbol is assigned to its corresponding element of the domain.

Next we define the truth of sentences by induction on the definition of a sentence. A ground atom $p(t_1, \ldots, t_n)$ of the new signature is said to be true in I if $\langle I(t_1), \ldots, I(t_n)\rangle \in I(p)$. If A and B are FOL sentences then $\neg A$ is true in I if A is not true in I; $A \wedge B$ is true in I if both A and B are true in I; $\forall\ X\ A$ is true in I if for every $d \in U$, $A(c_d)$ is true in I. (Note that here by $A(c_d)$ we denote the result of replacing all occurrences of X in A by c_d.) An interpretation I satisfies a sentence F or is a *model* of F ($I \models F$) if F is true in I. An FOL theory T is *satisfiable* (or *consistent*) if it has a model. Finally, a sentence F is a *consequence* of theory T ($T \models F$) if F is true in all the models of T; $T \models F$ is often referred to as the *FOL entailment relation* and is read as T entails F. In what follows we will often use a version of first-order logic whose syntax includes a special relation $=$. An interpretation I satisfies $t_1 = t_2$ iff I maps t_1 and t_2 to the same element of U.

By this definition, under interpretation I_1, $father(john, sam)$ is true, $father(sam, john)$ is false, $father(john, sam) \vee father(sam, john)$ is true, $\exists X\ father(X, sam)$ is true, $\forall X\ father(X, sam)$ is false, and so on. Under I_2, these formulas' truth values would become false, true, true, false, false, respectively.

[1] Note that this view of the universe is quite different from that which we intuitively use in logic programming where we normally have one (intended) universe. In our example the intended universe will contain exactly three people and two genders (each uniquely named by the corresponding constant in the language). In some cases the ability to have arbitrary universes is very important. This is especially true for representing mathematical knowledge.

Now suppose we have the following theory:

$$father(john, sam)$$
$$\neg father(sam, john)$$

You can see that this theory is satisfiable because there exists an interpretation, for example I_1, in which all its sentences are true. Note that this theory entails $\neg(father(john, sam) \wedge father(sam, john))$ but does not entail $\neg(father(alice, sam))$ because there is an interpretation in which the parameters of predicate *father* are assigned this way.

In addition to the definition of formula and consequence relation, first-order logic normally includes a collection of inference rules — syntactic rules which usually consist of premises and a conclusion. The inference rules of first-order logic are *sound*; i.e., they preserve the truth of first-order sentences. Probably the most famous example of such a rule is *modus ponens*:

$$\frac{A,\ A \supset B}{B}$$

which says that if A is true and $A \supset B$ is true then B is also true. We say that a sentence F is derived from a theory T using a collection of inference rules R ($T \vdash_R F$) if there is a sequence of sentences $F_1, \ldots, F_n$ such that $F_n = F$ and every F_i is an axiom of T or obtained from some previous elements of the sequence by an inference rule from R. We refer to such a sequence as a *derivation* of F from T in R. A collection R of first-order inference rules is called *complete* if for every theory T and formula F, $T \models F$ iff $T \vdash_R F$. There is a number of complete collections of inference rules of first-order logic. In Chapter 12 we will discuss a logic programming version of an inference rule, called *resolution*, which is often used in automated reasoning systems.

This theoretical machinery allows us to refine our notion of the axiomatic method. We can view axioms as a theory T — a collection of FOL sentences. A sentence F is a consequence of T if $T \models F$. A mathematical proof of F can be viewed as shorthand for a derivation of F from T using some complete collection of inference rules. First-order logic provides a powerful tool for knowledge representation and reasoning. It has been shown to be sufficient for formalization of a very large part of contemporary mathematical knowledge and reasoning.

3.2 Non-Monotonic Logics

Surprisingly, further experience of applying the logic-based approach to the design of intelligent agents showed that first-order logic may not be a fully adequate tool for representing nonmathematical (especially commonsense) knowledge. The main problem is the difficulty of dealing with defeasible (or non-monotonic) reasoning. Many researchers agree with the Stanford Encyclopedia of Philosophy which states:

> One of the most significant developments both in logic and artificial intelligence is the emergence of a number of non-monotonic formalisms, which were devised expressly for the purpose of capturing defeasible reasoning in a mathematically precise manner.

Among the pioneers of the field in the late 1970s were John McCarthy, Drew McDermott and Jon Doyle, and Raymond Reiter. An influential logical system developed by McCarthy is called **circumscription**. It is based on classical second-order logic and allows elegant model-theoretic characterization. McDermott and Doyle based their *non-monotonic logics* on modal logics capturing the notion of belief. (A few years later Robert Moore introduced a particular logic of this type, called **autoepistemic logic** which, among many other important things, served as a starting point for the development of stable model semantics of original Prolog. This semantics was the precursor of answer set semantics for more general logic programs we use in this book.) Reiter's formalism called **default logic** expands classical logic by allowing defeasible rules somewhat similar to those of ASP. The papers on circumscription, McDermott and Doyle's non-monotonic logics, and default logic appeared in the same issue of the Artificial Intelligence Journal (vol. 13, 1980) dedicated to these new formalisms. Many people view this event as the birth of non-monotonic logic. In what follows we give a brief introduction to these formalisms.

3.2.1 Circumscription

Circumscription is a non-monotonic logic based on the notion of *minimal entailment*. The basic idea is as follows. Let Σ be a first-order signature and $<$ be a partial order defined on interpretations of Σ. A model M is a *$<$-minimal model* of a first-order theory T if there is no other model M_0 of T such that $M_0 < M$. We say that T minimally entails formula F with respect to $<$ (or that $<$-circumscription of T entails F) and write $T \models_{min(<)} F$ or $circ(T, <) \models F$, if F is true in all *$<$-minimal* models of T.

Here is a simple example. Let Σ be a signature containing two object constants a and b and predicate constants p and q, and let T be a theory consisting of axioms:

$$a \neq b$$
$$\forall X(X = a \vee X = b)$$
$$p(a)$$
$$q(a)$$

and let $<_p$ be a partial order on interpretations of Σ defined as follows: $I_1 <_p I_2$ if I_1 and I_2 have the same universe and the same mapping of terms, and $I_1(p) \subset I_2(p)$. We show that

$$T \models_{<_p} \neg p(b)$$

$$T \not\models_{<_p} \neg q(b)$$

$$T \models_{<_p} \forall X\, X \neq b \supset q(X).$$

We start by finding $<_p$-minimal models of T. To satisfy the first two axioms of T, the universe of model M should consist of exactly two elements. To satisfy the third axiom, $M(p)$ should be equal to $\{M(a)\}$ or $\{M(a), M(b)\}$. However, the latter model is not minimal with respect to $<_p$ and should be discarded. $M(q)$ should be equal to $\{M(a)\}$ or $\{M(a), M(b)\}$. Since minimization of q is not required by relation $<_p$ both models are minimal. In more detailed notation a model of T has the universe consisting of two distinct elements, say, e_1 and e_2, with constants a and b mapped into these elements, e.g. $U = \{e_1, e_2\}$, $M(a) = e_1$ and $M(b) = e_2$. The corresponding $<_p$-minimal models M_1 and M_2 are obtained from M as follows:

$M_1(p) = \{e_1\}$
$M_1(q) = \{e_1\}$

and

$M_2(p) = \{e_1\}$
$M_2(q) = \{e_1, e_2\}$.

Note that strictly speaking we have an infinite collection of such models since $M(a)$ can be equal to e_2 and $M(b)$ to e_1 and, moreover, elements of U can be arbitrary pairs. But it can be easily shown that such models are isomorphic to M_1 and M_2 and can be ignored.

This proves the above entailments. Of course, if we expand our theory by $p(b)$ the new theory will entail $p(b)$. One can check that

$$T \cup \{p(b)\} \models_{<_p} p(b);$$

i.e., $p(b)$ is a consequence of the new theory. As expected, the new entailment relation is non-monotonic.

It may be instructive to see what happens if we remove from T some of its equality axioms. Let T_1 be obtained from T by removing the first axiom. The new theory has a model M whose universe consists of one element, say e; the model maps both constants of the language into this element — $M(a) = M(b) = e$; both p and q are mapped into $\{e\}$, i.e. $M(p) = M(q) = \{e\}$. This implies that

$$T \not\models_{<_p} \neg p(b).$$

Now let T_2 be obtained from the original theory T by removing the second axiom. T_2 has a $<_p$-minimal model M with the universe $U = \{a, b, c\}$, $M(a) = a$, $M(b) = b$, $M(p) = \{a\}$ and $M(q) = \{a, b\}$. This time we have

$$T \not\models_{<_p} \forall X\,(X \neq b \supset q(X)).$$

Circumscription is a powerful non-monotonic formalism which remains the language of choice for a number of researchers interested in knowledge representation and reasoning. Its minimality idea clearly influenced the development of ASP as well as other non-monotonic formalisms. Recent work established a close mathematical connection between circumscription and some powerful generalizations of ASP, but the full comprehension of the meaning of these results may require some additional non-trivial insights.

3.2.2 Autoepistemic Logic

Formulas of **autoepistemic logic** are built from propositional atoms using propositional connectives and the modal operator B. For instance, formula $Bp \supset p$ says that if p is believed then p is true, etc. The semantics of autoepistemic logic is given via a notion of stable expansion.

Definition 3.2.1. *(Stable Expansion)*
For any sets T and E of autoepistemic formulas, E is said to be a **stable expansion** *of T iff*

$$E = Cn(T \cup \{B\phi : \phi \in E\} \cup \{\neg B\psi : \psi \notin E\})$$

where Cn is a propositional consequence operator.

Intuitively, T is a set of axioms and E is a possible collection of reasoner's beliefs determined by T. A formula F is said to be *true* in T (or entailed by T) if F belongs to all stable expansions of T. If T does not contain the modal operator B, T has a unique stable expansion denoted by $Th(T)$. For instance, if $T = \{p, q \vee r\}$ then $Th(T)$ contains T together with an infinite collection of other formulas including Bp, $\neg Bq$, $\neg Br$, $B(q \vee r)$, $B\neg Bq$, etc. There is a close connection between classes of autoepistemic theories and classes of theories of ASP. We describe one such connection which served as a starting point for the development of stable model semantics of logic programs.

Consider a class G of programs of Answer Set Prolog which consists of rules of the form:

$$\begin{array}{ll} (i) & p_0 \leftarrow p_1, \ldots, p_m, \ not\ p_{m+1}, \ldots, \ not\ p_n \\ (ii) & \neg p \leftarrow \ not\ p \quad \text{(for every atom } p\text{)} \end{array} \tag{3.1}$$

where $0 \leq m \leq n$ and the ps are atoms. Let α be a mapping which maps rules (i) and (ii) into autoepistemic formulas:

$$\begin{array}{ll} \alpha(i) & p_1 \wedge \cdots \wedge p_m \wedge \neg B\, p_{m+1} \wedge \cdots \wedge \neg B\, p_n \supset p_0 \\ \alpha(ii) & \neg B\, p \supset \neg p \end{array} \tag{3.2}$$

and let

$$\alpha(\Pi) = \{\alpha(r) : r \in \Pi\}.$$

Proposition 3.2.1. *For any program $\Pi \in G$, and any set A of literals in the language of Π, A is an answer set of Π iff $Th(A)$ is a stable expansion of $\alpha(\Pi)$. Moreover, every stable expansion of $\alpha(\Pi)$ can be represented in the above form.*

A similar proposition proven in 1987 for stratified logic programs showed that, at least in a simple case, default negation can be interpreted as an epistemic operator. This connection played an important role in the later development of stable model semantics. There are other interesting mappings of programs of Answer Set Prolog into autoepistemic logic and its variants, but none of the suggested mappings seem to provide a really good explanation of the meanings of the *or* and $\leftarrow$ connectives of Answer Set Prolog in terms of autoepistemic logic.

3.2.3 Reiter's Default Theories

A Reiter's *default* is an expression of the form

$$\frac{p : M\ j_1, \ldots, M\ j_n}{f} \tag{3.3}$$

where p, f and js are quantifier-free first-order formulas; f is called the *consequent* of the default, p is its *prerequisite*, and js are its *justifications*. A default may have no prerequisite or no justification. An expression $M\, j$ is interpreted as "it is consistent to believe j". A pair $\langle D, W\rangle$ where D is a set of defaults and W is a set of first-order sentences is called Reiter's **default theory**.

Definition 3.2.2. *(Extension of a Default Theory)*
Let $\langle D, W\rangle$ be a default theory and E be a set of first-order sentences. Consider $E_0 = W$ and, for $i \geq 0$, let D_i be the set of defaults of form (3.3) from D such that $p \in E_i$ and $\neg j_1 \notin E, \ldots, \neg j_n \notin E$. Finally, let $E_{i+1} = Th(E_i) \cup \{conseq(\delta) : \delta \in D_i\}$ where $Th(E_i)$ is the set of all classical consequences of E_i and $conseq(\delta)$ denotes δ's consequent. The set E is called an **extension** *of $\langle D, W\rangle$ if*

$$E = \bigcup_{0}^{\infty} E_i.$$

Extensions of a default theory D play a role similar to that of stable expansions of autoepistemic theories. The simple mapping α from programs of ASP without disjunction to default theories identifies a rule r

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

with the default $\alpha(r)$

$$\frac{l_1 \wedge \cdots \wedge l_m : M\ \bar{l}_{m+1}, \ldots, M\ \bar{l}_n}{l_0} \tag{3.4}$$

(Recall that $\bar{l}$ stands for the literal complementary to l.)

Proposition 3.2.2. *For any non-disjunctive program Π of ASP*

(i) *if S is an answer set of Π, then $Th(S)$ is an extension of $\alpha(\Pi)$;*

(ii) *for every extension E of $\alpha(\Pi)$ there is exactly one answer set S of Π such that $E = Th(S)$.*

Thus, the class of non-disjunctive ASP programs can be identified with the class of default theories with empty W and defaults of the form (3.4). Perhaps somewhat surprisingly, the proposition is not easily generalized to a program with disjunction. One of the problems in finding a natural translation from arbitrary ASP programs to default theories is related to these

theories' inability to use defaults with empty justifications in reasoning by cases. The default theory with

$$D = \left\{ \frac{q :}{p}, \quad \frac{r :}{p} \right\}$$

and

$$W = \{q \vee r\}$$

does not have an extension containing p and, therefore, does not entail p. The corresponding logic program

$$\begin{array}{l} p \leftarrow q \\ p \leftarrow r \\ q \; or \; r. \end{array}$$

has two answer sets, $\{p, q\}$ and $\{p, r\}$, and hence entails p.

3.3 ASP and Negation in Logic Programming

In the late seventies and early eighties, different approaches to non-monotonicity were investigated in the area of logic programming. The goal was to give a declarative semantics for **negation as failure**[2] of logic-based programming language Prolog. The new connective was initially defined in procedural terms referring to a particular inference mechanism of Prolog called SLDNF resolution (for more details, see Chapter 12). The statement *not a* is viewed as true if SLDNF resolution finitely fails to prove a. The attempts to find suitable declarative semantics of this non-monotonic connective, together with the work on the general non-monotonic reasoning formalisms discussed above, played a major role in the discovery of ASP and several other logic-programming-based KR formalisms. In this section we briefly outline some important developments in this area. In what follows we limit our attention to logic programs consisting of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \; not \; a_{m+1}, \ldots, \; not \; a_n \tag{3.5}$$

where the as are atoms. For historical reasons we refer to such programs as **normal logic programs** or *nlp*s.

[2] In other parts of this book, we refer to this as default negation. In this historical context, we have left the name as is.

3.3.1 Clark's Completion

The research on finding a declarative semantics for negation as failure in *nlp*s started with the pioneering work of Keith Clark. He suggested that, given an *nlp*, we could view the bodies of rules with a predicate p in their heads as "sufficiency" conditions for inferring atoms formed by p from the program. Clark stated that the bodies of these rules could also be taken as "necessary" conditions, with the result that negative information about p could be assumed if all these conditions were not met. More precisely, let us consider the following two-step transformation of an *nlp* Π into a collection of first-order formulas:

Step 1: For every *nlp* rule (3.5) in Π let $a_0 = p(t_1, \ldots, t_k)$ and $Y_1, \ldots, Y_s$ be the list of variables appearing in r. By $\alpha_1(r)$ we denote an FOL formula:

$$\begin{array}{l} \exists\, Y_1 \ldots Y_s : X_1 = t_1 \wedge \cdots \wedge X_k = t_k \wedge \\ a_1 \wedge \cdots \wedge a_m \wedge \neg a_{m+1} \wedge \cdots \wedge \neg a_n \ \supset\ p(X_1, \ldots, X_k) \end{array} \tag{3.6}$$

where, $X_1 \ldots X_k$ are variables not appearing in r.

$$\alpha_1(\Pi) = \{\alpha_1(r) : r \in \Pi\}$$

Step 2: For each predicate p rename its variables to make all the implications in $\alpha_1(\Pi)$ be of the form

$$\begin{array}{l} E_1 \supset p(X_1, \ldots, X_k) \\ \vdots \\ E_j \supset p(X_1, \ldots, X_k) \end{array}$$

Next, replace these formulas by

$$\forall\, X_1 \ldots X_k : p(X_1, \ldots, X_k) \equiv E_1 \vee \cdots \vee E_j$$

if $j \geq 1$ and by

$$\forall\, X_1 \ldots X_k : \neg q(X_1, \ldots, X_k)$$

if $j = 0$.

Definition 3.3.1. *(Clark's Completion)*
The resulting first-order theory combined with natural axioms for equality called free equality axioms[3] *is called* **Clark's completion** *of* Π *and is denoted by* $Comp(\Pi)$. *A literal* l *is* entailed *by* Π *if* l *is the first-order consequence of* $Comp(\Pi)$.

[3]In addition to the usual equality axioms, this includes

To better understand the construction let us consider the following example: Let Π be the program

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

Then $\alpha_1(\Pi)$ has the form, say,

$$\begin{array}{l} X_1 = a \supset p(X_1) \\ X_2 = b \supset p(X_2) \\ \exists\, Y(X = Y \wedge p(X)) \supset q(X) \\ \neg s \supset r. \end{array}$$

It is easy to check that the third axiom is equivalent to

$$p(X) \supset q(X).$$

To properly prepare for Step 2 in which we collect our formulas under the universal quantifier, we replace X_1 and X_2 in the first two axioms by, say, X. The result, $Comp(\Pi)$ will be

$$\begin{array}{l} \forall X : p(X) \equiv X = a \vee X = b \\ \forall X : q(X) \equiv \exists\, Y(X = Y \wedge p(X)) \\ r \equiv \neg s \\ \neg s. \end{array}$$

The second formula can be simplified to

$$q(X) \equiv p(X).$$

The following theorem establishes the relationship between models of Clark's completion of Π and the notion of **supported model**. (A set S of atoms is supported by an *nlp* Π if, for every $a \in S$ there is a rule (3.5) such that $a = a_0$, $a_1, \ldots, a_m \in S$ and $a_{m+1}, \ldots, a_n \notin S$.)

- $f(X_1, \ldots, X_n) \neq g(X_1, \ldots, X_n)$ for each distinct pair of function symbols f and g of arity n.
- $t(X) \neq X$ for each term $t(X)$ (other than X) in which X occurs.
- $f(X_1, \ldots, X_n) = f(Y_1, \ldots, Y_n) \supset X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n$ for each n-ary function symbol f.

These axioms guarantee that ground terms are equal iff they are identical.

Theorem 1. *A set S of atoms is a model of Clark's completion of* Π *iff S is supported and satisfies the rules of* Π.

Models of Clark's completion may obviously differ from answer sets of Π. Program $p \leftarrow p$ has two Clark's models, $\{\ \}$ and $\{p\}$, but only one answer set $\{\ \}$. This should not be surprising — the completion semantics intends to capture the notion of finite failure of a particular inference mechanism, SLDNF resolution, while answer set semantics formalizes the more general notion of default negation. It is also important to note that the above theorem immediately implies that every answer of an *nlp* program Π is also a model of the completion of Π and hence every literal entailed by Π with respect to Clark's semantics is also entailed by Π with respect to the answer set semantics.

The existence of Clark's declarative semantics facilitated the development of the theory of logic programs. It made possible the first proofs of correctness of the inference mechanism of Prolog based on SLDNF resolution, proofs of program equivalence and some other properties of programs. It is still widely and successfully used for logic programming applications. Unfortunately in many situations Clark's semantics appears too weak. Consider for instance the following example:

Example 3.3.1. Suppose that we are given a graph, say,

$$edge(a, b). \quad edge(c, d). \quad edge(d, c).$$

and want to describe vertices of the graph reachable from a given vertex a. The natural solution seems to be to introduce the rules:

$$\begin{array}{lcl} reachable(a). & & \\ reachable(X) & \leftarrow & edge(Y, X), \\ & & reachable(Y). \end{array}$$

We clearly expect vertices c and d not to be reachable. However, Clark's completion of the predicate 'reachable' gives only

$$reachable(X) \equiv (X = a \vee \exists\, Y : reachable(Y) \wedge edge(Y, X))$$

from which such a conclusion cannot be derived.

The difficulty was recognized as serious and prompted the development of other logic programming semantics, including that of ASP. Even though now there are comparatively few knowledge representation languages that use Clark's completion as the basis for their semantics, the notion has not lost its importance for KR. As an illustration let us consider its use for computing answer sets of logic programs. We will need the following terminology.

Definition 3.3.2. *A* nlp Π *is called* **tight** *if there is a mapping* $||\ ||$ *of ground atoms of* Π *into the set of natural numbers such that for every rule (3.5) of* Π

$$||a_0|| > ||a_1||, \ldots, ||a_m|| \tag{3.7}$$

Theorem 2. *If* Π *is tight then* S *is a model of* $Comp(\Pi)$ *iff* S *is an answer set of* Π*.*

The above theorem is due to François Fages. There are some recent results extending the notions of Clark's completion and of tightness, and discovering more general conditions for equivalence of the two semantics. Note that whenever the two semantics of Π are equivalent, Π's answer sets can be computed by satisfiability solvers (specialized inference engines for propositional theories). The connections between answer sets of a program and models of its Clark's completion have been recently used to substantially improve efficiency of ASP solvers.

3.3.2 Well-Founded Semantics

In the late eighties there were several attempts to deal with the problems of Clark's semantics of *nlp*. Two of them — stable model and well-founded semantics — are probably most relevant to the use of logic programs for knowledge representation. The stable model semantics has an epistemic character and can be viewed as a variant of ASP semantics in the context of *nlp*. The well-founded semantics of Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf is based on a different idea. The semantics defines the notion of a unique three-valued model of an *nlp* program Π, called Π's *well-founded* model.

For any *nlp* Π, the function Γ_Π from sets of atoms to sets of atoms is defined by equation

$$\Gamma_\Pi(X) = ans(\Pi^X) \tag{3.8}$$

where $ans(\Pi^X)$ is the answer set of the reduct Π^X from the definition of answer set. It is clear that stable models of Π can be characterized as the fixpoints[4] of Γ_Π. It is not difficult to show that, if $X \subset Y$ then $\Gamma_\Pi(Y) \subset \Gamma_\Pi(X)$. This implies that the function Γ_Π^2 is monotone and hence has, by the Knaster-Tarski Theorem, a least and a greatest fixpoint. (Γ_Π^2 is shorthand

[4]An element x from the domain of a function f is called a *fixpoint* of f if $f(x) = x$. If f is defined on a collection of sets, then a fixpoint x is called a *least* fixpoint of f if no proper subset of x is a fixpoint of f. Similarly for a greatest fixpoint.

for $\Gamma_\Pi(\Gamma_\Pi(X))$.) Atoms belonging to the least fixpoint of Γ^2_Π are called *well-founded* relative to Π. Atoms belonging to the complement of the greatest fixpoint of Γ^2_Π are called *unfounded* relative to Π.

Definition 3.3.3. *A three-valued function which assigns* 1 *(*true*) to atoms well-founded relative to* Π, 0 *(*false*) to atoms unfounded relative to* Π, *and* 1/2 *(*undefined*) to all the remaining atoms is called the* **well-founded model** *of* Π. *An atom is a* **well-founded consequence** *of* Π *if it is true in* Π*'s well-founded model.*

From this definition one can easily see that every *nlp* has a unique well-founded model and that every well-founded consequence of Π is also Π's consequence with respect to the stable model semantics. To better understand the difference between the semantics, let us look at several examples.

Example 3.3.2. Consider the following program Π

$$\begin{aligned} a &\leftarrow \ not\ b.\\ b &\leftarrow \ not\ a.\\ c &\leftarrow a.\\ c &\leftarrow b. \end{aligned}$$

To construct its well-founded model, we will need to find the least fixpoint and the greatest fixpoint of Γ^2_Π. The empty set seems a good candidate for a least fixpoint, so we begin by letting $X = \emptyset$. The reduct of $\Pi^\emptyset$ is

$$\begin{aligned} &a.\\ &b.\\ c &\leftarrow a.\\ c &\leftarrow b. \end{aligned}$$

and, since $\{a, b, c\}$ is the reduct's answer set, $\Gamma_\Pi(\emptyset) = \{a, b, c\}$. Applying Γ_Π to $\Gamma_\Pi(\emptyset)$ gives us the empty set and thus, $\emptyset$ is a fixpoint of Γ^2_Π. Since the set is empty, there are no well-founded literals in the well-founded model of Π.

The set $\{a, b, c\}$ is the greatest candidate for a fixpoint we can test. The reduct of $\Pi^{\{a,b,c\}}$ is

$$\begin{aligned} c &\leftarrow a.\\ c &\leftarrow b. \end{aligned}$$

and $\Gamma_\Pi(\{a, b, c\}) = \emptyset$. Applying Γ_Π again, we get $\Gamma^2_\Pi\{a, b, c\} = \{a, b, c\}$. Thus, $\{a, b, c\}$ is the greatest fixpoint of Π. Its complement is $\emptyset$, so there

are no unfounded literals in the well-founded model of Π either. This means that, in the well-founded model, all of the program's literals are undefined.

This is not so for the stable model semantics under which there are two answer-sets: $\{a, c\}$ and $\{b, c\}$. You can see that c is true in both models, and thus, under stable model semantics Π entails c while under well-founded semantics it does not. This example demonstrates that well-founded semantics does not support reasoning by cases whereas this ability is a built-in feature of stable model semantics.

Example 3.3.3. Consider the program Π

$$p \leftarrow \ not\ p.$$

From the standpoint of stable model semantics, it is inconsistent. It has no stable model (and hence the set of stable consequences of Π contains all the *nlp* literals of the language of Π). In contrast, from the standpoint of the well-founded semantics, Π is consistent. It has empty sets of well-founded and unfounded atoms. Therefore, it has the well-founded model in which every *nlp* literal of Π is undefined, i.e. is assigned the value 1/2. Hence the answer given by the program to query p is *undefined*.

Example 3.3.4. Consider the following program Π :

$$\begin{aligned} p &\leftarrow \ not\ a.\\ p &\leftarrow \ not\ b.\\ a &\leftarrow \ not\ b.\\ b &\leftarrow \ not\ a. \end{aligned}$$

Π has two stable models $\{p, a\}$ and $\{p, b\}$. As in the previous example, the well-founded model of Π has empty sets of well-founded and unfounded atoms. Hence the well-founded model of Π assigns *undefined* to every *nlp* literal of Π. This means that p is a consequence of Π in the stable model semantics, while the answer to p in the well-founded semantics is *undefined*.

Finally, let us look at the following example from [36]:

Example 3.3.5. Consider Π consisting of rules:

$$\begin{aligned} a &\leftarrow \ not\ b.\\ b &\leftarrow c,\ not\ a.\\ c &\leftarrow a. \end{aligned}$$

This program has one answer set $\{a, c\}$, and thus has a and c as its consequences. The well-founded model of Π has no well-founded atoms. Its unfounded atoms are $\{a, b, c\}$ and hence, according to the well-founded semantics, atoms a, b, and c are undefined.

There are large classes of programs for which both well-founded and stable model semantics coincide. For instance, this happens for programs without recursive definitions or, more generally, without recursive definitions through negation (i.e., stratified programs). A careful reader may notice that the programs in the above examples do have such definitions. The first program attempts to define p in terms of *not* p. The second defines a in terms of *not* b and b in terms of *not* a. In the third, a is defined in terms of *not* b, and b is defined in terms of a and *not* a.

It is important to notice that the SLDNF resolution of Prolog is sound with respect to the well-founded semantics but it is not complete. Several attempts were made to define variants of SLDNF resolution that compute answers to goals according to the well-founded entailment. One interesting approach, **SLS resolution**, was introduced by Teodor Przymusinski. SLS resolution is based on a type of oracle and, therefore, cannot be viewed as an algorithm. There are, however, several algorithms and systems which can be viewed as SLS-based approximations of the well-founded semantics. One of the most powerful of such systems, **XSB**, expands SLDNF with tabling and loop checking.
(It can be found at `www.cs.sunysb.edu/~sbprolog/xsb-page.html`.)
Its use allows the avoidance of many of the loop-related problems of Prolog. For instance, XSB's answer to query $reachable(c)$ for the program from Example 3.3.1 will be *no*, while Prolog interpreters will loop on this query.

It is not difficult to expand well-founded semantics of *nlp* to programs allowing classical negation. Of course in this case, not every program will be consistent even according to well-founded semantics. $\{a., \neg a.\}$ is a simple example of a program that is inconsistent under both semantics). However, it proved to be more difficult to find an elegant extension of well-founded semantics to disjunctive programs. There are interesting knowledge representation languages based on the well-founded semantics of logic programs. The future may show if any of those languages will be competitive with ASP as a general KR language or will be shown preferable to ASP in some special cases. Well-founded semantics has already proven to be very useful for the development of efficient ASP solvers and question-answering systems that are sound with respect to ASP, such as XSB.

Summary

The chapter starts with a brief introduction to classical first-order logic with its notions of objects, functions, and relations, the methodology of describing

the world in these terms, clear separation of syntax and semantics, and ideas of interpretation, model and entailment. This is followed by a review of early non-monotonic formalisms that strongly influenced the development of ASP. We discuss *circumscription* which uses the language of first-order logic but only considers models minimal with respect to some ordering; *autoepistemic logic* and *default logic* which are propositional theories whose semantics are aimed at capturing the notion of rational belief. Next, we give a brief review of the negation as failure operator of Prolog and Clark's completion — the earliest attempt to give a declarative semantics of *nlp*. We conclude with a section on well-founded semantics which, for programs without disjunction, can be viewed as an alternative to the stable model semantics used in this book. The intent was to allow the reader to better understand the background in which ASP has been developed. Each of the formalisms outlined in this chapter is of independent interest and can be seriously studied using other sources.

References and Further Reading

Gottlob Frege presented a logical formalism similar to first-order logic in [44]. In addition to boolean connectives ([20]), the formalism contained quantified variables, which became a major tool in mathematics and mathematical logic. A good description of FOL and its relation to Knowledge Representation can be found in [68]. As a comprehensive introduction to logic for computer scientists, we recommend [81]. The original papers on circumscription, autoepistemic logic, and default logic appeared in the same issue of the *Journal of Artificial Intelligence* [74], [77], and [98]. The original formalisms were quickly extended and generalized by multiple authors (see, for instance, [66] and [80]). The monograph [71] gives a mathematically rigorous but very readable introduction to the field. Procedural definition of the negation as failure operator of Prolog can be found in [69]. Clark's completion was first presented in [26]; well-founded semantics in [47]; stable model semantics in [49]. The later was based on the relationship between negation as failure of Prolog and the belief operator of autoepistemic logic discovered in [48]. The relationship between logic programs (without disjunction) and Reiter's default logic was independently established in [50] and [17], [18]. A good early (and still very useful) survey of different declarative formalizations of negation as failure can be found in [6]. SLS resolution, published as a technical report in 1987, appeared as a journal publication in [95]. (See also related work in [101]). To learn more about the XSB system,

one can consult [25]. Theorem 1 first appeared in [73]. The notion of tightness (under a different name) together with Theorem 2 was introduced in [39] and generalized by a number of authors (see, for instance, [38]). The relationship between generalized stable model semantics and circumscription is presented in [41].

Exercises

1. Given universe $\{a, t, d, b, c\}$, FOL signature:

$$\begin{aligned}\mathcal{O} &= \{ant,\ tarantula,\ dragonfly,\ butterfly,\ centipede\}\\ \mathcal{F} &= \{\ \}\\ \mathcal{P} &= \{insect,\ spider,\ bigger\}\\ \mathcal{V} &= \{X, Y\}\end{aligned}$$

   and interpretation

$$\begin{aligned}&I(ant) = \{a\},\\ &I(tarantula) = \{t\},\\ &I(dragonfly) = \{d\},\\ &I(butterfly) = \{b\},\\ &I(centipede) = \{c\},\\ &I(insect) = \{\langle a\rangle, \langle d\rangle, \langle b\rangle\}\\ &I(spider) = \{\langle t\rangle\}\\ &I(bigger) = \{\langle t, a\rangle\}\end{aligned}$$

   check if the following formulas are true under this interpretation:

   (a) $bigger(butterfly, ant)$

   (b) $\forall X\,(insect(X) \supset \neg spider(X))$

   (c) $\neg\exists X\,(insect(X) \wedge spider(X))$

   (d) $\forall X\,(insect(X) \vee spider(X))$

2. Consider an FOL theory consisting of all four sentences of Exercise 1. Is it satisfiable? Justify.

3. Given the signature from Exercise 1 show that the theory consisting of sentence

$$\forall X\, insect(X)$$

   entails $insect(tarantula)$.

4. Given a signature consisting of object constants a and b and predicate constants p and q and a partial order $<_p$ as defined in the chapter, use circumscription to find $<_p$-minimal models of theory:

$$\begin{gathered}a \neq b\\ \forall X(X = a \vee X = b)\\ p(a) \vee q(a)\end{gathered}$$

Assume that the universe consists of two elements, e_1 and e_2, and $M(a) = e_1$ and $M(b) = e_2$.

5. Use the relationship between Answer Set Prolog and autoepistemic logic outlined in Proposition 3.2.1 to find a stable expansion of

$$\neg Bp \supset q$$
$$\neg Bq \supset \neg q$$

6. Use the relationship between Answer Set Prolog and default theory outlined in Proposition 3.2.2 to find an expansion of the default theory given by:

$$\left\{ \frac{p(a) : Mp(b)}{p(c)}, \quad \frac{M\neg p(a)}{p(a)} \right\}$$

7. Find Clark's completion of predicate *edge* given in Example 3.3.1.

8. Find Clark's completion of the following program:

$$tool(hammer).$$
$$person(fred).$$
$$animal(horse).$$
$$animate(X) \leftarrow person(X).$$
$$animate(X) \leftarrow animal(X).$$
$$inanimate(X) \leftarrow \ not\ animate(X).$$

9. Using the definitions in Section 3.3.2 and given the following program Π:

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

   (a) Find the least fixpoint of Γ^2_Π.

   (b) Find the greatest fixpoint of Γ^2_Π.

   (c) Find the well-founded model of Π.

Chapter 4

Creating a Knowledge Base

In order to reason about the world, an agent must have information about it. Since it is unreasonable to teach an agent everything there is to know, we decide on what kind of agent we are building and educate it accordingly. The collection of facts about the world we choose to give the agent is called a knowledge base. Here we will give several examples of creating knowledge bases using the declarative approach. The emphasis will be on the methodology of knowledge representation and the use of ASP. Through examples in several, very different domains, we wish to stress the importance of:

- modeling the domain with relations which ensure a high degree of elaboration tolerance;
- the difference between knowledge representation of closed vs open domains; i.e., we need to know when we can assume that our information about a relation is complete and when we should instead reason with incomplete information, but realize that we are doing so (reasoning with incomplete information is covered much more thoroughly in Chapter 5);
- representing commonsense knowledge along with expert knowledge;
- recursive definitions and hierarchical organization of knowledge.

The first knowledge base contains various relationships within the family group, the second models electrical circuits, and the third deals with a basic taxonomic hierarchy that includes classes such as "submarine" and "vehicle." (For now we will only make use of basic rules and recursion. More-sophisticated knowledge bases will be discussed in later chapters.)

Examples in this chapter will be given using notation necessary to run programs. To make ASP programs executable, we replace $\neg$ with `-`, $\leftarrow$ with `:-`, and *or* with `|`.

For information on systems used to run the examples, please see Appendix B.

4.1 Reasoning about Family

In this section we consider several extensions of the simple family knowledge base from Chapter 1. The first extension familiarizes the program with such basic terms as *brother*, *sister*, etc. The second defines a notion of orphan. The third deals with a recursive definition of the notion of ancestor. Each extension has its own assumption about completeness of information for various relations of the domain.

4.1.1 Basic Family Relationships

Suppose our family from the introduction has an exciting announcement — John and Alice had another baby boy, Bill. To record this joyous event, we add three new statements to our knowledge base: $father(john, bill)$, $mother(alice, bill)$, and $gender(bill, male)$. We will also need a sort $person$ containing names of people possibly relevant to the domain. For illustrative purposes we assume that the list consists of names used in the original knowledge base together with two new names, Bill and Bob. The new program, `family.lp`, looks like this:

```
person(john).
person(sam).
person(bill).
person(alice).
person(bob).

father(john,sam).
father(john,bill).

mother(alice,sam).
mother(alice,bill).
```

```
gender(john,male).
gender(alice,female).
gender(sam,male).
gender(bill,male).

parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

child(X,Y) :- parent(Y,X).
```

This may be an appropriate moment for teaching our agent a new family relation — "X is a brother of Y." Let us denote this relation by $brother(X, Y)$. At first glance, the following simple rule should suffice to define the new notion:

```
brother(X,Y) :- gender(X,male),
                father(F,X),
                father(F,Y),
                mother(M,X),
                mother(M,Y).
```

This rule says that X is a brother of Y if X is male and X and Y have the same parents. To check if the rule is properly understood, let us use STUDENT to answer queries

$$? \; brother(sam, bill).$$

$$? \; brother(sam, X).$$

While the first query is answered correctly, the second gives a surprising answer — $brother(sam, sam)$.

The agent is not at fault — we are. Of course, every one of us knows that a boy cannot be his own brother, but STUDENT does not share this knowledge. It must be explicitly stated in the rule. To do that we will use built-in operator `!=` which stands for $\neq$. Here is the correct definition:

```
brother(X,Y) :- gender(X,male),
                father(F,X),
                father(F,Y),
                mother(M,X),
                mother(M,Y),
                X != Y.
```

We can test the new program and see that now the answers are correct. We succeeded in teaching the agent a new family relation!

The problem we experienced in defining the notion of brother is symptomatic of a serious difficulty confronted by a programmer in the process of knowledge representation. *A very large part of our knowledge is so deeply engrained in us that we do not normally think about it.* Bringing this knowledge out in the open requires discipline and well-developed power of introspection. (In this declarative programming is not very different from procedural.)

Several fascinating sub-areas of AI deal with discovering and codifying such hidden "common sense" knowledge and with finding ways of its efficient and elegant representation. In the following chapters we will discuss some latest advances in this field.

Even though our agent can answer a large number of questions about relationships within the family, an experienced teacher will be able to discover serious gaps in its knowledge of the domain. Consider for instance the questions:

$$? \; father(alice, bill)$$

$$? \; father(bill, sam)$$

$$? \; father(john, bob)$$

What are the expected answers to these queries? Well, the answer to the first two questions we expect from humans are obviously *no*. STUDENT, however, will return *unknown* to both of them. This is again not surprising. To answer the first query correctly, the program should know that females cannot father children. This information can be easily incorporated into the program by the rule

```
-father(X,Y) :- person(X), person(Y),
                gender(X,female).
```

The second answer is justified because we know that a person can have only one father. This is expressed by the rule

```
-father(X,Y) :- person(X), person(Y), person(Z),
                father(Z,Y),
                X != Z.
```

Strictly speaking, this is not enough. Our program also needs to know that Bill and John are two different people. We do not need to add anything, though, because ASP has a built-in **Unique-Name Assumption (UNA)**.

This means that the objects in our program are considered distinct if their names are different unless we have specified otherwise. In other words, our agent considers "John" and "Dad" to be distinct people, even though in real life, he may answer to both. With the UNA, the first two questions are correctly answered by *no*.

Now try to use your common sense to answer the third question. Informal tests conducted with a fairly large number of students show that the responders are divided into two groups: a big one whose members respond with a definite *no*, and a smaller one with people that give a hesitant *maybe*. The difference can be explained by varying understanding of the context of our family story. The larger group apparently makes the so called Closed World Assumption (CWA) — they assume that the story contains *complete* information about John's family. This justifies the following simple argument: "The story does not mention that John is the father of Bob and, therefore, he is not." The members of the second group do not assume that the given information about John's family is complete, (e.g. Bob can be John's son from a previous marriage); hence, the cautious answer. The reasoner associated with our program belongs to the second, more cautious and deliberate group. To move it to the first group, we should be able to explicitly state that our information about John's fatherhood is complete. This can be done by using the default negation of ASP. The rule

```
-father(X,Y) :- person(X),person(Y),
                not father(X,Y).
```

says that if there is no reason to believe that X is the father of Y, then he is not. This is exactly the closed world assumption for fathers mentioned above. Similar rules can be added for other relations of our program. We suggest that the reader test the additions to the program to see that they do indeed give the correct results. For more practice, experiment with closed world assumptions for *mother*, *parent* and *brother*.

4.1.2 Defining Orphans

Here is another example. We are given a list of people in the domain, ($person(bob)$, etc.) and a list of children together with complete information about their parents, e.g.

```
child(mary).
child(bob).
father(mike,mary).
```

```
father(rich,bob).
mother(kathy,mary).
mother(patty,bob).
dead(rich).
dead(patty).
```

Completeness of information about parents of children can be expressed by axioms:

```
-father(F,C) :- child(C),
                person(F),
                not father(F,C).
-mother(M,C) :- child(C),
                person(M),
                not mother(M,C).
```

Let us also assume that the death records are complete, and that all persons considered to be children have a child record. This will be represented as

```
-child(X) :- person(X),
             not child(X).
-dead(X) :- person(X),
            not dead(X).
```

In our example, the first record describes a child, Mary, whose parents are Mike and Kathy. Since their death is not recorded, they must be alive. The next record describes child, Bob, whose parents, Rich and Patty, have passed on.

Now our goal is to teach an agent the notion of an orphan. Before we try to create a mathematical definition, we must first understand what it means for someone to be an orphan. The dictionary defines an orphan as a child whose father and mother are dead. The definition is simple enough, but there is important commonsense knowledge associated with it. For example, we have a sense for the maximum age of those considered children in the sense of the definition of orphan, but this is not well-defined. Also, our dictionary mentions that sometimes a child who has lost only one parent can be considered an orphan. Both questions must be resolved with the "users" prior to coding. We chose to simplify things by explicitly saying that someone is considered a child (that is why this information is made a part of our knowledge base). We also decided to stick with the first definition of orphan. Every programmer has been faced with the importance of precise specifications. A formal language that is used for commonsense reasoning

gives us even more opportunity to test our use and understanding of natural language[1].

Getting back to our problem and referring to the definition of orphan, we write the following rules:

```
parents_dead(P) :- father(F,P),
                   mother(M,P),
                   dead(F),
                   dead(M).

orphan(P) :- child(P),
             parents_dead(P).
-orphan(X) :- person(X),
              not orphan(X).
```

Here *parents_dead* is an auxiliary predicate added for readability. For simplicity we assume that users of the program will not even be aware of its existence. This allows us to skip defining when $parents\_dead(P)$ is false. The next rules encode the definition of the orphan. The closed world assumption is justified by completeness of our death records and the information about the names of children's parents. Let's add these rules to our program and see what STUDENT knows. If we ask whether Bob is an orphan, $orphan(bob)$, we get *yes*. Query $orphan(mary)$ will be answered by *no*.

4.1.3 Defining Ancestors

Consider the following problem. You are given Bill's family tree represented as a set of facts of the form:

```
father(bob,bill).
father(rich,bob).
father(mike,mary).
father(sam,rich).
mother(mary,bill).
mother(patty,bob).
mother(kathy,mary).
mother(susan,rich).
```

[1] Although we will not discuss methods for *automatically* translating from an English description to a formal specification, we would like to note here that this step, known as *natural language processing*, is also a very important part of the general problem of artificial intelligence.

and the list

```
person(bob).
person(bill).
...
```

of all the people mentioned in the tree shown in figure 4.1. We assume that

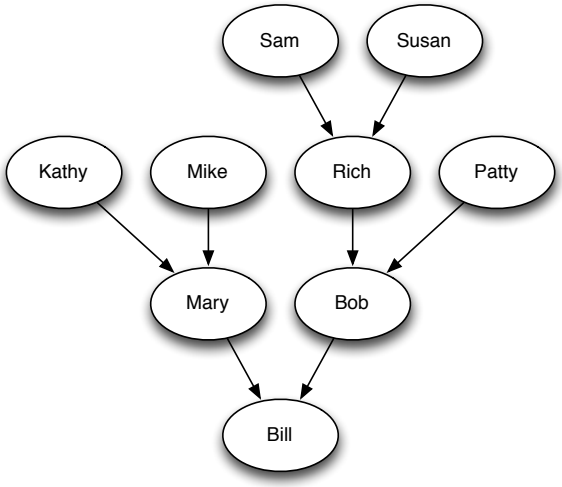


Figure 4.1: Ancestors

the knowledge base contains complete information about parents of each child in the domain, which can be expressed by the rule

```
-father(F,C) :- person(F), person(C),
                not father(F,C).
-mother(M,C) :- person(M), person(C),
                not mother(M,C).
```

(Note that by children we mean everyone except Sam, Susan, Mike and Kathy whose parents are unknown. But since the domain contains no persons not mentioned in the family tree, the closed world assumption above is justified even for them.)

Our goal is to teach the agent the notion of $ancestor(X, Y)$ — "X is an ancestor of Y." How would we define the notion of ancestor? Are your parents your ancestors? There are different definitions. Let us pick one and assume that they are. So are their parents' parents. And so on and so on. Unfortunately, the computer does not understand that last English

expression but, with the proper language, it can be made to reason about it using **recursive definitions**.

The general structure of a recursive definition requires that the base case for a concept be defined. For example, our closest ancestors are our parents. Then we assume that we know how to define some ancestor n and concentrate on expressing what it means to be an $(n+1)$-th ancestor.

ASP lends itself naturally to recursive definitions. This gives it a substantial advantage over knowledge representation methods that do not allow recursion (e.g., traditional relational databases). The end result is both more elegant and more economical. Here is an ASP definition of ancestor:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
-parent(X,Y) :- person(X), person(Y),
                not parent(X,Y).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y),
                 ancestor(X,Z).
-ancestor(X,Y) :- person(X), person(Y),
                  not ancestor(X,Y).
```

Note that the assumption of completeness of our information about children justifies the CWA rules for parents and ancestors. This program will allow our agent to conclude that Bob is Bill's ancestor, as well as Mary, Rich, Patty, Mike, Kathy, Sam and Susan. We also conclude that Mary is not Bob's ancestor, etc. In general the program will give a definite answer to a question $ancestor(a, b)$ for any two persons a and b.

The definition of ancestors concludes our discussion of family relationships. We hope that even this simple program is sufficient to give some insight into the power of ASP. The process of programming is rather natural and does not require a lot of knowledge about the language. The resulting program is clear, concise, and elaboration tolerant. Writing something comparable in traditional procedural or object-oriented programming languages like *C* or *C++* would require good knowledge of data structures and substantially more effort. The result would be less readable and more difficult to modify. A really good programmer, however, might achieve better efficiency than that which can be achieved by existing ASP inference engines; however, this would hardly matter for this domain (even for a family with hundreds of thousands of members). Of course, implementations of ASP are constantly improving as well so, without extra effort by the knowledge base

programmer, the efficiency of the implementation is getting better all the time, although it is unlikely to ever match a truly expert implementation in a lower-level language. The choice of language for a particular task will, as always, depend on the importance of program efficiency versus the price paid in developers'

4.2 Reasoning about Electrical Circuits

Suppose you were asked to describe a simple electrical circuit in ASP. What would be the first thing you would try to do? Desperately look for your old EE book to review what a circuit looks like? Describe gates? Wires? Connections? Just so we are all on the same footing, let's look at a picture of what we mean by a simple circuit. G_0 is a NOT gate, G_1 is an AND gate

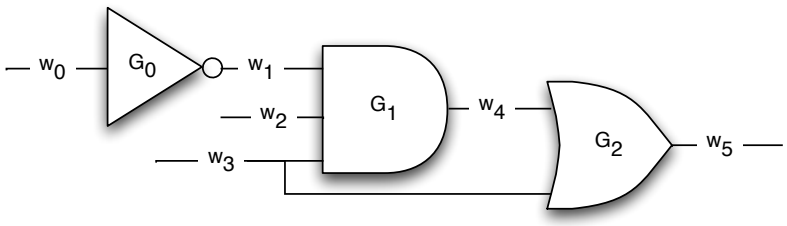


Figure 4.2: Electrical Circuit

and G_2 is an OR gate. Let's keep it simple and limit ourselves to two-valued circuits.

So, what next? When writing any logic program, a person should always ask oneself: "What are the objects and the relations that I am trying to represent?" The objects are wires and gates, and the connections are the relationships between them. Though it sounds obvious, this is not a trivial step. For example, it is tempting to think of a gate and its inputs and outputs as a single object. It turns out, however, that this would complicate the representations. Gates can have different numbers of inputs, forcing us to distinguish between gates that are otherwise similar. Output wires of some gates can be input wires to others, but storing them with the gates will not specify their function. The identification of objects and relations in a domain will greatly affect the representation and the ease and elegance of programming.

So, if we choose gates and wires as our objects, we can describe the above

circuit as follows:

```
wire(w0).  wire(w1).  wire(w2).  wire(w3).  wire(w4).  wire(w5).

gate(g0).
type(g0,not_g).
input(g0,w0).
output(g0,w1).

gate(g1).
type(g1,and_g).
input(g1,w1).
input(g1,w2).
input(g1,w3).
output(g1,w4).

gate(g2).
type(g2,or_g).
input(g2,w4).
input(g2,w3).
output(g2,w5).
```

Given the values on input wires, e.g.

```
val(w0,1).
val(w2,0).
val(w3,1).
```

the agent should be able to predict values on all other wires. To do this, it must know how NOT, AND and OR gates function. This is accomplished by the following recursive definition of relation *val*:

```
signal(0).
signal(1).

% A NOT gate flips the value of the signal:

opposite(0,1).
opposite(1,0).

val(W1,V1) :- output(G,W1),
              type(G,not_g),
```

```
              input(G,W0),
              val(W0,V0),
              opposite(V1,V0).

% The output of an AND gate is 0 if at least one input is 0:

val(W1,0) :- output(G,W1),
             type(G,and_g),
             input(G,W0),
             val(W0,0).

% It is 1 otherwise:

val(W1,1) :- output(G,W1),
             type(G,and_g),
             -val(W1,0).

% The output of an OR gate is 1 if at least one input is 1:

val(W1,1) :- output(G,W1),
             type(G,or_g),
             input(G,W0),
             val(W0,1).

% It is 0 otherwise:

val(W1,0) :- output(G,W1),
             type(G,or_g),
             -val(W1,1).
```

Finally, negation of the relation *val* is defined by the closed world assumption.

```
-val(W,V) :- wire(W),
             signal(V),
             not val(W,V).
```

We may also add the rule

```
-val(W,V1) :- signal(V1),
              signal(V2),
```

```
            val(W,V2),
            V1 != V2.
```

to avoid erroneous input assigning both, 0 and 1 to an input wire. It is not difficult to show that, given proper values of these wires the program computes the unique value for every other wire of the circuit.

Query STUDENT with $val(A, B)$. Does it predict the gate values correctly? For more practice, run the program on several sets of inputs and test if STUDENT predicts the outputs correctly.

Notice that it is easy to build on to the network. Adding objects amounts to naming them. If they are gates, we must specify their type. Hooking them up to the configuration requires stating the gate's inputs and outputs. The existing definitions do not need to be changed in any way.

What we just encoded can be viewed as a kind of expert knowledge about circuits. It is not unreasonable to imagine that this simple example could be expanded into some sort of advice-giving system for humans dealing with complex circuitry. If humans were to safely rely on it for procedural instructions, it must have some commonsense knowledge on top of its expert knowledge. Suppose this system was smart enough to determine that one of the gates was defective. First, it might need to be able to draw the conclusion that, if a component is defective, it must be replaced. Second, it may need to know that it is dangerous to replace a component in a system if there is current running through it. ASP gives us the power to add this knowledge without changing languages, systems, etc. A possible implementation of this idea looks like this:

```
%% Assume we have a sensor that tells us the actual
%% value of the output wire of a gate by setting the
%% value of predicate sensor_val for that wire.
%% Then if the sensor value does not match the
%% predicted value, the gate must be defective.

defective(G) :- gate(G),
                output(G,Output_wire),
                sensor_val(Output_wire, SV),
                val(Output_wire,V),
                SV != V.

needs_replacing(G) :- defective(G).

%% We can also encode the knowledge that
```

```
%% a gate is dangerous to replace if any of
%% its input wires might have the value 1;
%% i.e., if it is not known whether the value of W is 0.

dangerous_to_replace(G) :- gate(G),
                           input(G,W),
                           not val(W,0).
```

(Note that if we assume completeness of information about *val*, then `not val(W,0)` can be replaced by `-val(W,0)`.) Based on this information, the system might perform some pre-arranged function like notifying the operator or shutting off the current.

4.3 Hierarchical Information and Inheritance

Consider how one might represent the following information in ASP.

- The *Narwhal* is a submarine.
- A submarine is a vehicle.
- Submarines are black.[2]
- The *Narwhal* is a part of the U.S. Navy.

Here is one possible solution:

```
sub(narwhal).
vehicle(X) :- sub(X).
black(X) :- sub(X).
part_of(narwhal,us_navy).
```

This program is short and encodes exactly what was given. It allows an agent to conclude that the *Narwal* is a submarine, is a vehicle and is black. However, when a human hears the story, there is much commonsense information that is assumed. For example, suppose we ask our agent whether the *Narwhal* is a car. A human could answer this question without further information, but the STUDENT would rightfully answer *unknown*, since we

[2]The reader has probably noticed that our specification makes a generalization about the color of submarines. It would have been better to say that *normally* submarines are black. In the next chapter, we will revisit hierarchies and show how to express such statements, known as defaults.

did not teach it that cars are not submarines. Likewise, if we wanted to know if the submarine was red, we would have the same problem. This lack of negative information could be remedied by adding the following axioms to the program:

```
-car(X) :- sub(X).
-sub(X) :- car(X).
-red(X) :- black(X).
-black(X) :- red(X).
```

Notice that as soon as we decide we want to allow other types of vehicles and other colors in our program, we will need to add two lines for each addition just so that the agent could correctly answer some simple commonsensical queries.

We can come up with a better representation if we exploit the hierarchical nature of the information from our story. Humans are good at organizing information about the world into tree-like structures of classes and subclasses. For example, when we are told that the *Narwhal* is a submarine, we understand that it belongs to a class of things that are called submarines and, thus, has certain properties that all submarines are assumed to have. With a submarine being a vehicle, objects that are submarines will inherit properties of vehicles as well. An **inheritance hierarchy** is a collection of classes organized into a tree formed by the subclass relation. Figure 4.3 shows the hierarchical structure of the classes in the story. (This time the notions of *car* and *red* used in the above questions will be made part of the representation.) Note that children of the class do not necessarily form

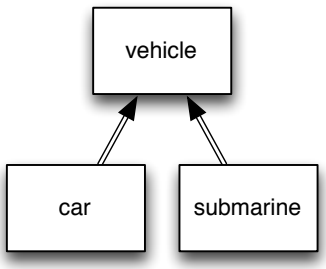


Figure 4.3: Subclasses in the Expanded Submarine Story

its partition. In other words we do not exclude the existence of classes not mentioned in our representation. Thus, there is the possibility that our domain contains unknown or irrelevant classes which the designer decided

not to include in the signature of our program. As a result the reasoner associated with the program will not be able to conclude that a vehicle x is either a submarine or a car — it could belong to some other class of vehicle left outside of our representation and not included in the hierarchy.

To represent this hierarchy we identify the implicit classes relevant to our story and make them objects of our domain. These classes will be organized into a hierarchy defined by relation *subclass*. Once concepts become objects of the domain, we can define relations on them and explicitly reason about whether they satisfy these relations. This process, often called **reification**, is frequently used to generalize and improve the quality of knowledge representation.

Syntactically reification of classes is accomplished by introducing a new sort *class*

```
class(sub).
class(car).
class(vehicle).
```

Relation $is\_subclass(C_1, C_2)$ corresponding to a subclass link of the hierarchy will be defined as follows

```
is_subclass(sub,vehicle).
is_subclass(car,vehicle).
```

The subclass relation will be defined as the transitive closure[3] of $is\_subclass$:

```
subclass(C1,C2) :- is_subclass(C1,C2).

subclass(C1,C2) :- is_subclass(C1,C3),
                   subclass(C3,C2).
```

As usual in recursive definitions we also add

```
-subclass(C1,C2) :- class(C1),
                    class(C2),
                    not subclass(C1,C2).
```

To be able to talk about *objects of the classes* of the hierarchy, like the *Narwhal*, we expand our picture by allowing a new type of link connecting objects to their corresponding classes (see Figure 4.4).

[3]A binary relation R^* is called the transitive closure of binary relation R if $R \subseteq R^*$ and for all X, Y, Z if $R(X, Z)$ and $R(Z, Y)$ then $R(X, Y)$ and no proper subset of R^* satisfies these properties.

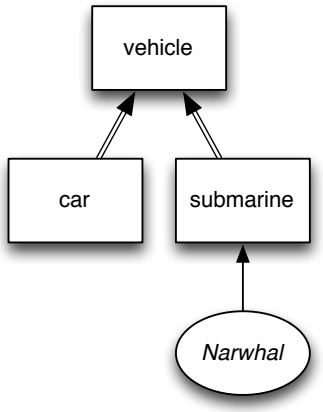


Figure 4.4: Adding an is_a Link

To represent these new links we introduce a new sort, *object*, and a new relation, $is\_a(X, C)$, where X is an object and C is a class. For our story, this will have the form

```
object(narwhal).
is_a(narwhal,sub).
```

Now we define the main relation between objects and classes, called $member(X, C)$. The positive part of membership will be defined as follows:

```
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).
```

But, unlike the case of the *subclass* relation, we do not have complete information about membership. For example, Figure 4.5 shows a vehicle called the *Mystery*, but we do not know what kind of vehicle it is.

```
object(mystery).
is_a(mystery,vehicle).
```

We allow such members in our hierarchies and expect an agent to answer *unknown* to queries about whether the *Mystery* is a car or a sub. Therefore, we do not wish to use the CWA to represent negative information about membership.

However, the designers of hierarchies often make a weaker assumption: normally, children of a class in a hierarchy are disjoint. Thus in the absence

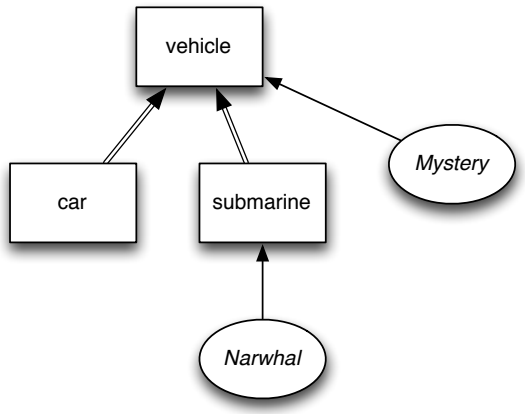


Figure 4.5: Incomplete Information about the Classification of the *Mystery*

of information to the contrary, it is reasonable to conclude that the *Narwhal* cannot be both a submarine and a car. Of course, it is possible to have exceptions to this rule, since plenty of action heroes have vehicles that are difficult to categorize, but we do not discuss handling such cases until the next chapter. For now, we drop the word "normally" and assume that sibling subclasses are disjoint. This can be expressed by the following rules:

```
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2.
```

That's it for the hierarchical structure.

In addition to classes of vehicles and their members, the story also talks about colors. Their representation will be substantially facilitated by the same process of reification we used for classes. We will make colors full-fledged citizens of our domain by introducing a sort *colors* and populating it with two colors mentioned in the story:

```
colors(black).
colors(red).
```

The fact that the submarines are black will be expressed as

```
color(X,black) :- member(X,sub).
```

Note that now we can ask what color something is, not just whether it is black. Recall that in our first implementation of the story, we wanted our program to realize that if a submarine is black, then it is not red. In other words, we had an implicit assumption that objects can only have one color. With our new representation this assumption can be made explicit by the following axiom which, unlike the axioms we were forced to add to the first version, will work once and for all, no matter how many colors we choose to introduce:

```
-color(X,C2) :- object(X),
                colors(C1),
                colors(C2),
                color(X,C1),
                C1 != C2.
```

(If we want to talk about multi-colored objects, we simply introduce new colors, e.g. *black_red*.)

The last line of the original program

```
part_of(narwhal,us_navy).
```

will remain unchanged.

The new program is clearly more powerful than the first. Anything the old program can do, this program can do better. Not only can our agent answer all previous questions correctly, including that the *Narwhal* is not a car and is not red, but it understands some deep, general notions about the nature of hierarchies and colors. With the first program, we can only ask if a particular submarine is a vehicle, not whether submarines in general are vehicles. It is easy to formulate these questions for the second program. It will correctly answer both queries $member(narwhal, vehicle)$ and $subclass(sub, vehicle)$.

Reification has not only made our program more general, but also more elaboration tolerant, i.e., more easily modifiable. For instance, to conclude that *Narwhal* is not white we simply familiarize our agent with that color by adding

```
colors(white).
```

If we wanted to say that vehicles are a type of machine, we could add two more lines:

```
class(machine).
is_subclass(vehicle, machine).
```

and, thanks to our subclass axioms, the whole tree structure would be correctly adjusted. Our agent would know that the *Narwhal* is a submarine, which is a type of vehicle, which is a type of machine. It would also know that it is not a car. Try asking $member(narwhal, X)$.

If we said that a vehicle is something that can be used for traveling, the second program could easily be expanded to allow the agent to deduce that this property would be true for all subclasses of vehicle. It would need no changes, just the following addition:

```
used_for_travel(X) :- member(X,vehicle).
```

If we wanted to say that machines are not alive, we could just add

```
-alive(X) :- member(X,machine).
```

We could then derive that machines, vehicles, cars, and submarines are not alive.

We have seen how the process of reification and the methodology of hierarchical organization can be very useful in creating elaboration tolerant programs. The main differences between the two solutions of the original problem are their brevity, generality and modifiability. In the beginning, the first program was somewhat shorter but, with minor subsequent changes, it lost this advantage. The second program is more general and more elaboration tolerant, which allows it to incorporate changes without substantial growth in size.

It is important to realize that the quest for elaboration tolerant programming paradigms has been ongoing since the beginning of computer science and included such developments as high-level languages, the notion of procedure and module, object-oriented programming, etc. However, having helpful tools does not necessitate their appropriate use. When developing programs in any language, we must always strive to predict just how much a program will be expected to change, and balance compactness with elaboration tolerance.

Summary

In this chapter we have seen that the ASP-based declarative approach to knowledge representation is applicable to a wide variety of domains. We showed it to be capable of representing definitions (including recursive ones), open and closed world assumptions, hierarchical knowledge, etc. In practice it has been used successfully in representing problems of circuit design,

decision support system for space shuttle controllers, team building that ensures fair scheduling of employees with necessary skills, automatic systems for software configuration management, data integration, semantic-web programming, and more, and applications are emerging in such varying fields as molecular biology, psychology, and linguistics.

We also discussed the basic methodology of representing knowledge in ASP including the importance of good selection of objects and relations of the domain, the power of reification, and the necessity of explicitly stating commonsense assumptions, thinking about possible extensions of the domain and the degree of elaboration tolerance of your program. In the next chapter we will show how this methodology can be extended to represent another important concept of knowledge representation — defaults and their exceptions.

References and Further Reading

The best source for learning more about representing knowledge in Answer Set Prolog is the book [13] by Chitta Baral. An ASO axiomatization of preference among defaults can be found in [53]. There is also a substantial amount of work on defining preferences between arbitrary logic programming rules as well as between answer sets of a logic program (see, for instance, [102] [34], [33], [22]). A general account of preferences and their role in non-monotonic reasoning can be found in [23]. Reasoning with inheritance hierarchies based on their graphical representation is described in [21]. For an approach to knowledge representation based on well-founded semantics of logic programs, one can consult [4]. To get a better idea of a range of applications of ASP one can look at [86] and [11] (decision support for the space shuttle controllers), [99] (automation of the team-building process for the largest Italian seaport, [104] (product configuration), [70] (information extraction from the WEB), [19] (music composition), and [12] (psychology).

Exercises

1. Define and test relation
   brothers(X,Y) — "X and Y are brothers".

2. Define and test relation
   uncle(X,Y) — "X is an uncle of Y".

To test this relation you will need to populate your world with relatives of John and/or Alice.

3. Add a gate (with some input and output wires) to the configuration in Section 4.2 and test the program on values with STUDENT.

4. A directed graph G can be described by a set of vertices, represented by facts $vertex(a), vertex(b), \ldots$ and a set of edges, represented by facts $edge(a, b), edge(a, c), \ldots$ Use ASP to define relation $connected(X, Y)$ which holds iff there is a path in G connecting vertices X and Y.

5. Consider a directed graph represented as in the previous exercise, but assume that some of its edges can be blocked (denoted as $blocked(X, Y)$). Redefine relation $connected(X, Y)$ as follows: two vertices X and Y of the graph are *connected* if there a path from X to Y such that no edge of this path is blocked.

6. "Jets are faster than birds. There is an eagle that is faster than every robin. The SR-71 Blackbird is a jet. Jo is a robin." Write a program to describe this story. Make sure that it can derive that the SR-71 is faster than Jo. *Hint:* Build a hierarchy of flying objects.

7. Modify the existing vehicle hierarchy to add the new subclasses and member shown in Figure 4.6. Add some property of water vehicles to your program and make sure the *Narwhal* inherits this property and *Abby* does not.

8. Consider a hierarchy where is_a is applicable only to members of leaf classes. Note that this would imply that information about membership is complete.

   (a) Give a recursive definition of membership using CWA. (Make sure $\neg member$ is part of your definition.)

   (b) The program in exercise 8a still has the assumption that sibling classes are disjoint. Create a new program by removing this assumption. Compare the two programs. Do they have the same answer sets? Do they behave equivalently under updates? *Hint:* Consider the addition of an element that belongs to two subclasses of the program.

9. Consider the description of hierarchy as in Section 4.3 with the difference that we assume that we have complete information about the

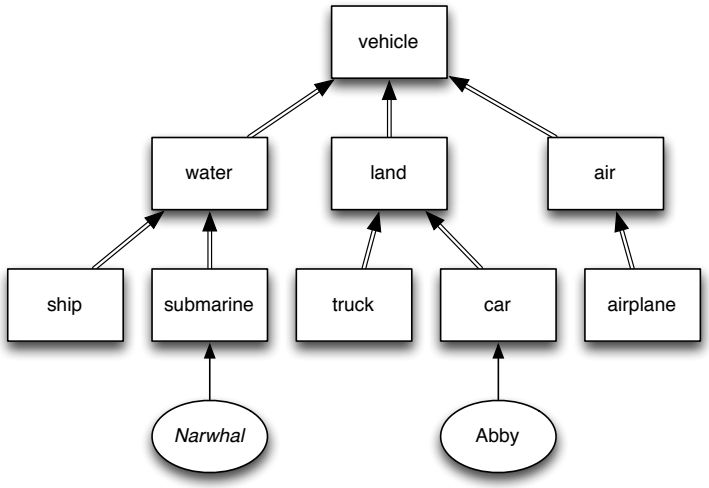


Figure 4.6: New Subclasses of Vehicles

subdivision of subclasses; i.e., a subclass description is a complete partition of the parent class, so that the sum of the parts form a complete whole. This can be expressed by adding the following rules:

$$member(X, C) \; or \; \neg member(X, C) \leftarrow object(X),\ leaf(C).$$
$$in\_a\_leaf(X) \leftarrow object(X),\ leaf(C),\ member(X, C).$$
$$\neg in\_a\_leaf(X) \leftarrow object(X),\ not\ in\_a\_leaf(X).$$
$$\leftarrow object(X),\ \neg in\_a\_leaf(X).$$

The first rule forces the agent to consider membership for each object and each leaf class, and the last three require an object to be in at least one leaf class.

(a) Define relation *leaf(C)* which holds only if C is a leaf class of the hierarchy. *Hint:* Define $\neg$*leaf(C)* first.

(b) Compare the answers that would be given by the hierarchy program that includes the *Mystery* with and without the new rule on the following query:

$$sub(mystery)\ or\ car(mystery)$$

Chapter 5

Representing Defaults

The closed world assumption introduced in the previous chapter is an example of a **default** — a statement of natural language containing words like "*normally*," "*typically*," or "*as a rule*." These sorts of beliefs are very useful to humans since we do not have complete information about the world, but must still be able to draw conclusions based on our knowledge of what is common. However, these conclusions are tentative and we may be forced to withdraw them when new information becomes available. In fact, a large part of our education seems to consist of learning various defaults, their exceptions, and the skill of reasoning with them. Defaults do not occur in the language of mathematics and, therefore, were not studied by classical mathematical logic. However, they play a very important role in everyday, commonsense reasoning, and present a considerable challenge to AI researchers. In this chapter we show how defaults and various forms of exceptions to them are represented in ASP and how this general representation can be used for reasoning in a variety of simple domains. After that we give a number of more advanced examples of the use of defaults. We start with reasoning about knowledge bases with incomplete information represented by so called null values. Next we show how defaults can be prioritized so that, in some cases, we prefer one default over another. Finally, we discuss the use of defaults when representing hierarchies of classes and the inheritance of class properties by subclasses and members.

5.1 A General Strategy for Representing Defaults

In this section we will present the strategy for representing defaults and their exceptions in ASP.

5.1.1 Uncaring John

To illustrate the general idea of defaults, let us go back to our family example from Chapter 1. Suppose you are Sam's teacher and you strongly believe that Sam needs some extra help to pass the class. You convey this information to Sam's father, John, and expect some action on his part. Your reasoning probably goes along the following lines:

1. John is Sam's parent.
2. *Normally*, parents care about their children.
3. Therefore, John cares about Sam and will help him study.

The second statement is a typical example of a default.

To model this reasoning we introduce relation

$$cares(X, Y) \text{ — "}X \text{ cares for } Y\text{."}$$

The first inclination may be to ignore the word *normally* and simply expand program

$$\begin{array}{l} person(john). \\ person(sam). \\ person(alice). \\ \\ father(john, sam). \\ mother(alice, sam). \\ \\ parent(X, Y) \leftarrow father(X, Y). \\ parent(X, Y) \leftarrow mother(X, Y). \\ \\ child(X, Y) \leftarrow parent(Y, X). \end{array}$$

by the new rule

$$cares(X, Y) \leftarrow parent(X, Y). \tag{5.1}$$

The new program derives $cares(john, sam)$.

Assume now that in addition to the default "normally parents care about their children" you learn that "John is an exception to this rule. He does not care about his children." In everyday reasoning this new information does not cause contradiction. We simply withdraw our previous conclusion, $cares(john, sam)$, and replace it by the new one, $\neg cares(john, sam)$. As was discussed in Chapter 2, reasoning which allows removal of previously

achieved conclusions when new information becomes available is called non-monotonic. Since classical mathematical logic is monotonic, it is poorly suited for representing defaults; ASP does not have this difficulty.

A default, d, stated as "Normally elements of class C have property P," is often represented by a rule:

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

Here, $ab(d(X))$ is read "X is abnormal with respect to d" and $not\ \neg p(X)$ is read "$p(X)$ *may* be true."

The same technique can be used if X is a list of variables. For instance, the default d_{cares} "normally parents care about their children" can be represented as follows:

$$\begin{array}{rcl} cares(X,Y) & \leftarrow & parent(X,Y), \\ & & not\ ab(d_{cares}(X,Y)), \\ & & not\ \neg cares(X,Y). \end{array} \tag{5.2}$$

Let us now compare the "strict" caring rule (5.1) used in the beginning with the new rule (5.2). Let F be the family knowledge base including the definition of parents, and let $\Pi_1 = F \cup (5.1)$ and $\Pi_2 = F \cup (5.2)$. You can check that both programs entail $cares(john, sam)$.

Now let's see what happens when we learn that John does not care about his children. There is no way to incorporate this information into Π_1 — the program will become inconsistent. We can, however, add this new knowledge to Π_2 using the rule:

$$\neg cares(john, X) \leftarrow child(X, john).$$

The new program is consistent and entails $\neg cares(john, sam)$ and $cares(alice, sam)$. Note that the new information about John forces the program to withdraw one of its previous conclusions and replace it by the new one.

Defaults can have two types of exceptions — weak and strong. **Weak exceptions** render the default inapplicable; i.e., one or more of the default's premises is not satisfied. This makes our agent unable to use the default to come to a hasty conclusion. **Strong exceptions** refute the default's conclusion; i.e., they negate it. Therefore, a strong exception will allow the agent to derive, with certainty, the opposite of the default. A weak exception $e(X)$ is encoded by the so-called **Cancellation Axiom**

$$ab(d(X)) \leftarrow\ not\ \neg e(X). \tag{5.3}$$

which says that d is not applicable to X if X *may be* a weak exception to d. If e is a strong exception we need one more rule,

$$\neg p(X) \leftarrow e(X) \tag{5.4}$$

which will allow us to defeat d's conclusion.

To illustrate the notion of weak exception, let us emulate a cautious reasoner who does not want to apply default d_{cares} to people whose spouses do not care about their children. Such a reasoner will prefer not to make any judgment on Alice's relation to Sam. This can be achieved by adding the following rule:

$$\begin{array}{lcl} ab(d_{cares}(P_1, C)) & \leftarrow & parent(P_1, C), \\ & & parent(P_2, C), \\ & & \neg cares(P_2, C). \end{array}$$

The new program answers *no* to query $cares(john, sam)$ and *maybe* to query $cares(alice, sam)$.

Uncaring John serves as an example of a strong exception. According to our general methodology, the fact that he does not care for his children is translated into ASP by the rules:

$$\neg cares(john, X) \leftarrow parent(john, X).$$

$$ab(d_{cares}(john, X)) \leftarrow person(X),\ not\ \neg parent(john, X).$$

Notice however that the second rule is useless and can be safely removed from the program. (Indeed if John is a parent of X, then the first rule applies and defeats the default. If no information about $parent(john, X)$ is available, the default is not applicable anyway.)

To better understand the need for the cancellation axioms, let's consider another strong exception to the "caring parents" default. Assume the existence of a mythical country, u, whose inhabitants do not care for their children. This exception is represented by rules

$$\begin{array}{lcl} \neg cares(P, X) & \leftarrow & parent(P, X), \\ & & born\_in(P, u). \\ ab(d_{cares}(P, X)) & \leftarrow & person(P),\ person(X), \\ & & not\ \neg born\_in(P, u). \end{array}$$

Suppose we have an extension of our family knowledge base which contains information about the national origin of most (but not all) recorded people.

Assume, for instance, that according to our records, Pit and Kathy are the father and mother of Jim. Kathy was born in Moldova, but the national origin of Pit is unknown. He could have been born in u. Without the cancellation axiom, the program would have derived that Pit did care about Jim, even though his origin was unknown. With it, we can see that the queries $cares(kathy, jim)$ and $cares(pit, jim)$ are correctly answered *yes* and *unknown*, respectively. If later we learn that Pit is indeed from u, then the second answer will be replaced by a definite *no*.

5.1.2 Cowardly Students

Now let us consider an example which has both, weak and strong exceptions. Consider the following information:

1. Normally, students are afraid of math.
2. Mary is not.
3. Students in the math department are not.
4. Those in CS may or may not be afraid.

The first statement corresponds to a default, say d. The next two can be viewed as strong exceptions to it. The fourth is a weak exception.

We will assume that we are given two classes of objects, *student* and *dept*, containing names of all students and departments of the domain. Suppose also that a (possibly incomplete) list L

$$\begin{array}{l} in(john, english\_dept). \\ in(mary, cs\_dept). \\ in(bob, cs\_dept). \\ in(pat, math\_dept). \end{array}$$

relates students to their unique departments. To define negative information for relations *student* and *dept*, we can use the CWA. However, since L may be incomplete, we need to find some other way to define negative information about in. The rule

$$\begin{array}{rcl} \neg in(S, D_1) & \leftarrow & student(S), \\ & & dept(D_1),\ dept(D_2), \\ & & in(S, D_2), \\ & & D_1 \neq D_2. \end{array}$$

does that by exploiting the uniqueness of departments. The resulting program entails $\neg in(mary, engl\_dept)$, etc.
The default (statement 1) is translated by the rule:

$$\begin{array}{lll} afraid(S, math) & \leftarrow & student(S), \\ & & not\ ab(d(S)), \\ & & not\ \neg afraid(S, math). \end{array}$$

According to statement 2 from the specification Mary is a strong exception to this default. According to our methodology, it can be represented as:

$$\begin{array}{l} ab(d(mary)). \\ \neg afraid(mary, math). \end{array}$$

Note that, in this case, $ab(d(mary))$ is not necessary and can be removed.

The strong exception for math students (statement 3) is expressed as follows:

$$\begin{array}{lll} \neg afraid(S, math) & \leftarrow & student(S), \\ & & in(S, math\_dept). \end{array}$$

The following cancellation rules for CS and math students (statement 4) allow us to express weak exceptions:

$$\begin{array}{lll} ab(d(S)) & \leftarrow & student(S), \\ & & not\ \neg in(S, cs\_dept). \\ ab(d(S)) & \leftarrow & student(S), \\ & & not\ \neg in(S, math\_dept). \end{array}$$

We recommend that the reader check that the following answers to queries are correct by informally tracing the rules of the program and by using STUDENT to answer these queries.

? *afraid(john,math)*	Yes
? *afraid(mary,math)*	No
? *afraid(pat,math)*	No
? *afraid(bob,math)*	Unknown

Now consider another student, Jake, whose department affiliation is unknown. An agent using this program will correctly answer that it does not know whether Jake is afraid of math. Notice that this can only be done thanks to our cancellation axioms which allow us to express weak exceptions. However, if it was known that Jake was neither in the computer science nor in the math department, then the agent would correctly derive that he was afraid of math.

5.1.3 A Special Case

Let d be a default "Elements of class C normally have property P" and e be a set of exceptions to this default. If our information about membership in e is complete, then its representation can be substantially simplified. If e is a weak exception to d then the Cancellation Axiom 5.3 can be written as

$$ab(d(X)) \leftarrow e(X). \tag{5.5}$$

If e is a strong exception then the axiom 5.3 can be simply omitted.

For instance in our cowardly students example, if we had a complete list of students in the CS department, we could replace our original cancellation axiom for CS students by

$$ab(d(X)) \leftarrow student(X),\ in(X, cs\_dept).$$

If, in addition, we knew that our information about student membership in the math department is complete, then the cancellation axiom for math students could simply be dropped.

5.2 Knowledge Bases with Null Values

Now let us look at the use of defaults for representing and reasoning about incomplete information in the presence of *null values* — constants used to indicate that the value of a certain variable or function is unknown. This technique is commonly used in relational databases but, due to the multiple and not always clearly specified meanings of these constants, it often leads to ambiguity and confusion. We will show how the use of defaults can alleviate this problem.

5.2.1 Course Catalog

Consider a database table representing a tentative summer schedule of a Computer Science department.

Professor	**Course**
mike	pascal
john	c
staff	prolog

Here "staff" is a null value which stands for an unknown professor. It expresses the fact that Prolog will be taught by *some* professor (possibly different from Mike or John).

To represent this information we introduce a relation $teaches(P, C)$ which says that professor P teaches a course C. We also assume that we are given complete collections of professors and courses.

The positive information from the table can be represented by a collection of facts:

$$teaches(mike, pascal).$$
$$teaches(john, c).$$
$$teaches(staff, prolog).$$

To represent negative information, we use default d: Normally, P teaches C only if this is listed in the schedule. Notice that d is not applicable to Prolog (or any other course taught by "staff"). This can be represented as follows:

$$\begin{array}{lcl} \neg teaches(P, C) & \leftarrow & prof(P), course(C), \\ & & not\ ab(d(P, C)), \\ & & not\ teaches(P, C). \\ ab(d(P, C)) & \leftarrow & teaches(staff, C). \end{array}$$

Check that the resulting program produces correct answers (*No* and *Unknown*) to queries $teaches(mike, c)$ and $teaches(mike, prolog)$, respectively.

There can be yet another type of incompleteness in database tables.

Professor	**Course**
mike	pascal
john	c
{mike, john}	prolog
staff	prolog

Here {mike, john} represents the second type of nulls in which the value is unknown, but one of a finite set. To represent this information we simply expand our program by

$$teaches(mike, prolog)\ or\ teaches(john, prolog).$$

With the new program, our agent's answers to queries $teaches(mike, c)$, $teaches(mike, prolog)$ and $teaches(mike, prolog) \wedge teaches(john, prolog)$ are *No*, *Unknown*, and *No*, respectively. Using connectives *or* and $\wedge$ allows us to work with this type of incompleteness.

5.3 Simple Priorities between Defaults

In this section we will look at a way to represent simple priorities between defaults. We start with assuming that the agent's knowledge base contains a database of records similar to that used in orphans story from Section 4.1. To make the example slightly more realistic we remove the assumptions that a child's record contains complete information about a child's parents. We still assume that deaths of all people whose records are kept in our knowledge base are properly recorded. Similarly for information about being a child.

In addition to these records let us supply our agent with knowledge about some fictitious legal regulations. The first regulation says that *orphans are entitled to assistance according to special government program 1*, while the second says that all *children are entitled to program 0.* The rules also say that *program* 1 *is preferable to program* 2, i.e. *a child qualified for receiving assistance from program* 1 *shall not receive assistance from program* 0 and that *no one can receive assistance from more than one program.*

Let us start with representing these regulations and than proceed with defining the records of the agent's knowledge base. Legal regulations usually come with exceptions and hence can be viewed as defaults. Here is a complete representation. We use relation $record\_for(P)$ to indicate that the agent's knowledge base contains a record for P. We start with listing the assistance programs, and follow by representing the first two regulations by standard default rules, and by a rule prohibiting assistance from more than one program.

```
program(0).
program(1).

entitled(X,1) :- record_for(X),
                 orphan(X),
                 not ab(d1(X)),
                 not -entitled(X,1).
entitled(X,0) :- record_for(X),
                 child(X),
                 not ab(d2(X)),
                 not -entitled(X,0).
-entitled(X,N2) :- program(N1), program(N2),
                   record_for(X),
                   entitled(X,N1),
                   N1 != N2.
```

The next two rule express preference of program 1 over program 2. Essentially we treats orphans (who are entitled to assistance from program 1) as strong exceptions to the second default. (In other circumstances preference can be expressed by weak exceptions.) The double negation in the last rule is needed since, due to incompleteness of our records we may not know if a child is an orphan or not.

```
-entitled(X,0) :- record_for(X),
                  orphan(X).
ab(d2(X)) :- record_for(X),
             not -orphan(X).
```

There are also the following strong exceptions to both defaults:

```
-entitled(X,N) :- record_for(X),
                  dead(X),
                  program(N).
-entitled(X,N) :- record_for(X),
                  -child(X),
                  program(N).
```

(Since information about *dead* and *child* is complete we do not need to use double negation in the premisses of these rules.) The rules guarantee that if Joe is an orphan then he will receive assistance from program 1. If Joe is a child who is not an orphan then he will be assisted by program 0. However if Joe is a child and it is not known whether he is an orphan or not, then Joe will receive no benefits. Of course this is not right. Something should be done about this case of insufficient documentation. The problem can be detected by the following rule:

```
check_status(X) :- record_for(X),
                   not -orphan(X),
                   not orphan(X).
```

In other words, the person in charge of financial assistance will need to go to some extra trouble to check person X's status if the system does not know whether or not X is an orphan.

Let us now describe records from the agent's knowledge base. We use a slight modification of the database from previous family example. As before, we assume that there is a relation *person* satisfied by any reasonable name we decide to use. But the rest of the information will be more structured. For testing purposes let us assume that there are records for the following people:

```
record_for(bob).
father(rich,bob).
mother(patty,bob).
child(bob).

record_for(rich).
father(charles,rich).
mother(susan,rich).
dead(rich).

record_for(patty).
dead(patty).

record_for(mary).
child(mary).
mother(patty,mary).
```

To express the assumption that deaths of all people whose records are kept in the knowledge base are properly recorded we expand the above records by CWA for *dead*:

```
-dead(P) :- record_for(P),
            not dead(P).
```

Similarly for the children.

```
-child(X) :- record_for(X),
             not child(X).
```

Notice that our closed world assumptions are only applied to people who have records in the agent's knowledge base. This is simply because we are not going to ask any questions about those who are not. If, by accident, such a question is asked the answer will be unknown.

Now we are ready to define a notion of orphan. The positive part of the definition remains the same as before.

```
orphan(P) :- child(P),
             parents_dead(P).
```

The negative part however will undergo a substantial change. The closed world assumption for orphans used in the previous example will be replaced by a weaker statement:

```
-orphan(P) :- record_for(P),
              not may_be_orphan(P).
```

where *may_be_orphan* will be defined as follows:

```
may_be_orphan(P) :- record_for(P),
                    child(P),
                    not -parents_dead(P).

parent(X,P) :-
   father(X,P).
parent(X,P) :-
   mother(X,P).

parents_dead(P) :-
       father(X,P),
       dead(X),
       mother(Y,P),
       dead(Y).

-parents_dead(P) :-
       parent(X,P),
       -dead(X).
```

It is easy to check that the program will entail

```
entitled(bob,1).
-entitled(bob,0).
check_status(mary).
```

Suppose that after checking the status of Mary the administrator discovered that Mary has a father, Mike, who is alive. This information will be recorded in the knowledge base as follows:

```
record_for(mary).
child(mary).
mother(patty,mary).
father(mike,mary).
```

and

```
record_for(mike).
```

Now the system will entail

```
entitled(bob,1).
-entitled(bob,0).
entitled(mary,0).
-entitled(mary,1).
```

Of course the system will also derive that no other person whose record is stored in the database is not entitled to any of the assistance programs. (Note, however, that the system will not be able to give a definite answer about entitlements for Charles and Susan who have no such records. Moreover, the records for these people cannot be created since we do not know if they are dead or alive. The addition would violate our assumption about completeness of the death records.) This is one more example of usefulness of discriminating between falsity and mere absence of information. This, together with simple expression of preferences between defaults allowed our program to produce correct conclusions based on such absence.

5.4 Inheritance Hierarchies with Defaults

Whenever humans have a hierarchical organization of information, we make assumptions about certain properties that members of classes share with each other. For example, if we find out that something is an animal, we can assume that it eats, breathes, etc. However, there can always be exceptions to such rules and it is foolish to cling too tightly to conclusions we make based on class membership. Default reasoning is essential for making commonsense assumptions, but taking exceptions into account is essential to true intelligence.

5.4.1 Submarines Revisited

Now that we have the power to represent defaults, let's return to the hierarchical representation of our *Narwhal* example from Section 4.3. Instead of saying that all submarines are black, we can say that normally submarines are black. In accordance with our general methodology we simply replace

```
color(X,black) :- member(X,sub).
```

by

```
color(X,black) :- member(X,sub),
                  not ab(dc(X)),
                  not -color(X,black).
```

Suppose that we learned about a submarine named *Blue Deep*. In accordance with its name this submarine is blue. (After all, it makes sense to use blue for better camouflage in case the screen door does not deter some of the fish.) The new information can be added as follows:

```
object(blue_deep).
is_a(blue_deep,sub).
color(blue_deep,blue).
```

The new submarine is a strong exception to our default. As expected the new program is consistent. It allows us to conclude that the *Blue Deep* is blue while retaining our ability to conclude that the *Narwhal* is black by default. If later we learn that the *Narwhal* is also blue, this would cause no contradiction.

5.4.2 Membership Revisited

Another lovely consequence of being able to add the word "normally" to our statements is that we can weaken our assumption that leaf classes of hierarchies are disjoint. The positive part of the definition of *member* and the definition of *sibling* remain unchanged:

```
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
```

The only change needed is the addition of the last line in the definition of $member(X, C)$:

```
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2,
                 not member(X,C2).
```

Now we can introduce an amphibious vehicle called *Darling* owned by a great man of action. It belongs both to the car and the submarine class. This can be recorded by:

```
object(darling).
is_a(darling, car).
is_a(darling, sub).
```

The resulting program will allow our agent to deduce that the *Darling* is a member of both subclasses, but that the *Narwhal* is a sub and not a car.

5.4.3 The Specificity Principle

Let's consider another example that illustrates a classic problem that arose with the study of inheritance hierarchies. "Eagles and penguins are types of birds. Birds are a type of animal. Sam is an eagle, and Tweety is a penguin. Tabby is a cat." We represent this hierarchy exactly as before:

```
class(animal).
class(bird).
class(eagle).
class(penguin).
class(cat).

object(sam).
object(tweety).
object(tabby).

is_subclass(eagle,bird).
is_subclass(penguin,bird).
is_subclass(bird,animal).
is_subclass(cat,animal).

subclass(C1,C2) :- is_subclass(C1,C2).
subclass(C1,C2) :- is_subclass(C1,C3),
                   subclass(C3,C2).

is_a(sam,eagle).
is_a(tweety,penguin).
is_a(tabby,cat).

member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).

siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
```

```
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2,
                 not member(X,C2).
```

Our agent should now be able to answer correctly that Tweety is not an eagle but that Tweety is a penguin, a bird and an animal. All these queries can be made using the *member* predicate.

Now we add default properties of classes. For example, *animals normally do not fly, birds normally fly, and penguins normally do not fly.* (The last default may look strange but we need it for illustrative purposes. After all we may eventually want to consider penguins that fly because they are sprinkled with pixie dust.) What does the new theory allow us to conclude about Sam's ability to fly? Since Sam is both a bird and an animal, his flying abilities are defined by two contradictory defaults. The program will have two answer sets containing $fly(sam)$ and $\neg fly(sam)$, respectively. Our common sense, however, tells us that only the first conclusion is justified. This is apparently the result of a broadly shared commonsense **specificity principle** which states that *more-specific information overrides less-specific information.* The principle, first formalized by David S. Touretzky, gives preference to defaults applicable to subclasses of a hierarchy. The default "normally elements of class C_1 have property P" is preferred to the default "normally elements of class C_2 have property $\neg P$" if C_1 is a subclass of C_2. The same rule is applicable to defaults with arbitrary incompatible conclusions. The following rules represent our defaults together with the specificity principle.

```
%% Animals normally do not fly.
-fly(X) :- member(X,animal),
           not ab(d1(X)),
           not fly(X).

%% Birds normally fly.
fly(X) :- member(X,bird),
          not ab(d2(X)),
          not -fly(X).

%% Penguins normally do not fly.
-fly(X) :- member(X,penguin),
           not ab(d3(X)),
           not fly(X).
```

```
%% X is abnormal with respect to d2 if X might be a penguin.
ab(d2(X)) :- not -member(X,penguin).

%% X is abnormal with respect to d1 if X might be a bird.
ab(d1(X)) :- not -member(X,bird).
```

The last rule, which prohibits the application of default d_1 to animals which might possibly be birds, expresses preference of default d_2 over default d_1. The previous rules does the same for defaults d_3 and d_2. We chose to express exceptions this way because our hierarchy allows objects for which the complete characterization of their membership relation is unknown. For example, if there is a bird that cannot be classified further, the agent should conclude "unknown" about its flying ability. After all, the bird might turn out to be a penguin.

Now our agent has no problem figuring out which animals and birds fly and which do not. Sam flies and Tweety and Tabby do not. And, if we wanted to teach the program about baby eagles that do not fly, or penguins that do, we could.

Note, however, that our formalization does not have a single rule expressing the inheritance principle. Instead we need to write a separate rule for each pair of the corresponding contradictory defaults. This is not surprising because to write such a rule we would need to reify defaults. There are commonsense theories in which defaults are reified and the specificity principle is stated as one rule but they are beyond the scope of this book. You can give it a try in the last exercise.

5.5 (*) Indirect Exceptions to Defaults

In this section we consider yet another type of possible exceptions to defaults, sometimes referred to as **indirect exceptions**. Intuitively, these are rare exceptions that come into play only as a last resort, to restore the consistency of the agent's world view when all else fails. The representation of indirect exceptions seems to be beyond the power of ASP. This observation led to the development of a simple but powerful extension of ASP called **CR-Prolog** (or ASP with consistency-restoring rules). To illustrate the problem let us consider the following example.

Consider an ASP representation of the default "elements of class c nor-

mally have property p":

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

together with the rule

$$q(X) \leftarrow p(X).$$

and two observations:

$$\begin{array}{c} c(x). \\ \neg q(x). \end{array}$$

It is not difficult to check that this program is inconsistent. No rules allow the reasoner to prove that the default is not applicable to x (i.e. to prove $ab(d(x))$) or that x does not have property p. Hence the default must conclude $p(x)$. The second rule implies $q(x)$ which contradicts the observation.

There, however, seems to exists a commonsense argument which may allow a reasoner to avoid inconsistency, and to conclude that x is an indirect exception to the default. The argument is based on the **Contingency Axiom** for default $d(X)$ which says that "Any element of class c can be an exception to the default $d(X)$ above, but such a possibility is very rare and, whenever possible, should be ignored." One may informally argue that since the application of the default to x leads to a contradiction, the possibility of x being an exception to $d(x)$ cannot be ignored and hence x must satisfy this rare property.

In what follows we give a brief description of CR-Prolog — an extension of ASP capable of encoding and reasoning about such rare events. We start with a description of syntax and semantics of the language.

A program of CR-Prolog is a four-tuple consisting of

1. A (possibly sorted) signature.
2. A collection of regular rules of ASP.
3. A collection of rules of the form

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k,\ not\ l_{k+1}, \ldots,\ not\ l_n \tag{5.6}$$

   where ls are literals. Rules of type (5.6) are called **consistency restoring rules (cr-rules)**.
4. A partial order, $\leq$, defined on sets of cr-rules. This partial order is often referred to as a **preference relation**.

Intuitively, rule (5.6) says that if the reasoner associated with the program believes the body of the rule, then it "may possibly" believe its head. However, this possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program. The partial order over sets of cr-rules will be used to select preferred possible resolutions of the conflict. Currently the inference engine of CR-Prolog supports two such relations. One is based on the set-theoretic inclusion ($R_1 \leq_1 R_2$ holds iff $R_1 \subseteq R_2$). Another is defined by the cardinality of the corresponding sets ($R_1 \leq_2 R_2$ holds iff $|R_1| \leq |R_2|$). To give the precise semantics we will need some terminology and notation.

The set of regular rules of a CR-Prolog program Π will be denoted by Π^r; the set of cr-rules of Π will be denoted by Π^{cr}. By $\alpha(r)$ we denote a regular rule obtained from a consistency restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to a set R of cr-rules, i.e. $\alpha(R) = \{\alpha(r) : r \in R\}$. As in the case of ASP, the semantics of CR-Prolog will be given for ground programs. A rule with variables will be viewed as a shorthand for a schema of ground rules.

Definition 5.5.1. *(Abductive Support)*
A minimal (with respect to the preference relation of the program) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (i.e. has an answer set) is called an **abductive support** *of Π.*

Definition 5.5.2. *(Answer Sets of CR-Prolog)*
A set A is called an answer set *of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.*

Consider, for instance, the following CR-Prolog program:

$$p(a) \leftarrow \ not\ q(a).$$
$$\neg p(a).$$
$$q(a) \stackrel{+}{\leftarrow}.$$

It is easy to see that the regular part of this program (consisting of the program's first two rules) is inconsistent. The third rule, however, provides an abductive support which allows to resolve inconsistency. Hence the program has one answer set $\{q(a), \neg p(a)\}$.

The previous example had only one possible resolution of the conflict and hence its abductive support did not depend on the preference relation of the program. This is of course not always the case. Consider for instance

the following collection of rules:

$$\begin{array}{rcl} p_1 & \leftarrow & not\ \neg p_1. \\ \neg p_1 & \stackrel{+}{\leftarrow} & . \\ p_2 & \leftarrow & not\ \neg p_2. \\ \neg p_2 & \stackrel{+}{\leftarrow} & . \\ p_3 & \leftarrow & not\ \neg p_3. \\ \neg p_3 & \stackrel{+}{\leftarrow} & . \\ r & \leftarrow & p_1. \\ \neg r & \leftarrow & p_2. \\ \neg r & \leftarrow & p_3. \end{array}$$

It is not difficult to see that a program consisting of these rules and set inclusion as the preference relation has two answer sets $\{\neg p_1, p_2, p_3, \neg r\}$ and $\{p_1, \neg p_2, \neg p_3, r\}$ corresponding to abductive supports $\{\neg p_1 \stackrel{+}{\leftarrow}\}$ and $\{\neg p_2 \stackrel{+}{\leftarrow},\ \ \neg p_3 \stackrel{+}{\leftarrow}\}$. (Note that the collection of all three cr-rules of the program does not provide a minimal conflict resolution.)

Consider now the program consisting of the above rules and the cardinality-based preference relation. Observe that this program has only one abductive support and one answer set, $\{\neg p_1, p_2, p_3, \neg r\}$.

Now let us show how CR-Prolog can be used to represent defaults and their indirect exceptions. The CR-Prolog representation of default $d(X)$ may look as follows

$$\begin{array}{rcl} p(X) & \leftarrow & c(X), \\ & & not\ ab(d(X)), \\ & & not\ \neg p(X). \\ \neg p(X) & \stackrel{+}{\leftarrow} & c(X). \end{array}$$

The first rule is the standard ASP representation of the default, while the second rule expresses the Contingency Axiom for default $d(X)$[1]. Consider now a program obtained by combining these two rules with an atom

$$c(a).$$

Assuming that a is the only constant in the signature of this program, the program's answer set will be $\{c(a), p(a)\}$. Of course this is also the answer

[1] In this form of Contingency Axiom, we treat X as a strong exception to the default. Sometimes it may be useful to also allow weak indirect exceptions; this can be achieved by adding the rule: $ab(d(X)) \stackrel{+}{\leftarrow} c(X)$.

set of the regular part of our program. (Since the regular part is consistent, the Contingency Axiom is ignored.) Let us now expand this program by the rules

$$q(X) \leftarrow p(X).$$
$$\neg q(a).$$

The regular part of the new program is inconsistent. To save the day we need to use the Contingency Axiom for $d(a)$ to form the abductive support of the program. As a result the new program has the answer set $\{\neg q(a), c(a), \neg p(a))\}$. The new information does not produce inconsistency as in the analogous case of ASP representation. Instead the program withdraws its previous conclusion and recognizes a as a (strong) exception to default $d(a)$.

Here is another small example: Consider a reasoning agent whose knowledge base contains a default which says that "people normally keep their cars in working condition". A (slightly simplified) version of this can be represented in CR-Prolog as follows:

$$\begin{array}{rcl} \neg broken(X) & \leftarrow & car(X), \\ & & not\ ab(d(X)), \\ & & not\ broken(X). \\ broken(X) & \stackrel{+}{\leftarrow} & car(X). \end{array}$$

Suppose also that the agent has some information about the normal operations of cars; e.g. it knows that turning the ignition key starts the car's engine. This knowledge can be represented by the rules:

$$\begin{array}{rcl} starts(X) & \leftarrow & turn\_key(X), \\ & & \neg broken(X). \\ \neg starts(X) & \leftarrow & turn\_key(X), \\ & & broken(X). \end{array}$$

(For simplicity we assume here that broken cars do not start.)

Given that the ignition key of car x was turned, i.e. statements

$$car(x).$$

$$turn\_key(x).$$

belong to the agent's knowledge base, the agent will be able to use regular rules to conclude $\neg broken(x)$ and $starts(x)$. If, however, in addition the agent learns

$$\neg starts(x)$$

both of the above conclusions will be withdrawn, and the cr-rule will be used to prove $broken(x)$, which, in turn, will imply $\neg starts(x)$.

The possibility to encode rare events which may serve as unknown exceptions to defaults proved to be very useful for various knowledge representation tasks, including planning, diagnostics, and reasoning about the agent's intentions. The later chapters contain a number of examples of such uses.

Summary

In this chapter we discussed the general strategy for representing defaults and their exceptions in the language of ASP. The strategy explains how to translate natural language statements of the form "normally, typically, as a rule" etc. into defeasible ASP rules and how to classify exceptions into strong, weak, and indirect. We showed how the first two types of exceptions can be recorded in ASP and gave a general representation of indirect exceptions in CR-Prolog. The non-monotonicity of the ASP entailment was crucial for reasoning which allows the retraction of the defaults' conclusions after new information about exceptions to them became available. A number of examples illustrated the use of defaults for various knowledge representation tasks. In particular we showed how defaults can be used to specify negative information in databases containing incomplete knowledge expressed by null values — a task notoriously difficult in weaker languages. Another example dealt with inheritance hierarchies with default properties of classes. It showed how defaults can be blocked from being inherited by subclasses or objects using the general strategy for encoding exceptions. This allowed us to formalize the informal "specificity principle" used by people in their commonsense reasoning. A similar strategy could be used for specifying general preferences between defaults.

The ability to represent and reason about defaults is a very substantial achievement in KR. The design of logic and languages capable of doing this and understanding the correct ways of reasoning with defaults and their exceptions took many years of extensive research. Now, however, it has been honed enough to be included in a textbook.

References and Further Reading

A number of examples and techniques introduced in this chapter (including the representation of defaults with strong and weak exceptions) were first

presented in [14]. The treatment of null values follows [109].

The specificity principle is due to David Touretzky [108].

CR-Prolog was first introduced by Marcello Balduccini and Michael Gelfond in [37] and [8].

Exercises

Define relations

1. $\neg parent(X, Y)$
2. $sister(X, Y)$
3. $\neg brother(X, Y)$

Use the methodology for encoding defaults to represent the knowledge given in the following stories in ASP.

4. "Apollo and Helios are lions in a zoo. Normally lions are dangerous. Baby lions are not dangerous. Helios is a baby lion." Assume that the zoo has a complete list of baby lions which it maintains regularly. Your program should be able to deduce that Apollo is dangerous, while Helios is not. Make sure that (a) if you add another baby lion to your knowledge base, the program would derive that it is not dangerous, even though that knowledge is not explicit; (b) if you add an explicit fact that Apollo is not dangerous, there is no contradiction and the program answers intelligently.

5. "John is married to Susan and Bob is married to Mary. Married people normally like each other. However, Bob hates Mary."

   (a) Make sure your program answers "yes" to queries
   ? $likes(john, susan)$,
   ? $likes(susan, john)$,
   ? $likes(mary, bob)$,
   and "no" to ? $likes(bob, mary)$.

   (b) Add the following knowledge to your program. "Arnold and Kate are also married, but Kate's behavior often does not follow predictable rules." Make sure your program can deduce that Arnold likes Kate, but it is unknown whether Kate likes Arnold. (Note that if you used the methodology properly, you should not need to change the first program but can just add to it.)

6. "American citizens normally live in the U.S. John is an American citizen but he lives in Italy. American diplomats may or may not live in the U.S." For testing, add U.S. citizens Miriam and Caleb. Miriam is an American diplomat.

   (a) Assume we do *not* have a complete list of American diplomats. Your program should not be able to conclude that Caleb lives in the U.S. If you add the knowledge that Caleb is not a U.S. diplomat, your program should conclude that he lives in the U.S.

   (b) Now assume we have a *complete* list of American diplomats. Use the simplified form of the cancellation axiom. Your program should conclude that Caleb lives in the U.S.

7. "Adults normally work. Children do not work. Students are adults but they normally do not work. John and Betty are students. John works. Bob and Jim are adults who are not students. Bob does not work. Kate is an adult who may or may not be a student. Mary is a child."
   Make sure that your program is not only capable of answering questions about who works and does not work, but also of who is or is not a child, an adult, etc.

8. "A field that studies pure ideas does not study the natural world. A field that studies the natural world does not study pure ideas. Mathematics normally studies pure ideas. Science normally studies the natural world. As a computer scientist, Daniela studies both mathematics and science. Both mathematics and science study our place in the world." Make sure your program can deduce that Daniela studies our place in the world.

9. "Cars normally have 4 seats. Pick-up trucks are exceptions to this rule. They normally have 2 seats. Pick-up trucks with extended cab are exceptions to this rule. They have 4 seats." Your program should work correctly in conjunction with complete lists of facts of the form

$$car(a), car(b), car(c), \ldots$$

$$pickup(b), pickup(c), \ldots$$

$$extended\_cab(c), \ldots$$

10. You are given three complete lists of facts of the form

$$course(math), course(graphs), \ldots$$

$$student(john), student(mary), \ldots$$

$$took(john, math), took(mary, graphs), \ldots$$

Students can graduate only if they have taken all the courses in the first list. Write a program that, given the above information, determines which students can graduate. Make sure that, given the following sample knowledge base,

$$\begin{array}{l} student(john). \\ student(mary). \\ course(math). \\ course(graphs). \\ took(john, math). \\ took(john, graphs). \\ took(mary, graphs). \end{array}$$

your program is able to conclude

$$\begin{array}{l} can\_graduate(john). \\ \neg can\_graduate(mary). \end{array}$$

11. Consider the problem presented in Exercise 10. This time, however, the list of courses that the students took may be incomplete. Write a program that determines:

- which students can graduate
- which students cannot graduate
- which students have incomplete career information listed in the knowledge base; in this case, the program must recommend a review of the student records.

Make sure that, given the following sample knowledge base,

$$
\begin{array}{l}
student(john).\\
student(mary).\\
student(bob).\\
student(rick).\\
course(math).\\
course(graphs).\\
took(john, math).\\
took(john, graphs).\\
\neg took(mary, math).\\
took(mary, graphs).\\
\neg took(bob, math).
\end{array}
$$

your program is able to conclude

$$
\begin{array}{l}
can\_graduate(john).\\
\neg can\_graduate(mary).\\
\neg can\_graduate(bob).\\
review\_records(bob).\\
review\_records(rick).
\end{array}
$$

12. Using the notions of hierarchy and defaults as detailed in Section 5.4, write an ASP program to represent the following information. Be as general as you can.

    - A Selmer Mark VI is a saxophone.
    - Jake's saxophone is a Selmer Mark VI.
    - Mo's saxophone is a Selmer Mark VI.
    - Part of a saxophone is a high D key.
    - Part of the high D key is a spring that makes it work.
    - The spring is normally not broken.
    - Mo's spring for his high D key is broken.

    Make sure that your program correctly entails that Jake's saxophone works while Mo's is broken. For simplicity, assume that no one has more than one saxophone and, hence, saxophones can be identified by the name of their owner.

13. Consider an alternative specification of exceptions to defaults which does not use double negation so that instead of

```
ab(d2(X)) :- not -member(X,penguin).
ab(d1(X)) :- not -member(X,bird).
```

as in Section 5.4.3, we have

```
ab(d2(X)) :- member(X,penguin).
ab(d1(X)) :- member(X,bird).
```

Given a bird, Squeaky, of unknown species, what would the agent answer about its flying ability with *ab* defined in this new way?

14. (*) We have noted before that to encode the specificity principle explicitly, we would have to reify the notion of default. Rewrite the Tweety program making the defaults objects of the domain and encode the specificity principle as a general rule about defaults.

Chapter 6

Answer Set Programming

So far we used our ASP knowledge bases to get information about the truth or falsity of some statements or to find objects satisfying some simple properties. These types of tasks are normally performed by database systems. Even though the language's ability to express recursive definitions and the methodology of representing defaults and various forms of incomplete information gave us additional power and allowed us to construct rich and elaboration-tolerant knowledge bases, the types of queries essentially remained the same as in databases.

In this chapter we will illustrate how significantly-different computational problems can be reduced to finding answer sets of logic programs. The method of solving computational problems by reducing them to finding the answer sets of ASP programs is often called the answer set programming paradigm. It has been used for finding solutions to a variety of programming tasks, ranging from building decision support systems for the Space Shuttle and computer system configuration, to solving problems arising in bioinformatics, zoology, linguistics, etc. In principle, any NP-complete problem can be solved in this way using programs without disjunction. Even more-complex problems can be solved if disjunctive programs are used. This chapter contains a number of examples of the use of ASP methodology. More advanced examples of the use of ASP involving larger knowledge representation components will be discussed in later chapters.

There are currently several ASP inference engines called *ASP solvers* capable of computing answer sets of programs with millions of ground rules. Normally, an ASP solver starts its computation by grounding the program, i.e. instantiating its variables by ground terms. The resulting program has the same answer sets as the original but is propositional. The answer sets

of the grounded program are then computed using generate-and-test algorithms which we will discuss in Chapter 7.[1]

You have, of course, already used an ASP solver if you used STUDENT to run examples or exercises in previous chapters; however, we kept the programs generic enough to run on any solver known to us. Different ASP solvers use various constructs not present in the original ASP discussed in this book. Some of the constructs are aimed at improving the efficiency of the solvers. Others provide useful syntactic sugar which saves programmer's time and may improve readability of programs. For instance, popular ASP solvers such as **Smodels** and **Clasp** use program grounders **Lparse** and **Gringo** whose input languages allow the definition of sorted variables and so-called choice rules not understood by another popular solver called **DLV**. In turn, DLV allows symbols for disjunction, for lists of terms, and for other constructs not understood by Lparse/Gringo.[2]

Among these differences, disjunction deserves special note. Without it, ASP becomes less powerful in what it can express. As we will see, users of Clasp can often avoid using disjunction altogether by using a special construct called the choice rule. And, disjunction can be simulated in Clasp for many cases by using the command-line option `--shift`. However, one must use a different solver, called **ClaspD**, in conjunction with Gringo, to get true disjunction. As we will see in the next chapter, a solver that must account for disjunction has a higher complexity by its very nature. Thus, the developers of Clasp decided to create a separate solver for when disjunction is absolutely necessary. For efficiency reasons, they chose to only allow the approximation of disjunction in their main system. When the `--shift` option is used, the disjunction (p *or* q) is translated into two rules:

$$p \leftarrow \text{not } q$$

$$q \leftarrow \text{not } p.$$

The transformation is only equivalent if p and q are "independent" of each other. To understand the condition, consider (p *or* p) translated by this transformation into $p \leftarrow \text{not } p$. The programs clearly are not equivalent. The first has answer set $\{p\}$ while the second has no answer set.

[1] These algorithms have much in common with classical satisfiability algorithms for propositional logic.

[2] The input language of `clingo`, Gringo, was originally based on Lparse, but is evolving. Since `clingo`, (Gringo + Clasp), seems to be very efficient and is currently being actively developed and improved, we will try to stay compatible with it rather than with Lparse. However, we still call the input language the "Lparse input language" because it was first.

There are currently efforts to standardize the input language. Meanwhile, we encourage readers to learn ASP programming by using their favorite ASP system. For standard ASP programs, this book will focus on two systems, `clingo` (Gringo + Clasp) and DLV, to show the reader some useful features of both input languages. It is likely, that by the time this book is published, the features we mention will become part of both systems and will be useful regardless of which system you choose. For a quick introduction and tips for using these systems, please see Appendix A. Both systems are good and have many things to make them special; however, in our wish to keep things simple, we will not be discussing all the new features of the systems.

6.1 Computing Hamiltonian Paths

We start with describing an ASP solution of finding Hamiltonian paths of a graph. Finding such paths through graphs has applications to numerous problems, including processor allocation and delivery scheduling. The general problem is stated as follows:

> Given a directed graph G and an initial vertex v_0, find a path from v_0 to v_0 which visits each vertex exactly once.

For example, in Figure 6.1, if our initial node is a, the Hamiltonian path through the graph is a, b, c, d, e, a. Path a, b, c, e, a, d visits every vertex exactly once but is not Hamiltonian because it does not end at a. Path a, b, c, d, a is not Hamiltonian because it misses e.

We will now show how the problem of finding a Hamiltonian path can be solved using Answer Set Programming. The main idea is to construct an ASP program, $\Pi_H(G)$, whose answer sets will correspond to Hamiltonian paths of graph G. Once this is done finding the paths will be reduced to finding answer sets of this program — a task which can be accomplished by an ASP Solver. We will first give our solution of the Hamiltonian path problem in DLV, and then give two alternative solutions using `clingo`.

Not surprisingly, the program $\Pi_H(G)$ will contain the description of a graph G. The graph will be represented by atoms describing its vertexes and edges:

$$\begin{gathered} vertex(v_0). \\ \vdots \\ edge(v_i, v_j). \\ \vdots \end{gathered}$$

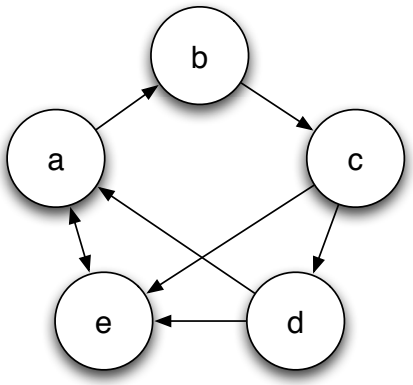


Figure 6.1: Directed Graph

The initial node will be specified by an atom:

$$init(v_0).$$

Note that the information about the input graph is complete and, strictly speaking, this should be indicated by the corresponding closed world assumptions. However, negative information about these relations is not relevant to our problem and will be omitted for simplicity.

Now we are confronted with the problem of representing Hamiltonian paths of G. To do that let us introduce a relation $in(V1, V2)$ read as "the edge from vertex $V1$ to vertex $V2$ is in the Hamiltonian path through the graph." The idea is to represent every Hamiltonian path of G by a collection of statements of the form

$$in(v_0, v_1).\ \ldots\ in(v_k, v_0).$$

which belongs to an answer set of $\Pi_H(G)$.

We start by describing conditions on a collection P of atoms of the form $in(v_1, v_2)$ that will make P a Hamiltonian path:

1. P visits each vertex at most once.[3]

2. P visits every vertex of the graph.

To encode the first condition, we state which edges cannot possibly be in the same path. We need two rules to reflect the symmetric nature of connection.

[3]The initial node is not considered "visited" until it is reached in the end.

(h1)
```
-in(V2,V) :- vertex(V1), vertex(V2), vertex(V),
             in(V1,V),
             V1 != V2.
```

(h2)
```
-in(V,V2) :- vertex(V1), vertex(V2), vertex(V),
             in(V,V1),
             V1 != V2.
```

For the second condition, we recursively define relation $reached(V)$ which holds if P visits vertex V on its way from the initial vertex:

(h3)
```
reached(V2) :- vertex(V1), vertex(V2),
               init(V1),
               in(V1,V2).
```

(h4)
```
reached(V2) :- vertex(V1), vertex(V2),
               reached(V1),
               in(V1,V2).
```

(h5)
```
-reached(V) :- vertex(V), not reached(V).
```

The constraint

(h6)
```
:- vertex(V),
   -reached(V).
```

guarantees that every vertex is reached.

To complete the solution we need to find some way to generate the collection of **candidate paths** and use the above conditions to select those which are indeed Hamiltonian. In DLV this can be done by the disjunctive rule

(h7)
```
in(V1,V2) | -in(V1,V2) :- edge(V1,V2).
```

which states that every given edge is either in the path or is not in the path. The rule requires our answer sets to contain information about each edge's inclusion in the path. To see this more clearly, let us denote the program consisting of the representation of graph G (edges, vertexes, and initial vertex) and rule (h7) by Π_0. Notice that there is a one-to-one correspondence between answer sets of Π_0 and arbitrary sets of edges of G. We will sometimes say that Π_0 generates these sets. Now let Π be Π_0 expanded by *testing rules* (h1)–(h6). This time, there is a one-to-one correspondence between answer sets of Π and Hamiltonian paths in G. They can be computed by DLV. Note that we are not really interested in all the information

contained in the answer sets of the program — all we need is to display our Hamiltonian paths, i.e. atoms formed by relation *in*. To make sure that DLV displays only atoms relevant to describing the Hamiltonian path, use option `-filter=in`. You can also use `-pfilter=in` to display only positive atoms formed by relation *in*.

To solve the problem in the example, we create a program `hamgraph.lp` consisting of rules (h1)–(h7) and the encoding of the graph from Figure 6.1 and initial node *a*. For future reference, let's call this list of facts

(h_graph)
```
vertex(a).
vertex(b).
vertex(c).
vertex(d).
vertex(e).
edge(a,b).
edge(b,c).
edge(c,d).
edge(d,e).
edge(e,a).
edge(a,e).
edge(d,a).
edge(c,e).
init(a).
```

Calling DLV with
`dlv -pfilter=in hamgraph.lp`
we get the following output:

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

which represents the only Hamiltonian path which starts at *a*.

The problem can also be solved by `clingo`. One way to do this is to replace our disjunctive rule (h7) by two non-disjunctive rules

(h7a)
```
in(V1,V2) :- edge(V1,V2),
             not -in(V1,V2).
```

(h7b)
```
-in(V1,V2) :- edge(V1,V2),
              not in(V1,V2).
```

It is easy to check that both programs have the same answer sets.

There is another solution to the problem which which uses a useful extension of the original ASP — the **Choice Rule**. The construct was first introduced in Smodels and now is implemented in all ASP solvers which use Lparse or Gringo as their grounder. Our description will be informal. Those interested in the formal definition should consult the related literature. A choice rule has two forms:

```
(a)  n1 {p(X) : q(X)} n2 :- body.
(b)  n1 {p(c1),...,p(ck)} n2 :- body.
```

Both $n1$ and $n2$ can be omitted. Rules of type (a) allow inclusion in the program's answer sets of arbitrary collections S of atoms of the form $p(t)$ such that

1. $n1 \leq |S| \leq n2$
2. If $p(t) \in S$ then $q(t)$ belongs to the corresponding answer set.

Rules of type (b) allow selection of such an S from atoms listed in the head of the rule.

Example 6.1.1. *(Choice Rule (a))*
Program

```
q(a).
{p(X) : q(X)}1.
```

has answers sets $\{q(a)\}$ *and* $\{q(a), p(a)\}$.

Example 6.1.2. *(Choice Rule (b))*
Program

```
q(b).
{p(a),p(b)}1.
```

has answers sets $\{q(b)\}$, $\{p(a), q(b)\}$, *and* $\{p(b), q(b)\}$. *Replacing the 1 in the last rule by a 2 gives the original three answer sets, plus one more —* $\{p(a), p(b), q(b)\}$.

Using the first form of the Choice Rule allows us to replace rule (h7) by:

(h7c) `{in(V1,V2) : edge(V1,V2)}.`

(h7d) `-in(V1,V2) :- not in(V1,V2).`

Note that the second rule is not really necessary for finding the solution to our puzzle and can be omitted.

In the input language of Lparse, display of relevant information is accomplished by two directives, `#hide` and `#show.` The first tells the program to hide all predicates while the second specifies which predicates the program should show. The following two statements tell the solver to suppress the display of atoms that are different from those formed by predicate *in*.

```
#hide.
#show in(X,Y).
```

You just saw your first example of a successful application of ASP methodology to solving a classical combinatorial problem. Now it may be prudent to spend some time reflecting on this experience. As you know one of the main goals of computer science is to discover new ways of solving computational problems. (Think of the impact the discovery of recursion had on our ability to do that!) From this perspective, it is instructive to compare the *processes* of finding "procedural" versus "declarative" solutions to the Hamiltonian path problem. They are markedly different and lead to markedly different implementations. The first focuses on data structure and algorithm; the second, on the appropriate encoding of the definition of the problem. We strongly encourage you to spend some time finding and implementing the procedural solution. But even without this exercise, one can probably see that the declarative solution is shorter, easier to implement (at least by those who mastered both methodologies), more transparent and more reliable. An important open question is "What are the limits of applicability of the second method?" (Perhaps some of you may decide to contribute to finding the answer to this question.)

Let us also mention that another declarative solution to the problem of finding Hamiltonian paths was tried before ASP was even developed. In this solution a graph G and the definition of Hamiltonian path was encoded by a propositional formula F. There is a one-to-one correspondence between models of F and Hamiltonian paths of G. A program, called a **satisfiability solver** finds the models. Computer scientists were developing satisfiability solvers for propositional logic for more than 50 years and succeeded in producing remarkably efficient systems. So, why use ASP? There are several reasons for this.

- The ASP encoding is much shorter and easier to understand. And, this seems to be the case frequently. Some recent mathematical results

show that this is not an accident: any equivalent translation from logic programs to propositional formulas involves a significant increase in size.

- There are complexity results which prove that ASP with disjunction has more expressive power then propositional logic. Such problems simply cannot be solved by satisfiability solvers.

This of course does not mean that all the remarkable work on satisfiability solvers is in danger of becoming useless. The advantages and disadvantages of both methods are still under investigation. But more importantly the developers of ASP solvers are rapidly finding ways to use ideas from satisfiability theory as well as actual, off-the-shelf satisfiability solvers to build new and more efficient answer set solvers.

6.2 Solving Puzzles

The ability to solve puzzles is an important part of human intelligence. The skill with which students are able to do this is often used in decisions of whether to admit them to graduate school, offer them a job, etc. Even though reasonable people can question the wisdom of such policies, few would argue that the ability to solve puzzles is at least one measure of intelligence. In this section we use ASP to design programs for solving several interesting puzzles.

6.2.1 Sudoku Puzzle

We start with a popular Japanese puzzle game called Sudoku. Our solution will simply represent the Sudoku rules in ASP. The answer sets of the resulting programs will correspond to solutions of the puzzle.

Figure 6.2 shows a typical puzzle. As you can see, the game is played on a 9 x 9 grid that is further subdivided into nine 3 x 3 regions. Initially the grid contains numbers in some of its locations. Players must place the numbers 1 through 9 on the board so that in each row, column, and region the following constraints are maintained:

1. No row contains the same number twice.
2. No column contains the same number twice.
3. No 3 x 3 region contains the same number twice.

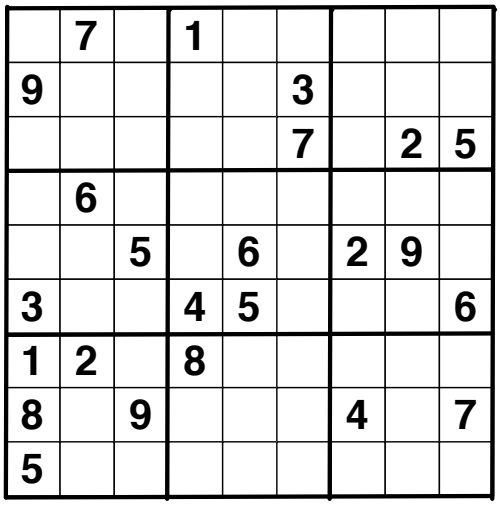


Figure 6.2: Sudoku Puzzle

In other words, each row, column and region must contain the numbers 1 through 9 and each number must appear once and only once in that row, column or region. (Typically, for a given initial situation, the puzzle has exactly one solution.)

To describe the Sudoku's domain, we need names for the grid's locations and regions. We use coordinates — pairs of numbers from 1 to 9 — to name locations. Regions will be numbered from 1 to 9. To describe numbers placed in the grid's locations, we use relation

$pos(N, X, Y)$ — "number N is placed in location (X, Y)."

We will also need a relation

$in\_region(X, Y, R)$ — "location (X, Y) belongs to region R."

We are looking for a collection of atoms of the form $pos(N, X, Y)$ which satisfy the following conditions:

(s1) Every position (X, Y) is assigned a number from 1 to N. In DLV this generating rule can be expressed as

```
num(1..9).
coord(X,Y) :- num(X),num(Y).
pos(1,X,Y) | pos(2,X,Y) | pos(3,X,Y) | pos(4,X,Y) | pos(5,X,Y) |
pos(6,X,Y) | pos(7,X,Y) | pos(8,X,Y) | pos(9,X,Y) :- coord(X,Y).
```

(s2) No row contains the same number twice.

```
-pos(N,X,Y2) :- num(N),
                coord(X,Y1),
                coord(X,Y2),
                pos(N,X,Y1),
                Y1 != Y2.
```

(s3) No column contains the same number twice.

```
-pos(N,X2,Y) :- num(N),
                coord(X1,Y),
                coord(X2,Y),
                pos(N,X1,Y),
                X1 != X2.
```

(s4) No region contains the same number twice.

```
-pos(N,X2,Y2) :- num(N),
                 coord(X1,Y1),
                 coord(X2,Y2),
                 pos(N,X1,Y1),
                 X1 != X2,
                 Y1 != Y2,
                 in_region(X1,Y1,R),
                 in_region(X2,Y2,R).
```

(s5) Now all we need to do to complete our program is define relation *in_region*. This can be done by the following formula:

```
in_region(X,Y,R) :- num(X), num(Y), num(Z1), num(Z2),
                    num(Z3), num(Z4), num(Z5), num(R),
                    Z1 = X-1,
                    Z2 = Z1/3,
                    Z3 = Z2*3,
                    Z4 = Y+2,
                    Z5 = Z4/3,
                    R = Z3 + Z5.
```

Now we can represent an initial position on the board from Figure 6.2 by a collection of atoms of the form $pos(n, x, y)$, and compute answer sets of this program. They will contain the solutions of the puzzle.

To better understand the program one can view rule (s1) as generating all possible assignments of numbers to locations which are checked against the remaining constraints. Again, this is a typical example of answer set programming methodology.

For example, the puzzle in Figure 6.2 can be encoded as follows. We'll call this list of facts

(s_puzzle)

```
pos(7,2,1).
pos(1,4,1).
pos(9,1,2).
pos(3,6,2).
pos(7,6,3).
pos(2,8,3).
pos(5,9,3).
pos(6,2,4).
pos(5,3,5).
pos(6,5,5).
pos(2,7,5).
pos(9,8,5).
pos(3,1,6).
pos(4,4,6).
pos(5,5,6).
pos(6,9,6).
pos(1,1,7).
pos(2,2,7).
pos(8,4,7).
pos(8,1,8).
pos(9,3,8).
pos(4,7,8).
pos(7,9,8).
pos(5,1,9).
```

Let's call the program consisting of rules (s1)–(s5) plus the encoding of this specific instance `sudokudlv.lp`. Calling DLV with
`dlv -pfilter=pos sudokudlv.lp`
produces the following output:

```
{pos(1,1,7), pos(1,4,1), pos(2,2,7), pos(2,7,5), pos(2,8,3), pos(3,1,6),
pos(3,6,2), pos(4,4,6), pos(4,7,8), pos(5,1,9), pos(5,3,5), pos(5,5,6),
pos(5,9,3), pos(6,2,4), pos(6,5,5), pos(6,9,6), pos(7,2,1), pos(7,6,3),
pos(7,9,8), pos(8,1,8), pos(8,4,7), pos(9,1,2), pos(9,3,8), pos(9,8,5),
pos(6,1,1), pos(4,1,3), pos(2,1,4), pos(7,1,5), pos(5,2,2), pos(1,2,3),
pos(8,2,5), pos(9,2,6), pos(3,2,8), pos(4,2,9), pos(2,3,1), pos(8,3,2),
pos(3,3,3), pos(4,3,4), pos(1,3,6), pos(7,3,7), pos(6,3,9), pos(2,4,2),
pos(6,4,3), pos(9,4,4), pos(3,4,5), pos(5,4,8), pos(7,4,9), pos(9,5,1),
pos(4,5,2), pos(8,5,3), pos(7,5,4), pos(3,5,7), pos(2,5,8), pos(1,5,9),
pos(5,6,1), pos(8,6,4), pos(1,6,5), pos(2,6,6), pos(4,6,7), pos(6,6,8),
pos(9,6,9), pos(3,7,1), pos(6,7,2), pos(9,7,3), pos(1,7,4), pos(7,7,6),
pos(5,7,7), pos(8,7,9), pos(4,8,1), pos(7,8,2), pos(5,8,4), pos(8,8,6),
pos(6,8,7), pos(1,8,8), pos(3,8,9), pos(8,9,1), pos(1,9,2), pos(3,9,4),
pos(4,9,5), pos(9,9,7), pos(2,9,9)}
```

(If you wish to check the answer, note that the first four rows contain the input information, while the rest contain the solution numbers in a reasonable order.)

The same problem can, of course, be solved in the input language of Lparse. The generating rule (s1) can be replaced by the following choice rule:

```
1{pos(A,X,Y):num(A)}1 :-
             coord(X,Y).
```

Rules (s2)–(s4) can be left as is or replaced by choice rules:

```
1{pos(N,X,Y):num(X)}1 :-
             num(N),
             num(Y).
1{pos(N,X,Y):num(Y)}1 :-
             num(N),
             num(X).
1{pos(N,X,Y):in_region(X,Y,R)}1 :-
             num(R),
             num(N).
```

Rule (s5) can be left the same or encoded as:

```
in_region(X,Y,((X-1)/3)*3+((Y+2)/3)) :- num(X), num(Y).
```

We recommend inserting

```
#hide.
#show pos(A,B,C).
```

for display purposes. The programs will have answer sets which only differ on negations of $pos(N, X, Y)$ which are irrelevant with respect to our original problem.

For people interested in design of efficient ASP programs, it may be instructive to compare speeds of the Sudoku solutions defined above.

6.2.2 Mystery Puzzle

A detective or mystery story can also be a diverting challenge (providing the author of the story does not cheat). Unlike in Sudoku, it is not enough to know the puzzle's "rules" to find its solution. One normally also needs to make some assumptions which come from our general knowledge about the world. It means that programs solving such puzzles should posses some commonsense knowledge. Let us look at an example.

> Vinny has been murdered, and Andy, Ben, and Cole are suspects. Andy says he did not do it. He says that Ben was the victim's friend but that Cole hated the victim. Ben says he was out of town the day of the murder, and besides he didn't even know the guy. Cole says he is innocent and he saw Andy and Ben with the victim just before the murder. Assuming that everyone — except possibly for the murderer — is telling the truth, use ASP to solve the case.

The story is about four people:

```
person(andy).
person(ben).
person(cole).
person(vinny).
```

The next several statements record their testimony. Relation $says(P, S, 1)$ holds if person P says that statement S is true; $says(P, S, 0)$ means that S is false. The corresponding statements will be represented by self-explanatory terms, e.g. $murderer(andy)$, $friends(ben, vinny)$, etc.

```
%% Andy says:
says(andy, murderer(andy), 0).      %% He didn't do it.
says(andy, hated(cole,vinny), 1).   %% Cole hated Vinny.
```

```
says(andy, friends(ben,vinny), 1).  %% Ben and Vinny were friends.

%% Ben says:
says(ben, out_of_town(ben), 1).    %% He was out of town.
says(ben, know(ben,vinny), 0).     %% He didn't know Vinny.

%% Cole says:
says(cole, innocent(cole), 1).       %% He is innocent.
says(cole, together(andy,vinny), 1). %% He saw Andy and Ben
says(cole, together(ben,vinny), 1).  %% with the victim.
```

(r1) The two rules below formalize the last statement of the puzzle, using relation *holds*(F) — "statement F is true": Everyone, except possibly for the murderer, is telling the truth:

```
holds(S) :- says(P,S,1),
            -holds(murderer(P)).
-holds(S) :- says(P,S,0),
             -holds(murderer(P)).
```

(r2) The rule below states that one of the suspects is a murderer:

```
holds(murderer(a)) | holds(murderer(b)) | holds(murderer(c))
```

Rules (r3)–(r9) contain some commonsense knowledge about the meaning of the relations used by the suspects.

(r3) We start with proclaiming our belief that normally people are not murderers:

```
-holds(murderer(P)) :- not holds(murderer(P)),
                       person(P).
```

Next we specify that some relations are symmetric and/or transitive:

(r4) Relation *together* is symmetric and transitive:

```
holds(together(A,B)) :- person(A), person(B),
                        holds(together(B,A)).
holds(together(A,B)) :- person(A), person(B), person(C),
                        holds(together(A,C)),
                        holds(together(C,B)).
```

(r5) Relation *friends* is symmetric:

```
holds(friends(A,B)) :- person(A), person(B),
                       holds(friends(B,A)).
```

Several other properties express mutual exclusivity of some of relations mentioned in the story. Since these conditions are not used to *define* the corresponding relations, but rather relate two concepts to each other, they will be represented by constraints.

(r6) Murderers are not innocent.

```
:- holds(innocent(P)), holds(murderer(P)), person(P).
```

(r7) A person cannot be seen together with people who are out of town.

```
:- person(A), person(B),
   holds(out_of_town(A)), holds(together(A,B)).
```

(r8) Friends know each other.

```
 :- -holds(know(A,B)), person(A), person(B),
    holds(friends(A,B))
```

(r9) A person who was out of town cannot be the murderer.

```
:- holds(murderer(P)), holds(out_of_town(P)),
   person(P).
```

(r10) To display the answer we introduce relation *murderer* defined as follows:

```
murderer(P) :- holds(murderer(P)),
               person(P).
```

The only answer set of the above program contains $murderer(ben)$, correctly concluding that Ben is the murderer. Clearly, enough commonsense knowledge was added to get the unique answer. Actually not all of this knowledge is even necessary — some of the constraints can be dropped without influencing the result. (We advise the reader to see which constraints can be safely eliminated.)

To rewrite the program for use with `clingo`, we replace rule (r2) by a choice rule:

```
1{murderer(andy), murderer(ben), murderer(cole)}1.
```

The rest of the program remains the same.

Summary

In this chapter we discussed the method of solving a computational problem P by writing an ASP program Π_P whose answer sets correspond to the problem's solutions and using ASP solvers to find those solutions. The program Π_P normally consists of several parts: a knowledge base containing general knowledge related to the problem, a description of a particular instance of the problem, and a generator of possible solutions. Informally the latter is used to generate candidates for the solutions of the problem instance, while the first two parts are used to check if a candidate is indeed a solution. The general knowledge about the Hamiltonian paths problem from Section 6.1 is given by rules (h1)–(h6). The candidate solution generation consists of rule (h7) (or, alternatively, (h7c) and (h7d)). The problem instance corresponding to graph 6.1 is given by (h_graph). Similarly for other examples. It may be interesting to notice that in the Hamiltonian path example, relevant knowledge has a form of the definition of Hamiltonian path which is completely separate from generation.

In the Sudoku examples the definition of a solution of the Sudoku puzzle consists of rules (s1)–(s5). Note that this includes the generating rule (which says that such a solution must be a function assigning numbers to coordinates). The specific instance is given by (s_puzzle).

In the Mystery puzzle the precise mathematical definition of the solution does not exist. Instead we have rules (r3)–(r9) containing commonsense knowledge about the meaning of terms used in the story. As was mentioned above, these rules are not the definitions of the corresponding relations. They only contain partial knowledge about those relation represented by logic programming constraints. As a result a lot of negative information about them is missing. Fortunately this lack of knowledge does not prevent us from correctly solving the problem.

The examples discussed in this chapter cover a number of different computational tasks. In the next few chapters, we will use the same techniques to solve several classical AI problems including those related to planning and diagnostics. First, however, we will present the basic algorithms for computing answer sets of a logic program.

References and Further Reading

Answer set programming as the method of solving non-trivial search problems was first advocated in [72] and [82]. The method only became possible because of the development of a number of efficient answer set solvers [103], [2], [45], [1], [65]. The book, [46], is a great introduction to practical applications of ASP with emphasis on efficiency and multiple advanced features of ASP languages not covered in our book.

Exercises

1. Run the Hamiltonian path program on the graphs given below. Give the answer set(s).

   (a)
   ```
   vertex(a).
   vertex(b).
   vertex(c).
   edge(a,b).
   edge(b,c).
   edge(c,a).
   init(a).
   ```

   (b) Same as exercise (1a) except replace edge(c,a) by edge(a,c).

   (c)
   ```
   vertex(a).
   vertex(b).
   vertex(c).
   ```

vertex(d).
edge(a,b).
edge(b,c).
edge(c,d).
edge(d,a).
edge(c,a).
edge(a,a).
init(a).

(d) Same as exercise (1c) except add
edge(b,a).
edge(c,b).
edge(d,c).
edge(a,d).

2. Give the answer sets for program:
q(a).
q(b).
1{p(X):q(X)}2.

3. Give the answer sets for program:
{p(a),p(b)}2.

4. Suppose we wanted to separate the definition of the Sudoku problem rules from the generating part (s1). We could replace it by:

```
num(1..9).
coord(X,Y) :- num(X), num(Y).
pos(I,X,Y) | -pos(I,X,Y) :- num(I), coord(X,Y).
```

What rules should be added to the program to complete the solution to the Sudoku problem? *Hint:* Encode the information necessary to describe the puzzle requirement that every square must be filled in with a unique number.

5. In the mystery puzzle, it is debatable whether rule (r9) about murderers being in town is a valid assumption. Is it necessary to solve the crime?

6. Use a solver to find the answer set for the mystery program without suppressing the output of other predicates.

(a) Does the program correctly reject the rest of Ben's testimony?

(b) What would happen if anyone could be lying? Remove rule (r1) and test if the program conforms to your intuition.

(c) What if we were to assume that friends do not murder each other? For simplicity, just use rule

```
:- person(P), holds(friends(P,vinny)), murderer(P).
```

which states that a friend of Vinny's would not murder him. What happens when you add this rule to the original program? Why?

7. Given a round table with ten chairs and a group of ten people, some of whom are married and some of whom do not like each other, use ASP to find a seating assignment for members of this group such that husbands and wives are seated next to each other and no neighbors dislike each other.

Chapter 7

Algorithms for Computing Answer Sets

In this chapter we give a short introduction to algorithms for computing answer sets of logic programs. These algorithms form the basis for the implementations of answer set solvers and query-answering systems used in previous chapters. For simplicity we limit ourselves to logic programs without classical $\neg$ and constraints (rules with an empty head). This is not a serious restriction since $\neg$ and constraints can always be eliminated from any program Π using Proposition 2.4.4.

The algorithms which we will describe can be viewed as typical examples of generate-and-test reasoning algorithms. They have their roots in the so called Davis-Putnam procedure for finding models of propositional formulas. Understanding this procedure is a stepping-stone to understanding the generate-and-test algorithms implemented in ASP solvers.

7.1 Finding Models of Propositional Formulas

The Davis-Putnam procedure forms the basis of satisfiability solvers; it finds models of propositional formulas by traversing a tree of all possible truth assignments for the variables in that formula. For example, let F be a propositional formula, say

$$F = (X_1 \vee X_2) \wedge \neg X_2.$$

The tree in Figure 7.1 shows all possible assignments of truth values to variables of F. The algorithm traverses the tree looking for a path satisfying

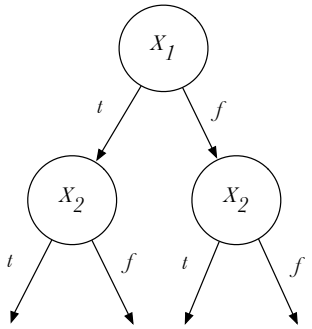


Figure 7.1: All Possible Truth Value Assignments for X_1 and X_2

F. A simple depth-first search with backtracking allows us to find a path $\langle t, f\rangle$ which is a model of F.

It is clear that the efficiency of this algorithm depends on the ordering of variables and the efficiency of testing if a vector $\bar{X}$ of variables falsifies a propositional formula. For instance a model of $F \vee (X_3 \vee \neg X_3)$ where F is an arbitrary formula could be found quickly if we were to start with assigning a value to X_3, and if our checking part were smart enough to recognize that no other variable needs to be examined — an arbitrary assignment of values to these variables would produce a model. Numerous papers address the problem of finding an effective ordering, data structures and algorithms which would allow for the efficient implementations of the Davis-Putnam procedure. Here we describe a simple basic algorithm. Before we go into the details of the algorithm, we define some necessary terminology:

- A *signature* is a set of propositional variables.
- A *clause* over signature Σ is a set $\{l_1, \ldots, l_n\}$ of literals of Σ denoting the disjunction $l_1 \vee \cdots \vee l_n$.
- A *formula* is a set $\{C_1, \ldots, C_m\}$ of clauses denoting the conjunction $C_1 \wedge \cdots \wedge C_m$.[1]
- 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; otherwise p is *false* in I.

[1] Of course, the standard definition of propositional formula is more general but the restriction is not overly strong since any such formula can be equivalently written as a conjunction of clauses.

- An *interpretation* is a partial interpretation defined on all variables of the language.
- A *model* of a formula F is an interpretation which makes the formula *true*.
- A formula is called *satisfiable* if it has a model.
- Partial interpretation I_2 is *compatible* with partial interpretation I_1 if $I_1 \subseteq I_2$.
- Variable p is *undefined* in partial interpretation I if it does not belong to the domain of I.

We are interested in the development of an algorithm $Sat(F)$ which takes a formula F as an input and returns a model of F if F is satisfiable and *false* otherwise. To define the algorithm recursively, we first describe a function $Sat(I, F)$ which searches for a model of formula F over signature Σ compatible with partial interpretation I. To find a model of F we will then simply call $Sat(\emptyset, F)$. Here is the code, followed by a more detailed explanation.

```
function Sat
   input: partial interpretation I0 and formula F0;
   output: a pair ⟨I, true⟩ where I is a model of F0 compatible with I0;
           ⟨I0, false⟩ if no such model exists;
var F : formula, I : partial interpretation, X : boolean;
begin
   F := F0;
   I := I0;
   ⟨F, I, X⟩ := Cons(F, I);
   if X = false then
      return ⟨I0, false⟩;
   if F = ∅ then
      return ⟨I, true⟩;
   select variable p undefined in I;
   ⟨I, X⟩ := Sat(I, F ∪ {p});
   if X = true then
      return ⟨I, true⟩;
   return Sat(I, F ∪ {p̄});
end;
```

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

interpretation. (Below we will describe a particular implementation of function $Cons$ which is called *unit propagation*.) Next we have two termination conditions. If $Cons$ found a contradiction Sat returns *false*. The second condition checks for success. Note that, if $F = \emptyset$ then every element of F is vacuously satisfiable and so is F; thus, the function will return *true* together with the new I. It is not difficult to show that I is a desired model of F. If neither condition holds, Sat makes a non-deterministic choice of an yet undefined variable p of F and calls itself recursively.

Now we are ready to define our version of function $Cons$.

function $Cons$ [Unit Propagation];
  input: partial interpretation I_0 and formula F_0 with signature Σ_0;
  output: $\langle F, I, true\rangle$ where I is a partial interpretation such that $I_0 \subseteq I$
    and F is a formula over signature Σ_0 such that
    M is a model of F_0 compatible with I_0 iff
    M is a model of F compatible with I.
  $\langle F_0, I_0, false\rangle$ if no such model exists.
var F : formula, I : partial interpretation;
begin
  $F := F_0$;
  $I := I_0$;
  while F contains a unary clause $\{l\}$ **do**
    remove from F all clauses containing l;
    remove from F all occurrences of $\bar{l}$;
    $I := I \cup \{l\}$;
  if $\{\ \} \in F$ **then**
    return $\langle F_0, I_0, false\rangle$;
  return $\langle F, I, true\rangle$;
end;

Note that the condition $\{\ \} \in F$ is used to check if F is already shown to be unsatisfiable. To better understand the condition note that to satisfy a clause we need to satisfy at least one element in it. Hence, the empty clause is unsatisfiable, and so is the collection F of clauses containing the empty clause. Note also that each call to $Cons$ eliminates occurrences of at least one literal from F, and hence Sat will eventually terminate.

Example 7.1.1. *(Tracing Sat)*
To illustrate the algorithm let us trace the computation of $Sat(I_0, F_0)$ where

$$I_0 = \emptyset$$

and

$$F_0 = \{\{X_1\}, \{\neg X_1, X_2, X_3\}, \{\neg X_1, X_4\}\}$$

First function *Cons* will go through two steps of the loop and return *true* together with

$$F = \{\{X_2, X_3\}\}$$

and

$$I = \{X_1, X_4\}.$$

It is easy to check that M is a model of F compatible with I iff it is a model of F_0 compatible with I_0. (This, of course, will be true after every iteration of the loop.) The termination conditions in *Sat* are not satisfied, so the algorithm will select a variable occurring in F and not occurring in I, say X_2, and call

$$Sat(\{X_1, X_4\}, \{\{X_2, X_3\}, \{X_2\}\}).$$

The new call to *Cons* will return $I = \{X_1, X_4, X_2\}$ and $F = \emptyset$. Now the second termination condition is satisfied and hence *Sat* returns $\langle\{X_1, X_4, X_2\}, true\rangle$.

Notice that I is a partial interpretation and hence, strictly speaking, is not a model of the original formula F_0. To make it a model we simply assign arbitrary values to variables which are not in I (in our case, to X_3).

The *Sat* algorithm is a typical example of the generate and test reasoning algorithms used for solving many complex problems in computer science. The practical efficiency of such algorithms depends on several factors including:

1. the quality of the ordering of variables which determine the selection of an undefined variable in the *Sat* algorithm. Such orderings are frequently done by *heuristics* — rules of thumb proved to be useful by experience. One can, for instance, use the heuristic which selects the variable and its boolean value which satisfies the maximum number of yet unsatisfied clauses. Sometimes it may be useful to select a variable with the maximum number of occurrences in clauses of minimum length — this increases our chances of arriving at an unsatisfiable clause or of obtaining a unary clause. In many cases the useful heuristics are much more complex and are based on the previous history of computation (e.g., a variable is selected from a clause which has caused the maximum number of conflicts). In all these cases the quality of a heuristic for a particular class of problems is normally determined by extensive experimentation.

2. the quality of the procedure computing consequences of a program and a new partial interpretation. The procedure should balance the ability

to compute a large number of consequences (which would of course allow to substantially decrease the search space) and the efficiency of computing these consequences.

3. the quality of data structures and the corresponding algorithms used in the actual implementation.

There is a substantial body of knowledge accumulated by computer scientists which allows the designers of solvers to make good choices related to all these (and other) factors. In particular the designers of answer set solvers learned a great deal from the research on the design of *Sat* solvers. We now describe two algorithms which adapt the basic ideas of *Sat* to the design of ASP solvers.

7.2 Finding Answer Sets of Logic Programs

The algorithm for computing answer sets of a logic program consists of two steps. In the first step of the computation, the algorithm replaces a program Π, which normally contains variables, by its ground instantiation $ground(\Pi)$. In practical systems $ground(\Pi)$ is not the full set of all syntactically constructible instances of the rules of Π; rather, it is an (often much smaller) subset having precisely the same answer sets as Π. The ability of the grounding procedure to construct small ground instantiation of the program may dramatically affect the performance of the entire system. The grounding techniques implemented by answer set solvers are rather sophisticated. Among other things, they utilize algorithms from deductive databases, and require a good understanding of the relationship between various semantics of logic programming. Nevertheless, the grounding of a program containing variables over large domains can be prohibitively large, which lead to the recent development of answer set solvers which only do partial grounding.

Once the variables have been eliminated from Π, the heart of the computation is then performed by a function *Solver*. In this book we present two versions of this function, $Solver1$ and $Solver2$, which work for programs without $\neg$, *or* and constraints. The first one uses a simple algorithm for computing consequences of the guessing decisions and is readily expandable to disjunctive programs. The second has a more sophisticated computation of consequences and is more efficient but tailored toward non-disjunctive programs. Even though the structure of *Solver* is very similar to that of *Sat*, it works on different objects with sightly different definitions of interpretation, consistency, etc. Here is the corresponding terminology:

- 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 *not* l we denote *not* $p(\bar{t})$ if $l = p(\bar{t})$ and $p(\bar{t})$ if $l =$ *not* $p(\bar{t})$.
- A set of e-literals is called *consistent* if it contains no e-literals of the form l and *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 *not* $p \in I$, then I is called an *interpretation*.
- An atom p can be *true* ($p \in I$), *false* (*not* $p \in I$) or *undefined* ($p \notin I$, *not* $p \notin I$), with respect to an interpretation I.
- An answer set A of a program Π will be represented as an interpretation I of the signature of Π such that

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

- 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 *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.

Now we are ready to define our first answer set finding algorithm, $Solver1$.

7.2.1 The First Solver

The Main Program

function $Solver1$
  input: partial interpretation I_0 and program Π_0;
  output: $\langle I, true\rangle$ where I is an answer set of Π_0 compatible with I_0;
    $\langle I_0, false\rangle$ if no such answer set exists;
var Π : program; I : set of e-literals; X : boolean;
begin
  $\Pi := \Pi_0$;
  $I := I_0$;
  $\langle \Pi, I, X\rangle := Cons1(I, \Pi)$;
  if $X = false$ **then**
    return $\langle I_0, false\rangle$;
  if no atom is undefined in I **then**

```
      if IsAnswerSet(I, Π) then
        return ⟨I, true⟩
      return ⟨I_0, false⟩;
    select a ground atom p undefined in I;
    ⟨I, X⟩ := Solver1(I ∪ {p}, Π);
    if X = true then
      return ⟨I, X⟩;
    return Solver1((I \ {p}) ∪ {not p}, Π)
end;
```

*Solver*1 starts by initializing variables Π and I and calling a function $Cons1(\Pi, I)$ which expands I by 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, *Cons*1 returns the new partial interpretation I and the simplified program, together with boolean value *true*. Otherwise, it returns the original input together with *false*.

The call to *Cons*1 is followed by two termination conditions. First *Solver*1 checks if *Cons*1 returns *false*. If so, I_0 is inconsistent with Π_0 and *Solver*1 returns *false*. Second it checks whether partial interpretation I is complete, i.e., if for every atom p either p or *not* p belongs to I. If this is the case then *Solver*1 checks if I is an answer set. If the answer is *yes* then the function returns *true* together with I. Otherwise, there is no answer set of Π_0 compatible with I_0 and the function returns *false*. If Π still contains some undefined atoms, the function selects such an atom p; *Solver*1 is (recursively) called to explore whether I can be expanded to an answer set of Π containing p. If this is impossible, then *Solver*1 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. Upon failure of both calls, the function returns *false*, as I cannot be expanded to any answer set.

As in the case of *Sat*, an answer set of Π is computed by calling $Solver1(\emptyset, \Pi)$.

Lower Bound — First Refinement of the Consequence Function

Now let us discuss a comparatively simple version of function *Cons*1 traditionally called *LB* (where *LB* stands for *lower bound*). A different version of *Cons*1 will be discussed later. In what follows we use the traditional name.

Function *LB* computes the 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.

Let $1 \leq i \leq 3$ be one of the first three inference rules defined above, Π be a program, I be an 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, $i\text{-}cons(i, \Pi, I, r) = \emptyset$. (Notice that, for the first inference rule, function $i\text{-}cons$ does not require Π as a parameter, but we nevertheless include it for uniformity of notation.) If $i = 4$ then $i\text{-}cons(i, \Pi, I)$ is the set of literals which do not occur in the heads of rules of Π not falsified by I.

Function $LB(I, \Pi)$ can be computed as follows:

```
function LB
  input: partial interpretation I_0 and program Π_0 with signature Σ_0.
  output: ⟨Π, I, true⟩ where I is a partial interpretation such that I_0 ⊆ I
            and Π is a program with signature Σ_0 such that for every A
            A is an answer set of Π_0 compatible with I_0 iff
            A is an answer set of Π compatible with I.
          ⟨Π_0, I_0, false⟩ if there is no answer set of Π_0 compatible with I_0.
var I, T : set of e-literals; Π : program;
begin
  I := I_0;
  Π := Π_0;
  repeat
    T := I;
    remove from Π all the rules whose bodies are falsified by I;
    remove from the bodies of rules of Π all e-literals satisfied by I;
    select an inference rule i from (1)–(4);
    if 1 ≤ i ≤ 3 then
      for every r ∈ Π satisfied by the if part of inference rule i
        I := I ∪ i-cons(i, Π, I, r);
    else
      I := I ∪ i-cons(4, Π, I);
  until I = T;
  if I is consistent then
    return ⟨Π, I, true⟩;
  return ⟨Π_0, I_0, false⟩
end
```

First, function LB initializes variables I and Π and simplifies Π by removing from it any rules and e-literals made useless by I. Then LB selects an inference rule (1)–(4) and uses it to expand I by e-literals derived by

this rule. The process continues until no more e-literals can be added. If the resulting set I is consistent, LB returns I, the simplified Π, and *true*; otherwise, it returns the original parameters and *false*.

The following example illustrates the function:

Example 7.2.1. *(Tracing LB)*
Consider program P_1:

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

with the signature determined by the rules of the program, and trace the execution of $LB(\emptyset, P_1)$. Initially I and T are set to $\emptyset$, while Π is set to P_1. Application of the two simplifying rules of LB does not change Π. Now the function non-deterministically selects an inference rule. Suppose it selects inference rule (1). The bodies of the first two rules of the program are not satisfied by $\emptyset$. The body of the third rule is empty and, hence, is satisfied by any set of e-literals, including the empty one. Thus, applying inference rule (1) to the program produces one consequence, $q(a)$, which is added to I.

On the next iteration the simplification deletes the first program rule and hence Π consists of the second and third rule of P_1. Suppose that the inference rule selected next is (4). Since neither $p(a)$ nor $q(b)$ belongs to the heads of the rules of the simplified program, I becomes $\{q(a),\ not\ p(a),\ not\ q(b)\}$.

On the third iteration Π is further simplified to become $\{p(b).\quad q(a).\}$; LB again selects inference rule (1), applies it to the first rule of Π and sets I to $\{q(a),\ not\ p(a),\ not\ q(b),\ p(b)\}$. Since the next iteration does not produce any new consequences and I is consistent, LB returns $\langle I, \Pi, true\rangle$.

Defining $IsAnswerSet$

Now we discuss a simple algorithm for computing function $IsAnswerSet(I, \Pi)$:

function $IsAnswerSet$
  input: interpretation I and program Π
  output: *true* if I is an answer set of Π; *false* otherwise
begin
  Compute the reduct, Π^I of Π with respect to I;
  Compute an answer set, A, of Π^I;
  Check whether $A = atoms(I)$ and return the result;
end;

The first and the last steps of the function are relatively straightforward but the second one requires some elaboration. We need the following concepts.

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 signature of Π. (In other words Π contains no default negation.)

Let Π be a definite program and T_Π be an operator defined on sets of atoms from the signature of Π as follows:

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

Intuitively, $T_\Pi(A)$ returns conclusions of all rules of Π whose bodies are satisfied by A. We use this operator to describe function $Least(\Pi)$ which takes as a parameter a finite definite program Π and returns its answer set. Since by definition of the reduct Π^I is obviously definite, this function can be used to find its answer set.

function *Least*
  input: a definite program Π
  output: the answer set of Π
var X, X_0 : set of atoms;
begin
  $X := \emptyset$;
  repeat
    $X_0 := X$;
    $X := T_\Pi(X)$;
  until $X = X_0$;
  return X
end

The following example illustrates the computation.

Example 7.2.2. *(Least)*
Consider a program

$$p(a) \leftarrow q(a).$$
$$q(a).$$

Its answer set is obtained as the result of the following computation:

$$T_\Pi(\emptyset) \Rightarrow T_\Pi(\{q(a)\}) \Rightarrow T_\Pi(\{q(a), p(a)\}) \Rightarrow \{q(a), p(a)\}.$$

It may be instructive to check that applying the same algorithm to $p(a) \leftarrow p(a)$ returns $\emptyset$.

This completes the refinement of our function $Solver1$. Now let us illustrate how it works by tracing several simple examples. (Remember that LB is just a different name for $Cons1$ from $Solver1$.)

Tracing $Solver1$

Example 7.2.3. *(A Simple Case)*
Consider program P_1:

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

from the previous example. To compute the answer set of P_1, we call $Solver1(\emptyset, P_1)$ which starts by initializing Π and I and calling $LB(\emptyset, \Pi)$. As discussed in Example 7.2.1 function LB sets Π to

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

I to $\{q(a),\ not\ p(a),\ not\ q(b),\ p(b)\}$, and X to *true*. $Solver1$ discovers that after the first application of LB, all the ground atoms from the signature of P_1 are defined. What is left is to check if I is an answer set of Π. $Solver1$ calls $IsAnswerSet(I, \Pi)$ which first computes the reduct Π^I of Π with respect to I. By the definition of reduct we have that Π^I is

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

$Least(\Pi^I) = \{q(a), p(b)\}$ which is equal to $atoms(I)$. Hence, $IsAnswerSet(I, \Pi)$ returns *true* and the solver returns *true* together with the answer set I of the program.

It is easy to see that program P_1 from the above example is stratified (see Chapter 2) and, hence, has at most one answer set. Our second example illustrates how $Solver1$ works on a program with multiple answer sets.

Example 7.2.4. *(Program with Multiple Answer Sets)*
Now let us compute an answer set of a program P_2:

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

$Solver1(\emptyset, P_2)$ initializes I and Π and calls $LB(\emptyset, \Pi)$. It is not difficult to see that Π cannot be simplified by $\emptyset$ and no inference rule used by LB is applicable to the program. Hence LB returns *true* without changing I and Π. $I = \emptyset$, no atom is yet defined and $Solver1$ starts the selection process. Let us assume that $Solver1$ selects $p(a)$ and makes the recursive call $Solver1(\{p(a)\}, \Pi)$, which, in turn, calls $LB(\{p(a)\}, \Pi)$. After the simplification Π becomes

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

Suppose that LB selects inference rule (2). It is applicable to the rule of the program, so LB computes a new consequence, $not\ q(a)$; I becomes $\{p(a),\ not\ q(a)\}$ and Π becomes

$$p(a).$$

(Another possibility is to select inference rule (4) instead of (2); this would lead to the same result.) Next, function $IsAnswerSet(\{p(a), not\ q(a)\}, \Pi)$ computes Π^I:

$$p(a).$$

and discovers that $Least(\Pi^I) = atoms(I)$; hence, $\{p(a)\}$ is an answer set of P_2. A different choice of a selected e-literal would lead to finding another answer set of the program, $\{q(a)\}$.

Now let us consider a program without answer sets.

Example 7.2.5. *(Detecting Inconsistency)*
Consider a program P_3:

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

and trace the execution of $Solver1(\emptyset, P_3)$. After the initialization the function calls $LB(\emptyset, \Pi)$. Since no simplification is possible and no inference rule is applicable to Π, $LB(\emptyset, \Pi)$ returns *true* without changing $I = \emptyset$ and $\Pi = P_3$, and $Solver1$ starts its selection process. If $p(a)$ is selected first then, the solver recursively calls $Solver1(\{p(a)\}, \Pi)$. This, in turn, calls $LB(\{p(a)\}, \Pi)$. After the simplification $\Pi = \emptyset$. By inference rule (4) LB concludes $not\ p(a)$, detects inconsistency, and returns *false* together with $\{p(a)\}$. The next call is $Solver1(\{not\ p(a)\}, \Pi)$. This time $LB(\{not\ p(a)\}, \Pi)$ simplifies Π to $p(a)$. Using rule (1) LB obtains inconsistency and $Solver1$ returns *false*. The program has no answer sets.

So far in all our examples $IsAnswerSet$ always returned *true*. In the next example this is not the case.

Example 7.2.6. *(The Importance of $IsAnswerSet$)*
Consider a program P_4:

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

and call $Solver1(\emptyset, P_4)$. The program cannot be simplified by the $\emptyset$, none of the four inference rules of LB are applicable to this program, so $LB(\emptyset, \Pi)$ returns *true* without changing I and Π. Now $Solver1$ may select $p(a)$ and call $Solver1(\{p(a)\}, \Pi)$. $LB(\{p(a)\}, \Pi)$ returns *true* and sets I to $\{p(a)\}$ and Π to $\{p(a).\}$. All atoms of P_4 are now defined and $Solver1$ calls $IsAnswerSet(\{p(a)\}, \Pi)$. The reduct $\Pi^I = \Pi$. Obviously, $Least(\Pi) = \emptyset$. Now $IsAnswerSet(\{p(a)\}, \Pi)$ compares $\emptyset$ and $\{p(a)\}$, discovers that they are not equal, and returns *false*. $Solver1$ tries another choice, selects *not* $p(a)$, and eventually correctly returns $\{not\ p(a), true\}$.

7.2.2 The Second Solver

In this section we give a different algorithm for computing answer sets of a program. The new algorithm, called $Solver2$, uses a more powerful method for computing consequences than the one used by function LB. In fact, the method is so powerful it makes checking whether the computed interpretation is indeed an answer set of the program unnecessary.

The new consequences-computing function $Cons2$ expands $Cons1$ by computing more negative consequences of the program. (For example, $Cons2$ is able to compute *not* p as a consequence of a program $p(a) \leftarrow p(a)$ with respect to $I = \emptyset$, while $Cons1$ can not.) The computation of these new consequences is done by a new function, $UB(I, \Pi)$ called the **upper bound** of I with respect to Π. Eventually we define $Cons2$ in terms of both functions, LB and UB.

function UB
  input: partial interpretation I_0 and program Π_0.
  output: A set N of e-literals of the form *not* p such that for every A
    A is an answer set of Π_0 compatible with I_0 iff
    A is an answer set of Π_0 compatible with $N \cup I_0$.
var M : partial interpretation; Π : program;
begin
  Let Π be the definite program obtained from Π_0 by
    removing from Π_0 all the rules whose bodies are falsified by I_0 and then
    removing all other occurrences of e-literals of the form *not* p;
  $M := Least(\Pi)$;

```
  M := {not p : p ∉ M};
  return M;
end;
```

Example 7.2.7. *(Upper Bound 1)*
Let us trace $UB(\emptyset, \Pi_4)$ where Π_4 is

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

It is easy to see that no simplification of Π_4 by $\emptyset$ is possible and that $Least(\Pi_4) = \emptyset$. Since the only atom in the signature of Π_4 is $p(a)$, function $UB(\emptyset, \Pi_4)$ returns $\{not\ p(a)\}$.

Example 7.2.8. *(Upper Bound 2)*
Consider now a program Π_5

$$p(a) \leftarrow s(a),\ not\ q(a).$$

This time $UB(\emptyset, \Pi_5)$ simplifies Π_5 and constructs program Π:

$$p(a) \leftarrow s(a).$$

$Least(\Pi)$ returns $\emptyset$. There are three atoms in the signature of Π_5: $p(a)$, $q(a)$ and $s(a)$. Hence UB returns $\{not\ p(a), not\ q(a), not\ s(a)\}$.

Now we are ready to define $Cons2$. The function computes consequences of I with respect to Π using both LB and UB.

```
function Cons2
  input: partial interpretation I_0 and program Π_0 with signature Σ_0.
  output: ⟨Π, I, true⟩ where I is a partial interpretation such that I_0 ⊆ I
            and Π is a program with signature Σ_0 such that
            A is an answer set of Π_0 compatible with I_0 iff
            A is an answer set of Π compatible with I.
          ⟨Π_0, I_0, false⟩ if there is no answer set of Π_0 compatible with I_0.
var I, T : set of e-literals; Π : program; X : boolean;
begin
  I := I_0;
  Π := Π_0;
  ⟨Π, I, X⟩ := LB(I, Π);
  if X = true then
    T := UB(I, Π);
    I := I ∪ T;
    if I is consistent then
      return ⟨Π, I, true⟩;
  return ⟨Π_0, I_0, false⟩
end;
```

Example 7.2.9. *(Cons2 1)*
Consider a program Π_4 from Example 7.2.7 and trace $Cons2(\emptyset, \Pi_4)$. Recall that Π_4 consists of rule

$$p(a) \leftarrow p(a)$$

and that $LB(\emptyset, \Pi_4)$ returns $\langle \Pi_4, \emptyset, true \rangle$. As shown before, T will be set to $\{not\ p(a)\}$ and the function will return

$$\langle \Pi_4, \{not\ p(a)\}, true \rangle.$$

Example 7.2.10. *(Cons2 2)*
Consider now a program Π_6

$$p(a) \leftarrow s(a), p(a),\ not\ q(a).$$
$$s(a).$$

and trace $Cons2(\emptyset, \Pi_6)$. This time LB will return $\langle \Pi, \{s(a),\ not\ q(a)\}, true \rangle$ where Π is

$$p(a) \leftarrow p(a).$$
$$s(a).$$

The UB will set T to $\{\ not\ p(a)\}$ and $Cons2$ will return

$$\langle \Pi, \{s(a),\ not\ q(a),\ not\ p(a)\}, true \rangle.$$

Now we can give the new answer set finding algorithm $Solver2$:

function $Solver2$
  input: partial interpretation I_0 and program Π_0;
  output: $\langle I, true \rangle$ where I is an answer set of Π_0 compatible with I_0.
    $\langle I_0, false \rangle$ if no such answer set exists;
var Π : program; I : set of e-literals; X : boolean;
begin
  $\Pi := \Pi_0$;
  $I := I_0$;
  $\langle \Pi, I, X \rangle := Cons2(I, \Pi)$;
  if $X = false$ **then**
    return $\langle I_0, false \rangle$;
  if no atom is undefined in I **then**
    return $\langle I, true \rangle$;
  select a ground atom p undefined in I;
  $\langle I, X \rangle := Solver2(I \cup \{p\}, \Pi)$;
  if $X = true$ **then**
    return $\langle I, X \rangle$;
  return $Solver2(I \cup \{not\ p\}, \Pi)$
end;

In comparing the efficiency of the two answer set solvers one can see that *Solver*1 spends less time computing the consequences of the program but pays for this by spending additional time checking if the computed interpretation is an answer set. *Solver*2 spends more time computing consequences but does not need the additional checking. Extensive experimentation has shown that the second method of computing this function is usually more efficient. In this case spending more time in computing consequences and avoiding the checking increases the efficiency of the solver. Note, however, that correctness of our second solver is much less obvious than that of the first. In fact the theorem showing that the interpretation computed by *Solver*2 is an answer set of the program is rather non-trivial.

Another way to improve performance of answer set solvers is to employ a good heuristic for the selection of undefined ground atoms. Selection of a good heuristic is an interesting and important topic for any generate-and-test algorithm. There is a substantial body of research related to the subject. In fact, the area of search and heuristics deserves its own course. In this section we only briefly mention a particular heuristic used in some answer set solvers. The heuristic is rather application-independent and leads to good performance across a range of applications. It can be taken as a starting point for developing more-refined heuristics for particular application areas. First we replace the selection of an atom by selection of an e-literal. There is no reason to try an atom p first. Sometimes *not* p can do as well or better. Next we try to select an e-literal that has the greatest possibility of changing current partial interpretation I, thereby helping the algorithm to discover conflicts or find complete sets of e-literals with a minimal number of choices. For illustrative purposes we give a simple refinement of this idea. First we need a definition. A rule

$$p_0 \leftarrow p_1, \ldots, p_m, \ not\ p_{m+1}, \ldots\ not\ p_n$$

is called *applicable* with respect to a partial interpretation I if

1. $\{p_1, \ldots, p_m\} \subseteq I$,
2. there is no k such that $m+1 \leq k \leq n$ and $p_k \in I$, and
3. $p_0 \notin I$.

Our heuristics *selects an e-literal not p from the body of an applicable rule with the least number of negated atoms not belonging to I*. For instance, if $I = \{b,\ not\ f\}$ and Π consists of two rules

$$a \leftarrow b,\ not\ c,\ not\ d$$
$$a \leftarrow b,\ not\ e,\ not\ f$$

the heuristic selects *not* e. The selection ensures that a is immediately added to I. If we were to select, say, *not* c, the expansion of I would have to wait for the next selection of an atom.

7.2.3 Finding Answer Sets of Disjunctive Programs

So far we only discussed solvers for logic programs not containing disjunction. Dealing with disjunctive programs requires some additional ideas which we will only briefly discuss in this section. This additional difficulty is not surprising since it follows from theoretical analysis of complexity of these two tasks. Finding answer sets of programs not containing epistemic disjunction is an NP-complete problem; therefore, testing (in our case done by $IsAnswerSet$) can be done in polynomial time (or even eliminated altogether). The problem of computing answer sets for disjunctive programs belongs to a higher complexity class and, hence, checking if a given set of literals is an answer set cannot always be done in polynomial time. (There are, however, large classes of disjunctive logic programs for which the complexity of computing answer sets is NP-complete. Actually *all* of the disjunctive programs we considered so far belong to such a class.) This complexity consideration implies that $Solver2$, which does not check if a computed interpretation is an answer set of a program, cannot be easily adapted to deal with disjunction. $Solver1$, however, is more amenable to change. All we need to do is to replace function $IsAnswerSet$ by a more complex version which works for disjunctive programs. Instead of using a simple computation incorporated in $Least$, we will need to check that I satisfies all the rules of the program and, more importantly, that there is no I' such that $atoms(I') \subset atoms(I)$ which also satisfies these rules. This second condition is exactly the one which adds complexity to the algorithm.

7.2.4 Answering Queries

The method for computing answer sets of ASP programs illustrated in the previous section can be used to implement STUDENT-like query-answering systems of ASP. Suppose, for instance, that we are given a consistent program Π and would like to know the answer to a ground query q where q is a literal. The following algorithm allows us to answer this question. (Note that by the call to $Solver$ below, we mean $Solver1$ or $Solver2$, whichever implementation is appropriate.)

function *Query*
input: ground literal l and consistent program Π.

```
   output: yes if l is true in all answer sets of Π,
           no if l̄ is true in all answer sets of Π,
           unknown otherwise.
begin
if Solver(∅, Π ∪ {← not l}) = false then
   return yes
if Solver(∅, Π ∪ {← not l̄}) = false then
   return no
return unknown
end;
```

Of course the consistency of Π can be checked in advance by a single call to $Solver(\emptyset, \Pi)$.

To answer a query $q_1 \wedge \cdots \wedge q_n$ where qs are ground literals, we expand Π by rules:

$$\begin{array}{rcl} q & \leftarrow & q_1, \ldots, q_n \\ \neg q & \leftarrow & \neg q_1 \\ & \vdots & \\ \neg q & \leftarrow & \neg q_n \end{array}$$

where q is a new atom. Denote the new program by Π'. Now the question can be answered by a single call to $Query(q, \Pi')$. A similar technique can be used to answer disjunctive query q_1 *or* ... *or* q_n. In this case Π should be expanded by rules:

$$\begin{array}{rcl} q & \leftarrow & q_1 \\ & \vdots & \\ q & \leftarrow & q_n \\ \neg q & \leftarrow & \neg q_1, \ldots, \neg q_n \end{array}$$

Not surprisingly, the approach does not apply to non-ground queries. They can be answered by a simple (but not always efficient) algorithm which computes and stores all the answer sets of Π. The analysis of these answer sets allows to return all ground terms t such that query $q(t)$ is true in all the answer sets.

Summary

In this chapter we started by outlining a *Sat* algorithm for finding models of formulas of propositional logic. This algorithm can be viewed as a typical example of the generate-and-test reasoning algorithm used for solving

many complex problems in computer science. We briefly discussed the general structure of such algorithms and several methods for improving their efficiency. Next we showed how the basic ideas of *Sat* could be adapted to the problem of computing answer sets of non-disjunctive logic programs. In particular we presented two versions of such an algorithm which differ primarily by the functions they use for computing consequences of a program and the new partial interpretation. A short discussion explained how these algorithms can be used to answer simple queries and how the first one of them could be adapted to work for disjunctive programs. Of course this is only a brief introduction. Serious study of *Sat*-like methods used to solve problems of non-polynomial complexity requires much more time. Moreover, we are rather far of fully understanding these algorithms. There are many unanswered and fascinating questions which remain — after all, many computational problems which need solutions are not polynomial. Discovering methods which allow us to find practical solutions of such problems is crucial for our understanding of computation and for many applications.

Exercises

1. (a) Use the first solver to compute answer set(s) of program Π. (Assume that a and b are the only constants of the signature of Π.)

$$\Pi \begin{cases} p(a) \leftarrow \ not\ \neg p(b). \\ \neg p(b) \leftarrow \ not\ p(a). \\ p(a) \leftarrow \ not\ r(a). \\ r(a) \leftarrow \ not\ p(a). \\ q(X) \leftarrow \ not\ p(X). \\ \neg p(X) \leftarrow \ not\ p(X). \\ \leftarrow r(a). \\ r(b). \end{cases}$$

   (b) How does Π answer queries

$?q(a)$	$?q(b)$
$?r(a)$	$?r(b)$

Chapter 8

Modeling Dynamic Domains

So far, we have limited our attention to *static* domains — no attempt was made to represent a domain's evolution in time. Recall from the introduction that we are interested in agents that are intended to populate *dynamic*, changing domains, and should, therefore, be able to plan, explain unexpected observations, and do other types of reasoning requiring the ability to predict effects of series of complex actions. This can be done only if the agent has sufficient knowledge about actions and their effects. In this chapter we discuss one of several current approaches to representing and reasoning with such knowledge. We will start by looking at an extended example in order to illustrate some of the issues that arise when we attempt to represent actions and their effects on the world. Once some of the issues become clear, we will present a general, formal theory of actions and change, with further examples on how to apply it to various domains. The theory views the world as a dynamic system whose states are changed by actions, and provides an "action language" for describing such systems. This language allows concise and mathematically accurate descriptions of the system's states and possible state-action-state transitions; it allows us to represent dynamic domains and their laws. Such representations can be translated into ASP programs that will be used to give the precise semantics of the language. Later, we show how this and similar translations can be used to answer queries about the effects of actions in a given situation. In keeping with our plan to separate the representation of our world from using that representation to perform intelligent tasks, we save the discussion of planning and other types of reasoning for later chapters in which we will apply the knowledge of answer set programming that we presented previously.

8.1 The Blocks World — A Historic Example

In 1966 a group of researchers at SRI International (then known as the Stanford Research Institute) decided to build a reasoning robot. The result was Shakey, (named for its jerky motions). The robot could receive a description of a goal from a human, make a plan, and execute the necessary actions to achieve the goal.

One task that the researchers believed a robot should be able to perform was to move blocks and create various configurations with them. This blocks world became what some have termed the "Drosophila of AI"[1] because this simple domain provided so much research potential. It has been especially popular in the planning community because of its simplicity, and its search space that grows rapidly with addition of blocks.

The work on Shakey that continued through 1972 brought to light many of the challenges involved in building and programming such a machine. Naturally, there were many hardware challenges. For example, making a robotic arm actually capable of picking something up turned out to be very difficult. Vision is also a very complex topic. A robot may have an onboard camera, but how does it parse out the necessary images? Shakey had to settle for pushing blocks around and it could not evaluate whether it achieved the goal by seeing the result of its actions.

The software challenge turned out to be no less. Many questions arose, such as

- How do we represent knowledge?
- How do we teach a robot to use this knowledge to make plans?
- How do we teach a robot to evaluate if its execution of a plan was successful?
- How should the robot re-plan if there are changes?

Shakey used LISP and a theorem-proving planner called STRIPS. Its planning was domain-specific. Many planners that followed were also domain-specific, difficult to modify, and slow. Huge progress has been made since

[1] The Drosophila Fly, commonly known as the fruit fly, was chosen as a subject of study by geneticists because it was so easy to take care of and reproduced so quickly. It turned out that studying this little fly, so seemingly unlike humans, unlocked many secrets of our own genetic code. The blocks world, often criticized as a "toy" example so unlike the real world, nonetheless allowed scientists to learn much about commonsense reasoning in a changing world.

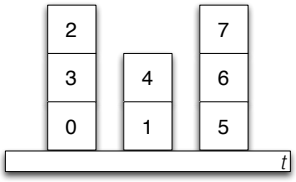


Figure 8.1: Initial Configuration

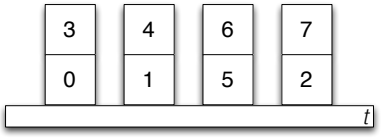


Figure 8.2: Final Configuration

then, both in efficiency of the planners, and in the kind of information they were able to represent. The approach we present separates representation of a domain from the reasoning that will be done in that domain; i.e., teaching a computer about the laws of a world will be separate from teaching it how to use this information in various tasks. This allows us both to use the same method of representation for whatever domain we wish and to use whatever reasoning algorithm we choose with our domain.[2] With the same information, an agent can plan its actions, attempt to explain unexpected events in the world, etc. Separating the representation from the reasoning component allows for clearer and much more elaboration-tolerant programs.

Let's consider how a blocks world problem can be encoded in ASP. First, let us define the particular version of the blocks world that we will use:

> The **basic blocks world** consists of a robotic arm that can manipulate configurations of same-sized cubic blocks on a table. There are limitations to what the robotic arm can do. It can move *unoccupied* blocks, one at a time, onto other unoccupied blocks or onto the table. (An unoccupied block is one that does not have another block stacked on it.) At any given step, a block can be in at most one location; in other words, a block can be directly on top of one other block, or on the table. We do not impose a limit on how tall our towers can be. Our table is big enough to hold all the blocks, even if they are not stacked. We do not take into account spatial relationships of *towers*, just which *blocks* are on top of each other and which blocks are on the table.

Here is an example problem in this domain. Blocks 0–7 are stacked on table t as shown in Figure 8.1. Our robot could, for example, turn this

[2]This does not mean that we cannot use domain-specific information to guide our reasoning. We will show how we can do this with the logic-programming approach in Chapter 9.

configuration into the one shown in Figure 8.2 by putting block 2 on the table and block 7 on block 2.

We would like to be able to write a program to model the transformation of the domain caused by the robotic arm's activity; i.e., given an initial position of blocks and a sequence of actions, our program should be able to answer queries about the positions of blocks after the execution of these actions. You should recall that writing a declarative program involves identifying the objects and the relations between the objects that you are interested in. The blocks and the table are some obvious objects. We would also like to talk about possible locations, so that we do not always have to distinguish between blocks and the table. We will denote blocks by $b0 \ldots b7$ and locations will be blocks plus the table (named t). To describe a configuration of blocks, we will use terms of the form $on(b, l)$ where b is a block and l is a location; the term states that block b is on location l. Thus, a configuration S is a set of terms $on(b, l) \in S$ if and only if b is on l. For instance, the configuration from Figure 8.1 can now be described by a collection of terms

$$\sigma_0 = \{on(b0, t), on(b3, b0), on(b2, b3), on(b1, t), on(b4, b1),$$
$$on(b5, t), on(b6, b5), on(b7, b6)\}.$$

The action of the robotic hand moving block B to location L will be denoted by terms of the form $put(B, L)$. Action $put(b2, t)$ changes our initial configuration into

$$\sigma_1 = \{on(b0, t), on(b3, b0), on(b2, t), on(b1, t), on(b4, b1),$$
$$on(b5, t), on(b6, b5), on(b7, b6)\}.$$

Action $put(b3, b2)$ transforms block configuration σ_1 into configuration

$$\sigma_2 = \{on(b0, t), on(b3, b2), on(b2, t), on(b1, t), on(b4, b1),$$
$$on(b5, t), on(b6, b5), on(b7, b6)\}.$$

This matches Figure 8.2. The execution of a sequence of actions of type $put(B, L)$ in configuration σ_0 determines the system's trajectory

$$\langle \sigma_0, put(b2, t), \sigma_1, put(b3, b2), \sigma_2 \rangle.$$

which describes its behavior.

To describe the changes our system undergoes, we use integers from 0 to some finite n to denote *steps* of the corresponding trajectories. (We limit

the length of the trajectory for computational reasons.) We also distinguish between **fluents** — properties which can be changed by actions (like one block being on top of another), and **statics** — properties which can not (like the state of something being a block). Blocks, locations, configurations, steps, actions, and fluents are the objects in our domain. Now it's time to consider the relationships between them. Let $\langle \sigma_0, a_0, \sigma_1, \ldots, a_n, \sigma_{n+1} \rangle$ be a trajectory of our system. We use relation $holds(f, i)$ to indicate that fluent f is true after the execution of action a_{i-1} in this trajectory. We use relation $occurs(a, i)$ where a is an action and i is a step to describe transitions between states in the trajectory; i.e., the system transitions from configuration to configuration with the occurrence of an action. 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. In our example, we define relation $holds(on(B, L), I)$, which says that block B is on location L at step I. When we want to say that block B was put on location L at step I, we simply say $occurs(put(B, L), I)$.[3]

We begin our program by defining the objects of the domain and the corresponding variables.[4] Note that in our definition of steps, we use a new construct `#const` to define a constant `n` which is then used in a range declaration `step(0..n)` to specify the maximum number of steps.

```
%% blocks:
block(b0).
block(b1).
block(b2).
block(b3).
block(b4).
block(b5).
block(b6).
block(b7).
#domain block(B).
#domain block(B1).
#domain block(B2).
```

[3]We could have made our time steps part of our fluent and action predicates and just said $on(B, L, I)$ and $put(B, L, I)$ instead of using *holds* and *occurs*. However, reifying actions and fluents allows us to introduce rules involving these concepts themselves, rather than their specific instances. We will see examples of such rules later in the chapter.

[4]Lparse syntax used with `clingo` allows us to use variable declarations so that we can avoid repetitive grounding in the bodies of rules. We use the *#domain* statement to associate a variable with its type. If using DLV, do not include these statements. Instead, ground all variables in the bodies of rules as seen in previous examples.

```
%% A location can be a block or the table.
location(X) :- block(X).
location(t).
#domain location(L).
#domain location(L1).
#domain location(L2).

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

%% "Block B is on location L" is a property that changes with time.
fluent(on(B,L)).
#domain fluent(F).

%% "Put block B on location L" is a possible action in our domain
%% provided we don't try to put a block onto itself.
action(put(B,L)) :- B != L.
#domain action(A).
```

Although we do not need variables for fluents and actions yet, we add them for consistency. Eventually, we will want our program to reason about fluents and actions in general.

To illustrate the behavior of our program, we will fix the initial configuration to the one shown in Figure 8.1. We can describe it by specifying the locations of the blocks at step 0, and including the closed world assumption for the *holds* relation for the initial situation:

```
%% holds(on(B,L),I): a block B is on location L at step I.
holds(on(b0,t),0).
holds(on(b3,b0),0).
holds(on(b2,b3),0).
holds(on(b1,t),0).
holds(on(b4,b1),0).
holds(on(b5,t),0).
holds(on(b6,b5),0).
holds(on(b7,b6),0).

%% If block B is not known to be on location L at step 0,
%% then we assume it is not.
```

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

Note that the above logic program completely defines the initial configuration of the blocks; i.e., we know the values of all the fluents of the domain at step 0 of our trajectory.

Now let's define the theory of the blocks world. To do this, we describe the effects of its actions. Since each action takes one step, rule 1

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

describes an effect of action $put(B, L)$. It states that putting block B on location L at step I causes B to be on L at step I+1. We assume for now that the robot never drops a block and is otherwise perfect in its execution of actions.

Rule 1 can be viewed as a special case of a **causal law** — a statement of the form

$$a \textbf{ causes } f \textbf{ if } p$$

which says that action a executed in a state of the domain satisfying conditions p causes fluent f to become true in the resulting state. Such general systems will be discussed in the next section.

The new location of block B can be viewed as a "direct" effect of action put. There are also "indirect" effects caused by relationships between fluents. For example, performing action $put(b2, t)$ in the initial situation has the direct effect of placing $b2$ on the table, and the indirect effect of $b2$ being removed from $b3$. This type of indirect conclusions can often be obtained from direct ones by using relations between fluents. In the case above it is sufficient to know that a block occupies a single location. This can be expressed by the following rule.

```
-holds(on(B,L2),I) :- holds(on(B,L1),I),    %% rule 2
                      L1 != L2.
```

Another useful rule says that no block can support more than one block directly on top of it

```
-holds(on(B2,B),I) :- holds(on(B1,B),I),    %% rule 3
                      B1 != B2.
```

These rules can be viewed as special cases of **state constraints** — statements of the form

$$f \textbf{ if } p$$

which say that every state satisfying p must also satisfy f. As usual, we test the success of our representation by inputting it into STUDENT and checking what it "knows." Of course, we gave a complete initial situation, so it knows which blocks are where at step 0. Now let's expand our program by a new statement:

```
occurs(put(b2,t),0).
```

This allows us to ask the program questions about what is true in step 1, and it should be able to figure out this information on its own, based on the laws we've encoded. Try giving STUDENT query $holds(on(b2, t), 1)$. It should answer *yes*. Now try query $holds(on(b0, t), 1)$. What do you expect? What does STUDENT answer?

A person would assume that $b0$ is still on the table because it was not moved, but STUDENT will answer "maybe." Is STUDENT not smart enough, or does it not have enough information? In most cases, unless they are told otherwise, humans live with an operative assumption that things normally stay as they are. Picking up objects does not normally affect properties of every other thing around us. We go on with life feeling pretty safe that bunnies are still furry and the ocean is still big. It seems reasonable to assume that moving a single block does not cause other blocks to move so much that their locations change. At least for purposes of this example, we had assumed that our robot is good at manipulating blocks, so we would like STUDENT to believe that the only change that occurred is that $b2$ is now on the table and no longer on $b3$. We must teach it that, lacking evidence to the contrary, it should assume that normally things stay as they are. This principle is known as the **Inertia Axiom** and can be expressed by adding rules 4a and 4b.

```
holds(F,I+1) :- holds(F,I),              %% rule 4a
                not -holds(F,I+1),
                I < n.

-holds(F,I+1) :- -holds(F,I),            %% rule 4b
                 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. This is a typical representation of defaults as in Chapter 5. Note that *not* $ab(d(F, I))$ is not included, which simplifies the program. Since our states are complete and the rule has no

weak exceptions, the omission is justified. (Recall from Section 5.1.1 that a strong exception refutes the default's conclusion and a weak one renders the default inapplicable.)

Rules 1–4 give a complete description of a configuration σ_1 which results from executing action $put(B, L)$. Add the new rules to your program and ask STUDENT some questions about its new configuration. Its answers should be more intuitive now. Later we study how to define "successor" states in more complex dynamic domains.

Is our description of the blocks world complete? To answer this question, change the *occurs* statement of the program to contain an action that should not be executable:

```
occurs(put(b6,t),0).
```

Since $b6$ is occupied, this action should not be allowed. Ask STUDENT to describe the next state by asking it for all answers to query $holds(on(B, L), 1)$. You will see that the program derives that block $b6$ is on the table at step 1 and that $b7$ is still on top of it. This answer does not match our intuition unless we believe the robot arm to be very coordinated. Once again, we see that STUDENT needs more information; namely, we need to tell it which actions are not allowed. The next two rules are constraints on the executability of actions.

```
-occurs(put(B,L),I) :- holds(on(B1,B),I).     %% rule 5a

-occurs(put(B1,B),I) :- holds(on(B2,B),I).    %% rule 5b
```

Rule 5a says that it is impossible to move a block that is occupied. Rule 5b says that it is impossible to move a block onto an occupied block. These rules are examples of **executability conditions** whose general form is

$$\textbf{impossible } a_1 \dots a_k \textbf{ if } p$$

Intuitively, the law states that it is impossible to execute actions $a_1 \dots a_k$ simultaneously in a state satisfying conditions p. Add these rules and ask STUDENT to describe again what is true at Step 1. It should answer that there are no models, which indicates that the suggested scenario is inconsistent with the rules of the program, as we would expect.

Rules 1–5 constitute a simple theory of the blocks world. Let's see how well STUDENT does if we add another step. To ask questions about the state of the world after block $b2$ is moved on the table and $b7$ is put on $b2$, we need to get rid of our bad *occurs* statement and write the following two statements instead:

```
occurs(put(b2,t),0).
occurs(put(b7,b2),1).
```

Let us call this program `blocks1.lp`. Querying STUDENT with $holds(on(B, L), 2)$ should give us everything that is true after the two actions are executed. Finally our representation mirrors what we expect the program to know, and it can answer our queries intelligently. For practice run STUDENT with `blocks1.lp`. Try changing the two *put* statements in the program to the non-executable statements $occurs(put(b2, b4), 0)$ and $occurs(put(b7, b4), 1)$. What is the result? Is rule 5b necessary for STUDENT to give the right answers?

So far our model of the blocks world contained only fluents formed by relation *on*. Let us now see if we can expand our model by introducing a new fluent, $above(B_2, B_1)$ — "block B_2 is located above block B_1". This can be done by the following recursive definition:

```
holds(above(B2,B1),I) :-
          holds(on(B2,B1),I).
holds(above(B2,B1),I) :-
          holds(on(B2,B),I),
          holds(above(B,B1),I).
-holds(above(B2,B1),I) :-                %%CWA
          not holds(above(B2,B1),I).
```

This is very similar to other recursive definitions we discussed in Chapter 4 such as the definition of ancestors in Section 4.1.3 or subclasses in Section 4.3. But definitions of fluents in dynamic domains are slightly more subtle. To see the problem, let us consider program `blocks2.lp` obtained by expanding `blocks1.lp` by adding the definition of *above*, and running it in conjunction with rules

```
occurs(put(b2,t),0).
:- -holds(above(b2,b0),1).
```

(Do not forget to add statement `fluent(above(B2,B1))` to your list of fluents.) Since intuitively $\neg holds(above(b2, b0), 1)$ should be true, the resulting program should be inconsistent. But this is not what will happen. You will see that the solver returns a model containing $holds(above(b2, b0), 1)$. The problem is caused by the unintended interplay between the Inertia Axiom and the CWA part of the definition of *above*. Informal reasoning which corresponds to this model goes as follows: b_2 is above b_0 at step 0. The Inertia

Axiom allows us to conclude that b_2 is still above b_0 even after b_2 is put on the table. CWA from the definition is blocked, the constraint is satisfied and the model is found. Of course, if in the attempt to construct a model we were to apply the CWA first, then the Inertia Axiom would be blocked, and we would not be able to satisfy the constraint. (It may be instructive to check that program `blocks2.lp` without the constraint has exactly two models outlined above.)

To remedy the problem, it is sufficient to notice that the new fluent, *above*, is uniquely defined by the values of fluent *on* and, therefore, should not be made subject to the Inertia Axiom. This observation suggests a division of fluents of our domain into two classes — **inertial** and **defined fluent**. Intuitively, an inertial fluent is subject to the law of inertia; its value can be (directly or indirectly) changed by an action. If no such action occurs, the value of the fluent remains unchanged. A defined fluent is not subject to the Inertia Axiom and cannot be directly caused by any action; instead it is defined in terms of other fluents.

To reflect the division of fluents into two types, let's create `blocks3.lp` by modifying `blocks2.lp` as follows:

1. Replace statement `fluent(on(B,L))` in program `blocks2.lp` by statement `fluent(inertial,on(B,L))`.

2. Change rules 4(a) and 4(b) (Inertia Axiom) to apply only to inertial fluents by adding condition `fluent(inertial,F)` to their bodies.

3. List *above* as a defined fluent in our list of fluents by stating `fluent(defined,above(B2,B1))`.

Now the notion of *above* is properly introduced. Run `blocks3.lp` and check that the results correspond to our intuition.

8.2 A General Solution

The knowledge base from the example in the previous section defines a dynamic system containing all possible trajectories of the blocks world. In our general theory of actions and change, a dynamic system will be modeled by a transition diagram whose nodes correspond to physically possible states of the domain and whose arcs are labeled by actions. Such models are called Markovian.

A transition $\langle \sigma_0, a, \sigma_1 \rangle$ of the diagram where $a = \{a_1, \ldots, a_k\}$ is a set of actions executable in state σ_0 indicates that σ_1 may be a result of simultaneous execution of these actions in σ_0. Our representation guarantees that

the effect of an action depends only on the state in which that action was executed. The way in which this state was reached is irrelevant.

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

In the dynamic system from the blocks-world example, states correspond to configurations of blocks satisfying the constraints from rules 2 and 3. Actions are of the form $put(Block, Location)$. Program rules 1–5 define the corresponding state transitions.

A system may often have a large and complex diagram. The problem of finding its concise and mathematically accurate description is not trivial and has been a subject of research for over 30 years. Its solution requires a good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents. An additional level of complexity is added by the need to specify what is not changed by actions. As noticed by John McCarthy, the latter, known as the **Frame Problem**, can be reduced to finding a representation of the Inertia Axiom, which states that "Things normally stay as they are." In our blocks-world example, we represented this axiom by logic programming rules 4a and 4b. Notice that default theory was instrumental in solving this problem.

As we have seen, causal effects of actions can be defined by causal laws of the form:

$$a \textbf{ causes } f \textbf{ if } p$$

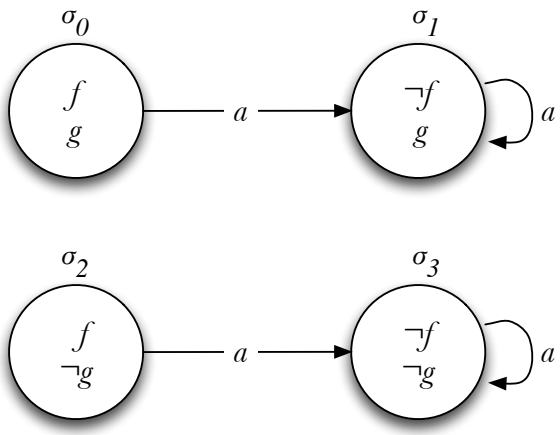
which says that action a, executed in a state satisfying conditions p, causes fluent f to become true in the resulting state. We saw the use of such laws in our blocks-world example when we defined the effect of action $put(Block, Location)$.

Consider another, simpler example of a dynamic system whose states are described by two Boolean inertial fluents, f and g, and arcs are labeled by one action a whose effect is described by a single causal law a **causes** $\neg f$ **if** f. We will denote this description of our system by $\mathcal{D}_0$. Common sense suggests that if a is executed in $\sigma_0 = \{f, g\}$, the new, successor state will contain $\neg f$ implied by the causal law. Note also that g will remain true by the Inertia Axiom. Figure 8.3 shows the transition diagram for $\mathcal{D}_0$.

Causal and other relations between fluents can be described by state constraints — statements of the form

$$f \textbf{ if } p$$

which say that every state satisfying conditions p must also satisfy f. They



$\mathcal{F} = \{f, g\}$ $\mathcal{A} = \{a\}$
a **causes** $\neg f$ **if** f

Figure 8.3: Transition Diagram of System $\mathcal{D}_0$

are used to define indirect effects of actions. Finding concise ways for defining these effects is called the **Ramification Problem**. Together with the Frame Problem discussed above, the Ramification Problem caused substantial difficulties for researchers in their attempts to precisely define transitions of discrete dynamic systems.

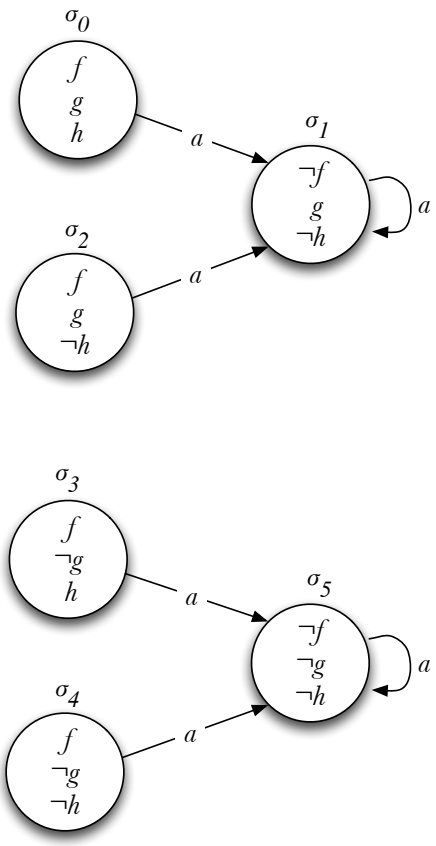
As we have seen, blocks-world rules 2 and 3 are examples of state constraints. They dictate the value of certain fluents in a state based on the values of other fluents in the state. Note that they are not dependent on actions.

To illustrate this let us expand description $\mathcal{D}_0$ by adding an inertial fluent h and state constraint $\neg h$ **if** $\neg f$. Figure 8.4 shows the transition diagram of $\mathcal{D}_1$. State σ_1 contains $\neg f$ by the causal law, g by inertia, and $\neg h$ by the new state constraint which is responsible for the indirect effect of $\neg f$ becoming true in σ_1.

Now let's create $\mathcal{D}_2$ by expanding $\mathcal{D}_1$ by executability condition

impossible a **if** $\neg f$

which says that it is impossible to perform action a in any state which contains fluent $\neg f$. Its transition diagram is shown in Figure 8.5. The diagram differs from Figure 8.4 in that it has fewer transitions; namely, the



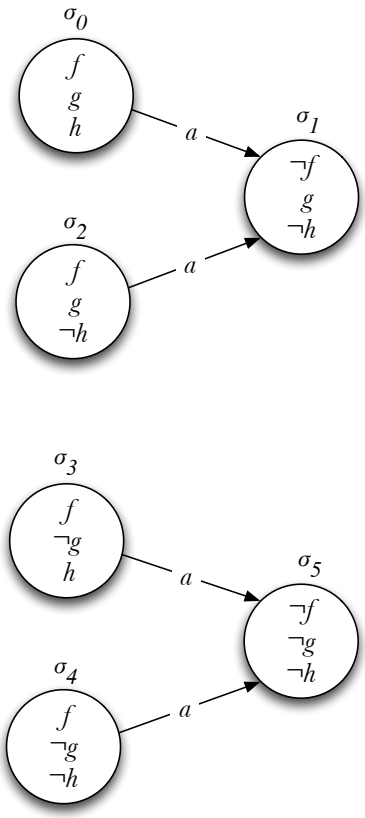
$\mathcal{F} = \{f, g, h\}$ $\mathcal{A} = \{a\}$

a **causes** $\neg f$ **if** f

$\neg h$ **if** $\neg f$

Figure 8.4: Transition Diagram of System $\mathcal{D}_1$

cycles in σ_1 and σ_5 have been eliminated.



$\mathcal{F} = \{f, g, h\}$ $\mathcal{A} = \{a\}$
a **causes** $\neg f$ **if** f
$\neg h$ **if** $\neg f$
impossible a **if** $\neg f$

Figure 8.5: Transition Diagram of System $\mathcal{D}_2$

Description $\mathcal{D}_0$, $\mathcal{D}_1$ 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 behavior of dynamic systems. Another way to look at them is as tools for describing transition diagrams. We have seen a brief introduction to the syntax of these languages in our examples. The semantics was given by our intuitive understanding of what these laws might mean. Now we will give a formal, mathematical definition of the syntax and

semantics of $\mathcal{AL}$.

8.3 $\mathcal{AL}$ Syntax

Let us begin with some basic terminology. Action language $\mathcal{AL}$ is parametrized by a sorted signature containing three special sorts: *statics*, *fluents* and *actions*. The fluents are partitioned into two sorts: *inertial* and *defined*. We will 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; S is called **consistent** if there is no p such that $p \in S$ and $\neg p \in S$.

Definition 8.3.1. *(Statements of $\mathcal{AL}$)*
Language $\mathcal{AL}$ allows the following types of statements:

1. *Causal Laws:*

$$a \textbf{ causes } l_i \textbf{ if } p$$

2. *State Constraints:*

$$l \textbf{ if } p$$

3. *Executability Conditions:*

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p$$

where a is an action, l is an arbitrary domain literal, l_i is a literal formed by an inertial fluent, p is a set of domain literals, and $k \geq 0$. Moreover, no negation of a defined fluent can occur in the heads of state constraints.

The collection of state constraints whose head is a defined fluent f is referred to as the *definition* of f. As in logic programming definitions, f is true if it follows from the truth of the body of at least one of its defining rules. Otherwise, f is false.

Definition 8.3.2. *(System Description)*
A **system description** *of $\mathcal{AL}$ is a collection of statements of $\mathcal{AL}$.*

8.4 $\mathcal{AL}$ Semantics — The Transition Relation

A system description $\mathcal{SD}$ serves as a specification of the transition diagram $\mathcal{T}(\mathcal{SD})$ defining 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.

8.4.1 States

We start with the states. If $\mathcal{SD}$ does not contain defined fluents, the definition of a state is simple — a state is simply a complete and consistent set of domain literals satisfying state constraints of $\mathcal{SD}$. This definition was used in Examples $\mathcal{D}_0$, $\mathcal{D}_1$ and $\mathcal{D}_2$ above. For system descriptions with defined fluents the situation is more subtle. The following simple example illustrates the problem.

Example 8.4.1. *(State)*
Let us consider a system description D_3 with two inertial fluents, f and g and a fluent h defined by the following rules:

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

Clearly, $\{f, g, h\}$ is a state of D_3 and $\{f, g, \neg h\}$ is not. But what about $\{\neg f, g, h\}$? It does not seem to satisfy the definition of h since the truth of h does not follow from any of its defining rules. So the intended answer to our question seems to be *no*. But $\{\neg f, g, h\}$ satisfies the constraints! The suggested definition does not work. It may be tempting to consider an additional condition requiring a defined fluent to be true *iff* at least one of its defining rules is satisfied. This would give the correct answer to our example, but would not work if we were to expand the definition by an extra rule:

$$h \textbf{ if } h$$

The tautological rule should not change the states of D_3, but it does. According to the suggested modification, $\{\neg f, g, h\}$ is a state. An attentive reader will notice the similarity between this situation and the use of Clark's completion for defining semantics of logic programs. This similarity leads to *the idea of defining states via logic programming under answer set semantics*, as shown below.

We will need the following notation. 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:

$$\neg f \leftarrow \ not\ f.$$

For any set σ of domain literals let σ_{nd} denote the collection of all literals of σ formed by inertial fluents and statics. (The $_{nd}$ stands for non-defined.)

Definition 8.4.1. *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 the 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 non-defined literals from σ, the encoding of the state constraints, and the CWA for each defined fluent. *Note that (a) Every state of system description $\mathcal{SD}$ satisfies the state constraints of $\mathcal{SD}$ and (b) If the signature of $\mathcal{SD}$ does not contain defined fluents, a state is simply a complete, consistent set of literals satisfying the state constraints of $\mathcal{SD}$.*

Example 8.4.2. *(Example 8.4.1 Revisited)*
Let us consider a system description D_3 from Example 8.4.1. Program $\Pi_c(\mathcal{SD})$ consists of rules

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

It is easy to check that the transition diagram defined by D_3 has the following states: $\{f, g, h\}$, $\{f, \neg g, h\}$, $\{\neg f, \neg g, \neg h\}$, $\{\neg f, g, \neg h\}$. To check that $\sigma_0 = \{f, \neg g, h\}$ is a state, it is enough to check that σ_0 is the only answer set of $\Pi_c(\mathcal{SD}) \cup \{f, g\}$. Similarly for other states. To see that $\sigma = \{\neg f, g, h\}$ is not a state, it suffices to see that σ is not the answer set of $\Pi_c(\mathcal{SD}) \cup \{\neg f, g\}$.

The next example explains the importance of the uniqueness requirement of the definition.

Example 8.4.3. *(Mutually Recursive Laws)*
Let us consider a system description D_4 with two defined fluents f and g which are defined by the following mutually recursive laws:

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

Let us check if $\{f, \neg g\}$ is a state of $\mathcal{T}(D_4)$. Program $\Pi_c(D_4)$ from Definition 8.4.1 consists of rules

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

Since all the fluents of D_4 are defined, $\sigma_{nd} = \emptyset$ and program $\Pi_c(D_4) \cup \sigma_{nd}$ has two answer sets, $\{f, \neg g\}$ and $\{g, \neg f\}$. This violates the uniqueness condition of Definition 8.4.1 and hence $\mathcal{T}(D_4)$ has no states. This is intended. Mutually recursive laws of D_4 are not strong enough to uniquely define f and g; thus, the definition is rejected.

We conclude our definition of state by giving a sufficient condition which guarantees that defined fluents of a system description are uniquely defined by the system's statics and inertial fluents. To mathematically capture this property we use the following definition:

Definition 8.4.2. *(Well-Founded System Description)*
A system description $\mathcal{SD}$ of $\mathcal{AL}$ is called **well-founded** *if for any complete and consistent set of fluent literals σ satisfying the state constraints of $\mathcal{SD}$, the program*

$$\Pi_c(\mathcal{SD}) \cup \sigma_{nd} \tag{8.1}$$

has at most one answer set.

To give the sufficient condition guaranteeing well-foundedness of $\mathcal{SD}$ we need the following notions:

Definition 8.4.3. *(Fluent Dependency Graph)*
The **fluent dependency graph** *of a system description $\mathcal{SD}$ is the directed graph such that*

- *its vertices are arbitrary literals,*
- *it has an edge*

- *from l to l' if l is formed by a static or an inertial fluent and $\mathcal{SD}$ contains a state constraint with the head l and the body containing l',*
- *from f to l' if f is a defined fluent and $\mathcal{SD}$ contains a state constraint with the head f and the body containing l' and not containing f.*
- *from $\neg f$ to f for every defined fluent f.*

The fluent dependency graphs for Examples 8.4.2 and 8.4.3 are given in Figures 8.6 and 8.7, respectively. Recall that h is the defined fluent in the first example, while f and g are the defined fluents in the second.

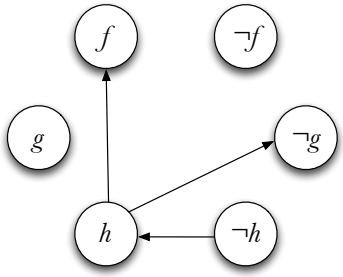


Figure 8.6: Fluent Dependency Graph for Example 8.4.2

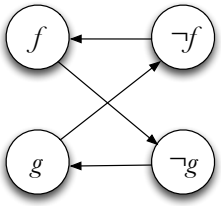


Figure 8.7: Fluent Dependency Graph for Example 8.4.3

Definition 8.4.4. *(Weak Acyclicity)*
A fluent dependency graph is **weakly acyclic** *if it does not contain paths from defined fluents to their negations. By extension, a system description with a weakly acyclic fluent dependency graph is also called* weakly acyclic*.*

Consequently, as expected, the graph in Figure 8.6 is weakly acyclic, while the one in Figure 8.7 is not.

Proposition 8.4.1. *(Sufficient Condition for Well-Foundedness)*
If a system description $\mathcal{SD}$ of $\mathcal{AL}$ is weakly acyclic then $\mathcal{SD}$ is well-founded.

8.4.2 Transitions

Our *definition of transition relation of $\mathcal{T}(\mathcal{SD})$ is also based on the notion of answer set 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 system description $\mathcal{SD}$, initial state σ_0, and set of actions a, such that answer sets of this program determine the states the system can move into after the execution of a in σ_0.

Definition 8.4.5. *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 $sig(\mathcal{SD})$ of the signature of $\mathcal{SD}$.

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

    $$sort\_name(c) \tag{8.2}$$

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

    $$static(g) \tag{8.3}$$

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

    $$fluent(inertial, f) \tag{8.4}$$

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

    $$fluent(defined, f) \tag{8.5}$$

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

    $$action(a) \tag{8.6}$$

- **Encoding of Statements of $\mathcal{SD}$**

  For this encoding we only need two steps, 0 *and* 1*, which stand for the beginning and the end of a transition. This is sufficient for describing a single transition; however, later, we will want to describe a longer chain of events and let steps range over* $[0, n]$ *for some constant* n*. To allow an easier generalization of the program we encode steps by using constant* n *for the maximum number of steps, as follows:*

  $$\#const\ n = 1. \tag{8.7}$$

  $$step(0..n). \tag{8.8}$$

  As in our blocks-world example, we introduce a relation $holds(f, i)$ *which says that fluent* f *is true at step* i*. To simplify the description of the encoding, we also introduce a new notation,* $h(l, i)$ *where* l *is a domain literal and* i *is a step. If* f *is a fluent then by* $h(l, i)$ *we denote* $holds(f, i)$ *if* $l = f$ *or* $\neg holds(f, i)$ *if* $l = \neg f$*. If* l *is a static literal then* $h(l, i)$ *is simply* l*. If* p *is a set of domain literals, then* $h(p, i) =_{def} \{h(l, i) : l \in p\}$*. We also need relation* $occurs(a, i)$ *which says that action* a *occurred at step* i*;* $occurs(\{a_0, \ldots, a_k\}, i) =_{def} \{occurs(a_i) : 0 \leq i \leq k\}$*.*

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

  – *For every causal law*

  $$a \textbf{ causes } l \textbf{ if } p$$

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

  $$\begin{array}{rcl} h(l, I+1) & \leftarrow & h(p, I), \\ & & occurs(a, I), \\ & & I < n. \end{array} \tag{8.9}$$

  – *For every state constraint*

  $$l \textbf{ if } p$$

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

  $$h(l, I) \leftarrow h(p, I). \tag{8.10}$$

  – $\Pi(\mathcal{SD})$ *contains the CWA for defined fluents:*

  $$\begin{array}{rcl} \neg holds(F, I) & \leftarrow & fluent(defined, F), \\ & & not\ holds(F, I). \end{array} \tag{8.11}$$

– *For every executability condition*

$$\textbf{impossible } a_0, \ldots, a_k \textbf{ if } p$$

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

$$\neg occurs(a_0, i) \textit{ or } \ldots \textit{ or } \neg occurs(a_k, i) \quad \leftarrow \quad h(p, i). \tag{8.12}$$

– $\Pi(\mathcal{SD})$ *contains the Inertia Axiom:*

$$\begin{array}{lll} holds(F, I+1) & \leftarrow & fluent(inertial, F), \\ & & holds(F, I), \\ & & not\ \neg holds(F, I+1), \\ & & I < n. \end{array} \tag{8.13}$$

$$\begin{array}{lll} \neg holds(F, I+1) & \leftarrow & fluent(inertial, F), \\ & & \neg holds(F, I), \\ & & not\ holds(F, I+1), \\ & & I < n. \end{array} \tag{8.14}$$

– $\Pi(\mathcal{SD})$ *contains CWA for actions:*

$$\neg occurs(A, I) \leftarrow \ not\ occurs(A, I). \tag{8.15}$$

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

To continue with our definition of transition $\langle \sigma_0, a, \sigma_1 \rangle$ we describe the two remaining parts of program $\Pi(\mathcal{SD}, \sigma_0, a)$ — the encoding $h(\sigma_0, 0)$ of initial state σ_0 and the encoding $occurs(a, 0)$ of action a:

$$h(\sigma_0, 0) =_{def} \{h(l, 0) : l \in \sigma_0\}$$

and

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

To complete program $\Pi(\mathcal{SD}, \sigma_0, a)$ we simply gather our description of the system's laws, together with the description of the initial state and the actions that occur in it:

Definition 8.4.6.

$$\Pi(\mathcal{SD}, \sigma_0, a) =_{def} \Pi(\mathcal{SD}) \cup h(\sigma_0, 0) \cup occurs(a, 0).$$

Now we are ready to define the notion of transition defined by a system description $\mathcal{SD}$ of $\mathcal{AL}$.

Definition 8.4.7. *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 description $\mathcal{SD}$. A state-action-state triple $\langle \sigma_0, a, \sigma_1 \rangle$ is a* **transition** *of $\mathcal{T}(\mathcal{SD})$ iff $\Pi(\mathcal{SD}, \sigma_0, a)$ has an answer set A such that $\sigma_1 = \{l : h(l, 1) \in A\}$.*

We now have a program that, like a rational human reasoner, can predict what the state of the world will be once an action is performed in a given state. In the next section, we give examples of specific domains that can be described by $\mathcal{AL}$ and how they can be translated into ASP programs using Definition 8.4.6.

8.5 Examples

As we have seen, in order to model a dynamic domain, we need to describe what actions cause what effects under what conditions. To do this, we need to identify:

1. the objects, properties, and actions of the domain;

2. the relationships between the properties;

3. the executability conditions and causal effects of actions.

In other words, we must come up with an $\mathcal{AL}$ system description for our domain. In this section we give several examples of such descriptions and walk through the steps of constructing the corresponding transition diagrams.

8.5.1 The Briefcase Domain

Consider a briefcase with two clasps. We have an action, toggle, which moves a given clasp into the up position if the clasp is down, and vice versa. If both clasps are in the up position, the briefcase is open; otherwise, it is closed. Create a (simple) model of this domain.

The signature of the briefcase domain consists of sort $clasp = \{1, 2\}$, inertial fluent $up(C)$ which holds iff the clasp C is up, defined fluent $open$ which holds iff both clasps are up, and action $toggle(C)$ which toggles clasp C.

The system description, $\mathcal{D}_{bc}$, of our domain will consist of axioms

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

where C ranges over the sort *clasp*. Since these laws contain variables, they are not, strictly speaking, proper statements of our action language. Individual laws can be obtained by grounding the variables. Since our signature is sorted and variable C ranges over clasps, the grounding will respect this sorting information and replace C by 1 and 2. Laws with variables are often referred to as **schemas**. The first schema above can be viewed as a shorthand for two laws:

$$\begin{array}{l} toggle(1) \ \textbf{causes} \ up(1) \ \textbf{if} \ \neg up(1) \\ toggle(2) \ \textbf{causes} \ up(2) \ \textbf{if} \ \neg up(2) \end{array}$$

Now let us figure out what states of our domain look like. According to Definition 8.4.1 we need a program $\Pi_c(\mathcal{D}_{bc})$ that consists of the following two rules:

$$open \leftarrow up(1), up(2) \tag{1}$$

$$\neg open \leftarrow \ not \ open \tag{2}$$

Consider a collection

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

Is this a state? First we need to check if it is complete and consistent. By definition, it is. Next we need to consider

$$\sigma_{nd} = \{\neg up(1), up(2)\}$$

and check if σ is the only answer set of the program $\Pi_c(\mathcal{D}_{bc}) \cup \sigma_{nd}$ consisting of rules (1), (2) above, and facts $\neg up(1)$ and $up(2)$. Clearly, it is and hence, as expected, σ is a state of our transition diagram.

Now let σ be $\{\neg up(1), up(2), open\}$. According to the description of the briefcase, this is a physical impossibility and, hence, should not be a state. Indeed this is the case according to our definition. Although it is complete and consistent, it is not the answer set of the program consisting of rules (1) and (2) and facts $\neg up(1)$ and $up(2)$. Therefore, σ is not a state.

Now let us construct program $\Pi(\mathcal{D}_{bc})$ which is needed to define transitions of our system. The program will be written in the syntax of Lparse

and named `bc.lp`. In its construction we slightly deviate from the standard encoding from Definition 8.4.5 in order to avoid long listing of fluents, actions, and other sorts. Instead we define sorts using logic programming rules with variables. Definitions of fluents and actions in the program below can serve as an example of this technique. Of course the use of rules is just a matter of convenience. They can be eliminated and replaced by collections of atoms. Here is the translation of our system description $\mathcal{D}_{bc}$ into logic program:

```
%% Domain Signature
clasp(1).
clasp(2).
#domain clasp(C).

fluent(inertial, up(C)).
fluent(defined, open).
action(toggle(C)).
```

Clearly the definition of inertial fluent which is simply a shorthand for a rule

```
fluent(inertial, up(C)) :- clasp(C).
```

can be replaced by two atoms

```
fluent(inertial, up(1)).
fluent(inertial, up(2)).
```

Similarly for actions. This competes our definition of the sort *clasp* and fluents and actions of the domain.

Next, we translate our axioms. The first two axioms of our system description are causal laws. Using rule (8.9) from Definition 8.4.5 we get

```
#const n = 1.
step(0..n).
#domain step(I).

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

%% 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 rule is a state constraint. Using rule (8.10) we get

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

We add the closed world assumption for defined fluents as dictated by rule (8.11) as follows:

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

(Notice that, in accordance with the general translation, this rule encodes CWA for all defined fluents. Currently, we only have one such fluent, but we might wish to add more. CWA for defined fluents is taken care of once and for all.)

Then we add the Inertia Axiom as defined by rules (8.13) and (8.14), allowing us to derive values of inertial fluents, even is no action explicitly dictated their change.

```
%% General 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.
```

Last we add the closed world assumption for actions as given by rule (8.15):

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

This concludes our encoding of the briefcase domain. Figure 8.8 shows the transition diagram for this system.

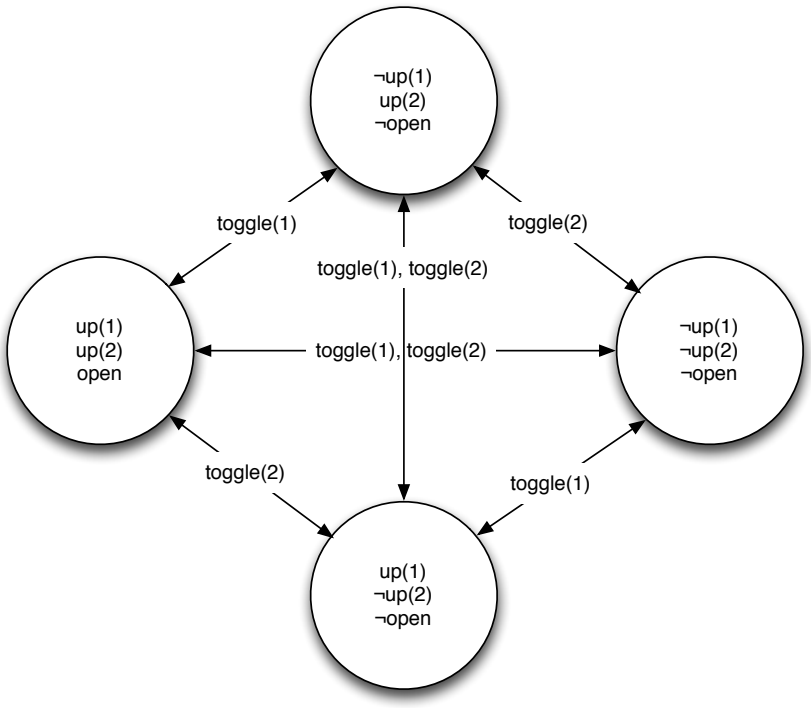


Figure 8.8: The Two-Clasp Briefcase Domain

Using this program one can check, for instance, that the system contains transitions

$\langle\{\neg up(1), up(2), \neg open\}, toggle(1), \{up(1), up(2), open\}\rangle$,
$\langle\{up(1), up(2), open\}, toggle(1), \{\neg up(1), up(2), \neg open\}\rangle$,
$\langle\{\neg up(1), \neg up(2), \neg open\}, toggle(1), toggle(2), \{up(1), up(2), open\}\rangle$,
etc.

Simply add the appropriate information about the initial state and the action that occurred in it as in Definition 8.4.7. For example, to check the first transition, add statements:

```
%% Initial Situation
-holds(up(1),0).
holds(up(2),0).
-holds(open,0).
%% Action
occurs(toggle(1),0).
```

```
#hide.
#show holds(A,B), -holds(A,B).
```

and invoke the program to get the values of the fluents. The transition with $toggle(1)$ and $toggle(2)$ occurring simultaneously can be tested in the same manner by using two actions, $occurs(toggle(1), 0)$ and $occurs(toggle(2), 0)$.

Note that encoding of our initial state can be simplified by omitting `-holds(open,0).` This is true because *open* is a defined fluent and its value will be computed by the corresponding defaults of $\Pi(\mathcal{D}_{bc})$. This property, of course, holds in general and from now on we will not include the encoding of the values of defined fluents in the encoding of a state.

8.5.2 The Blocks World Revisited

Now let's go back to our basic blocks world example from Section 8.1, with two fluents *on* and *above*, and construct a system description $\mathcal{D}_{bw}$ in $\mathcal{AL}$ representing the blocks-world transition diagram.

The signature of $\mathcal{D}_{bw}$ consists of sorts $block = \{b_0 \dots b_7\}$ and $location = \{t\} \cup block$, inertial fluent $on(block, location)$, defined fluent $above(block, block)$ and action $put(block, location)$. We assume that $put(B_1, B_2)$ is an action only if $B_1 \neq B_2$.

The laws of the blocks world are

1. $put(B, L)$ **causes** $on(B, L)$
2. $\neg on(B, L_2)$ **if** $on(B, L_1), L_1 \neq L_2$
3. $\neg on(B_2, B)$ **if** $on(B_1, B), B_1 \neq B_2$
4. $above(B_2, B_1)$ **if** $on(B_2, B_1)$
5. $above(B_2, B_1)$ **if** $on(B_2, B), above(B, B_1)$
6. **impossible** $put(B, L)$ **if** $on(B_1, B)$
7. **impossible** $put(B_1, B)$ **if** $on(B_2, B)$

where (possibly indexed) Bs and Ls stand for blocks and locations, respectively.

Now let us translate $\mathcal{D}_{bw}$ into the corresponding logic program `bw.lp`. The collection of blocks will be represented by

```
block(b0).
block(b1).
block(b2).
block(b3).
block(b4).
block(b5).
block(b6).
block(b7).
#domain block(B).
#domain block(B1).
#domain block(B2).
```

Instead of listing the collection of facts defining sort *location*, we save ourselves some typing and define locations using logic programming rules

```
location(X) :- block(X).
location(t).
#domain location(L).
#domain location(L1).
#domain location(L2).
```

We also list fluents, actions, and steps:

```
fluent(inertial, on(B,L)).
fluent(defined, above(B2,B1)).
action(put(B,L)) :- B != L.

#const n = 1.
step(0..n).
#domain step(I).
```

Now let's use the standard encoding from Definition 8.4.5 to encode the blocks-world laws. Law 1 is a causal law with fluent $on(B, L)$ and action $put(B, L)$. Following the directions for causal laws, we use encoding (8.9):

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

Laws 2–5 are typical state constraints and we translate them using (8.10) as follows:

```
-holds(on(B,L2),I) :- holds(on(B,L1),I), L1 != L2.
```

```
-holds(on(B2,B),I) :- holds(on(B1,B),I), B1 != B2.

holds(above(B2,B1),I) :- holds(on(B2,B1),I).

holds(above(B2,B1),I) :- holds(on(B2,B),I),
                         holds(above(B,B1),I).
```

Using (8.12), executability conditions from Laws 6 and 7 become

```
-occurs(put(B,L),I) :- holds(on(B1,B),I).

-occurs(put(B1,B),I) :- holds(on(B2,B),I).
```

Last, we add the rules needed for all domains: the CWA for defined fluents (8.11), the Inertia Axiom (8.13 and 8.14), and the CWA for actions (8.15).

```
%% CWA for Defined Fluents

-holds(F,I) :- fluent(defined,F),
               not holds(F,I).

%% General 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) :- not occurs(A,I).
```

This concludes the encoding `bw.lp` of system description $\mathcal{D}_{bw}$. The resulting program defines the transition diagram of the dynamic system associated with the blocks world.

You might have noticed that our translation algorithm produces the same encoding as that in `blocks3.lp`. This is of course not an accident —

insights obtained by researchers representing the blocks world and similar problems in ASP lead them to the development of $\mathcal{AL}$ and its corresponding translation given in Definition 8.4.5. So why use $\mathcal{AL}$? Notice that the description of the blocks world in $\mathcal{AL}$ is substantially shorter than that in ASP. The former does not explicitly mention steps and contains neither inertia axioms nor defaults. The statements of $\mathcal{AL}$ have rather clear intuitive meaning and can be used by knowledge engineers who do not have a clear notion of default negation and other non-trivial ASP concepts. All this is hidden in the translation which can easily be automated. This of course should not come as a surprise to computer scientists who are by now well familiar with the idea of high-level languages and language translators.

8.5.3 Blocks World with Concurrent Actions

So far we have dealt with a one-arm blocks-world domain. Suppose now that we have two robotic arms capable of avoiding collisions in the air. Now that two blocks can be moved simultaneously, we must make sure that our robot can behave sensibly. Clearly the new system description $\mathcal{D}_{tbw}$ must contain all the statements of $\mathcal{D}_{bw}$, but we may also need some additional executability conditions. Let's consider what problems our new system may run into. For example, we must consider what would happen if both arms were to try to move the same block to two separate locations at once; e.g., we should not allow $occurs(put(b2, t), 0)$ and $occurs(put(b2, b4), 0)$, even though each individual action is perfectly legal. It can be easily seen, however, that this is already guaranteed by the laws of $\mathcal{D}_{bw}$. If the two actions were executed in parallel then by causal law (1), $b2$ will be located on both $b4$ and the table. This would contradict constraint (2), and hence such a parallel execution is already prohibited. Indeed, if we combine `bw.lp` with the initial situation described in Figure 8.1 and actions $occurs(put(b2, t), 0)$ and $occurs(put(b2, b4), 0)$, the program will be inconsistent. So far so good. Now suppose we instead want to simultaneously execute actions $put(b2, t)$ and $put(b7, b2)$. Replacing the previous actions by these would produce a program with the answer set containing $b2$ on the table and $b7$ on $b2$. Is this the answer you expected? If you assumed that our robot arms were coordinated in such a way as to be able to stack blocks while moving them and then put them down, then you answered "yes." But if you, like us, are not as confident in the state-of-the-art of robotics and considered the concurrent execution of these two actions to be physically impossible, then the program's answer is wrong. However, this is to be expected since nothing in $\mathcal{D}_{bw}$ prevents this from happening. To avoid the problem, let's teach the

system that actions which put B_1 on B_2 and simultaneously move B_2 are impossible:

$$\textbf{impossible } put(B_1, L), put(B_2, B_1) \textbf{ if}$$
$$action(put(B_1, L)), action(put(B_2, B_1))$$

Translating into ASP we get

```
-occurs(put(B1,L),I) | -occurs(put(B2,B1),I) :-
                                        action(put(B1,L)),
                                        action(put(B2,B1).
```

Note that this impossibility condition differs from the previous ones in that the requirement for $put(B_1, L)$ and $put(B_2, B_1)$ being actions is explicitly stated in the premise of the rule. Previously this was implicit. The change is needed only if we want to run the corresponding program using `clingo` with option `--shift` as follows:

```
clingo 0 --shift twoarms.lp
```

By explicitly requiring $put(B_1, L)$ and $put(B_2, B_1)$ to be actions[5], we avoid the application of this transformation to

```
-occurs(put(B,B),I) | -occurs(put(B,B),I)
```

which could cause unintended inconsistency of the program. The problem, of course, does not exist for systems like `claspD` and DLV, which implement real disjunction.

To test the new system description let's add this statement to `bw.lp`, denote the resulting program by `twoarms.lp` and compute the answer sets of `twoarms.lp` combined with the usual initial situation and statements $occurs(put(b2, t), 0)$ and $occurs(put(b7, b2), 0)$. To accommodate the new, disjunctive rule, use `claspD` or invoke `clingo` with option `--shift`. The program should say that there are no models.

If instead we use the program together with concurrent actions

$$occurs(put(b2, t), 0), occurs(put(b4, b7), 0)$$

we will obtain the expected results. But what would happen if we were to attempt to simultaneously move $b4$ to $b7$ and $b2$ to $b1$? Physically this of course also depends on the degree of coordination of the robot's arms, but this degree of coordination seems reasonable. Our program, however,

[5]Recall that we have defined $put(B_1, B_2)$ to be an action only if $B_1 \neq B_2$.

will not believe so. Not surprisingly it will tell us that such a move is impossible. This happens because of constraint 7 which does not allow us to stack a block on top of a block occupied at the beginning of the action. Can we allow such a move? Fortunately, the answer is yes. It is simply sufficient to remove constraint (7) from $\mathcal{D}_{bw}$. Of course we need to make sure that this would not allow some impossible moves. After all, the constraint was added for this purpose. Some thought will show that no impossible moves will be allowed after all. Indeed this constraint was unnecessary from the beginning. Moving blocks B_1 and B_2 onto block B would cause two blocks to be located on the same block which is prohibited by law (3) of $\mathcal{D}_{bw}$.

What happened is a typical and rather frequently occurring problem called **over-specification**. The condition, which was simply unnecessary in the original situation, caused an unexpected problem when the domain was expanded. In other words over-specification can decrease the degree of elaboration tolerance of the program.

Experience in many areas of Computer Science shows that programmers must be careful to avoid unwanted side effects when allowing concurrent actions, but the payoff may be substantial. The discussion in this section shows that our approach incorporates concurrent actions and restrictions on them naturally and intuitively. It requires no special accommodations to be made for concurrency other than those required by the nature of the domain. These, in turn, can be incorporated within the original framework by simply adding the required executability conditions.

8.6 Non-determinism in $\mathcal{AL}$

A transition diagram $\mathcal{T}$ is called **deterministic** if for every state σ and action a there is at most one state σ' such that $\langle \sigma, a, \sigma' \rangle \in \mathcal{T}$. Perhaps somewhat surprisingly, system descriptions of $\mathcal{AL}$ are capable of describing non-deterministic diagrams. Consider, for instance, the system described by the following laws:

1. a **causes** f **if** $\neg f$
2. $\neg q$ **if** f, p
3. $\neg p$ **if** f, q

where f, q, and p are inertial fluents. Suppose action a is executed in state $\sigma_0 = \{\neg f, p, q\}$. What would be the states the system can move into as a result of this execution? It is clear that every such state must contain

f. But what about p? One may notice that there are two possibilities: first, p may maintain its value by inertia. In this case the system will move into state $\sigma_1 = \{f, p, \neg q\}$ (where q is false due to the second state constraint of our system description). Symmetrically, q may be preserved by inertia and the system may move into $\sigma_2 = \{f, \neg p, q\}$. Figure 8.9 shows the possible transitions in the system; the non-deterministic transitions are shaded. One can easily check that these are exactly the transitions produced by our semantics.

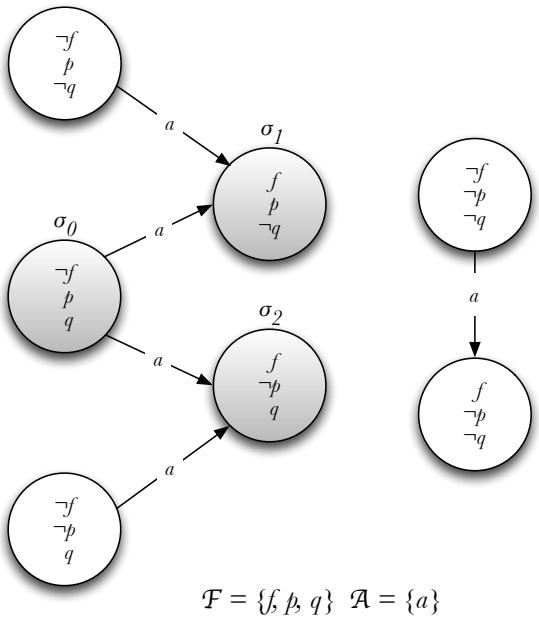


Figure 8.9: Non-Deterministic System Description

The non-determinism of the resulting system can be attributed to the incompleteness of our specification and not to the real non-determinism of causal effects of action a. On the other hand, experts disagree whether actions are really non-deterministic or the impression of non-determinism is caused by our lack of knowledge. We will not discuss these issues here. Instead we simply advise users of $\mathcal{AL}$ to make sure that their descriptions are sufficiently complete and, hence, the corresponding systems are deterministic.

8.7 Temporal Projection

In this chapter we introduced the syntax and semantics of action language $\mathcal{AL}$ which allows a concise and accurate description of transition diagrams of dynamic systems. The fact that the semantics of the language is given in terms of the semantics of ASP allows us to reduce answering questions about the values of fluents of the domain along a given trajectory to computing answer sets of simple logic programs. The task of predicting such values is often referred to as **temporal projection**. We already encountered this task in Section 8.1 where we dealt with predicting positions of blocks after a sequences of moves. The method is rather general. Given a system description $\mathcal{D}$, an initial state σ_0, and a sequence of consecutive occurrences of actions $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$, one can compute the trajectories of the system by combining $\Pi^n(\mathcal{D}) \cup h(\sigma_0, 0)$ with the encoding

$$occurs(a_0, 0).$$

$$\vdots$$

$$occurs(a_{n-1}, n-1).$$

of the sequence and computing the answer sets of the resulting program $\Pi^n(\mathcal{D}, \sigma_0, \alpha)$. It is not difficult to show that a sequence

$$\langle \sigma_0, a_0, \ldots, a_{n-1}, \sigma_n \rangle$$

is a trajectory of $\mathcal{D}$ defined by sequence α iff there is an answer set A of $\Pi^n(\mathcal{D}, \sigma_0, \alpha)$ such that for every $0 \leq i \leq n$

$$\sigma_i = \{l : h(l, i) \in A\}.$$

Consider the briefcase example. If we want to predict what will happen in the state where neither clasp is up if we toggle the first clasp then toggle the second clasp, we construct $\Pi^2(\mathcal{D}_{bc}, \sigma_0, \alpha)$. This is done by replacing `#const n = 1` by `#const n = 2` and combining $\Pi^2(\mathcal{D}_{bc})$ with the encoding of the initial state

```
-holds(up(1),0).
-holds(up(2),0).
```

and the encoding of the action sequence

```
occurs(toggle(1),0).
occurs(toggle(2),1).
```

The answer set of the resulting program will uniquely define the resulting state.

The following chapters will show how we can use material introduced here to do more-complex reasoning tasks including planning and diagnostics. It is worth noting that this approach to reasoning about actions and change could not have been possible without a good understanding of the representation of defaults in ASP.

Summary

In this chapter we discussed an ASP-based methodology for representing knowledge about discrete dynamic systems. Mathematically, such a system is represented by a transition diagram whose states correspond to physically possible states of the domain and whose arcs are labeled by actions. A transition $\langle \sigma_0, a, \sigma_1 \rangle$ belongs to the diagram iff the execution of action a in state σ_0 can move the system to state σ_1. Even for simple systems the corresponding diagrams may be huge and difficult to represent. In this chapter we showed how this problem can be solved using a simple action language $\mathcal{AL}$. We showed how transition diagrams can be described by collections of $\mathcal{AL}$ statements called system descriptions. These representations are rather concise (a comparatively small system description can define a very large diagram), have intuitive informal semantics, and a comparatively high degree of elaboration tolerance. The formal mathematical semantics of $\mathcal{AL}$ is given in terms of logic programs under answer set semantics. The ability of ASP to represent defaults and direct and indirect effects of actions helped solve the Frame and Ramification problems which, for a long time, prevented researchers from finding precise mathematical definitions of transitions of discrete dynamic systems. We also describe some simple but useful mathematical properties of system descriptions of $\mathcal{AL}$ and illustrate how translations of its system descriptions can be used to solve an important computational task, called temporal projection, consisting of computing the states that the system can move into after the execution of a sequence of actions in a given initial state. In the following chapters we show how this connection between system descriptions and logic programs allows answer set programming to be used to solve a number of more-complex computational problems. The problem of representing dynamic systems is an active and important area of knowledge representation and artificial intelligence. There is by now a number of action languages which differ from each other by underlying fundamental assumptions, the type of dynamic systems they are

meant to model, their ability to combine knowledge into modules, etc. For instance, while $\mathcal{AL}$ is based on the Inertia Axiom, another action language called $\mathcal{C}$ uses a different fundamental assumption called the law of universal causation, which says that everything that is true in the world must have a cause. Language H is an extension of $\mathcal{AL}$ allowing representation of so-called hybrid dynamic systems, i.e. systems allowing both discrete and continuous change. Another extension of $\mathcal{AL}$, called $\mathcal{ALM}$, supplies $\mathcal{AL}$ with modular structure, which supports reusability and simplifies the development of knowledge representation libraries. This is, of course, a very incomplete list. Development of action languages and the study of relationships between them is an important topic for future research.

References and Further Reading

Information about Shakey the robot and STRIPS can be found in [85] and [42]. STRIPS evolved into action description language ADL [90] and planning domain definition language, PDDL, [32]. The latter language and its variants are commonly used in the planning community. The language $\mathcal{AL}$ used in this book is an extension of action language $\mathcal{B}$ [52] which is essentially a subset of the action language from [111]. The investigation of the relationship between these languages and logic programming started in [51] and continued in a number of papers including [111] and [9].

Exercises

1. Prove that
   (a) Every state of system description $\mathcal{SD}$ satisfies the state constraints of $\mathcal{SD}$.
   (b) If the signature of $\mathcal{SD}$ does not contain defined fluents, a state is a complete, consistent set of literals satisfying the state constraints of $\mathcal{SD}$.
2. Given the following $\mathcal{AL}$ system description where fluents f and g are inertial and h is defined:

$$a \textbf{ causes } f \textbf{ if } g$$

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

   (a) Show its translation into the corresponding ASP program.

(b) Check if each of the following is a valid state:

i. $\sigma = \{f, \neg g, \neg h\}$
ii. $\sigma = \{\neg f, \neg g, \neg h\}$
iii. $\sigma = \{\neg f, g, h\}$
iv. $\sigma = \{f, g, \neg h\}$

(c) Draw the transition diagram for the system.

3. Consider the system description presented in Section 8.6.

(a) Explain why $\{f, p, q\}$ is not a valid state of this system description.

(b) Add causal law b **causes** $\neg f$ **if** f to the system description and draw the corresponding transition diagram.

4. Given the following story: *Jenny painted the wall white.*

(a) Represent the story in $\mathcal{AL}$. Assume that to paint a wall a given color, one must have paint of the appropriate color. Initially the wall is yellow and Jenny has the white paint. Jenny paints the wall at step 0. Make sure that your theory entails that at the end of the story the wall is white and not yellow.

(b) Translate the representation to ASP and run it using an ASP solver to predict the values of fluents if Jenny paints the wall white.

(c) Now suppose Jenny has black paint as well and, after painting the wall white, decides to paint it black. Use ASP to do some temporal projection about the values of the fluents after both actions are performed sequentially.

(d) Now go back to the original story, but add another person, say Jill. Jenny and Jill have white paint. Jenny painted a wall white. Jill painted another wall white. Assume two people can't paint the same wall at the same time, but they can paint different walls concurrently. Modify your $\mathcal{AL}$ representation to accommodate the new law, initial situation, and trajectory. Translate to ASP and run the program to compute the answer set after Jill and Jenny painted concurrently. Make sure that if you told the program that they painted the same wall concurrently, there would be no answer set.

5. Given the following story: *Claire always carries her cell phone with her. Claire is at the library.*

   (a) Represent the story in $\mathcal{AL}$. Include any commonsense knowledge necessary to answer the question: "Where is Claire's cell phone?" Make the representation general enough so that when you add the fact that Claire went home, her cell phone's location changes accordingly. Also make it general enough that if we change the cell phone to a pet chihuahua, your program will still make the proper conclusions. Assume locations are distinct. *Hint:* Use inertial fluent $carried(Obj, Person)$ as in, the object is carried by the person.

   (b) Translate the representation to ASP and run it using an ASP solver to predict the values of fluents after Claire returns home.

   (c) Modify your program to include that Rod carries a towel with him everywhere and that Claire and Rod are never at the same place at the same time. Make sure that your program can conclude that Claire's cell phone and Rod's towel are also not at the same place at the same time, even when it does not know where Rod is initially.

6. Given the following story: *Amy, Bruce, Carrie, and Don are vendors at a farmers' market. Amy sells apples and carrots, Bruce sells lettuce, Carrie sells apples, and Don sells cabbage and pears. For simplicity, assume that when a customer buys a type of produce from a vendor, he buys the whole quantity that the vendor has to offer.*

   (a) Suppose that *Sally goes to the farmers' market and buys all the lettuce that Bruce sells and all the apples that Carrie sells. Peter goes to the farmers' market afterwards.* Represent both parts of the story in $\mathcal{AL}$. Make sure that temporal projection on your representation entails that Peter cannot buy lettuce but can buy apples.

   (b) Translate the representation to ASP and run it using an ASP solver to answer the questions.

7. Given the following story: *Jonathan has requirements for playing the Wii: he should make sure that his homework is done, the bed is made, and he has practiced Tae Kwon Do. He can only do one thing at a time. Of course, he cannot make the bed if it already made or do his homework if it is already done or if none was assigned.*

(a) Select an initial situation and a sequence of Jonathan's actions which would allow him to play the Wii. Represent the resulting story in $\mathcal{AL}$. *Hint:* Create a sort *activity* for homework, make_bed, TKD and Wii. Then use actions $do(hw)$, $do(make\_bed)$, $do(tkd)$ and $do(wii)$ to make it easy to express the requirement that actions be mutually exclusive.

(b) Translate the representation to ASP and run it using an ASP solver to answer questions about whether a boy may play the Wii at various future moments. Make sure that the system description part of your program works for other variants of this story.

Chapter 9

Planning Agents

In the next several chapters we will discuss the application of the methodology for representing knowledge about dynamic domains and ASP programming to the design of intelligent agents capable of acting in a changing environment. The design will be based on the agent architecture from Section 1.1. In this chapter we address *planning* — one of the most important and well studied tasks which an intelligent agent should be able to perform (see step 3 of the agent loop from Section 1.1).

9.1 Classical Planning with a Given Horizon

We start with **classical planning** in which a typical problem is defined as follows:

- A **goal** is a set of fluent literals which the agent wants to become true.
- A **plan** for achieving a goal is a sequence of agent actions which takes the system from the current state to one which 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.

In this chapter we consider a special case of the classical planning problem in which the agent has a limit on the length of the allowed plans, and will show how this task can be solved using the ASP programming techniques. The limit is often referred to as the **horizon** of the planning problem.

To solve a classical planning problem $\mathcal{P}$ with horizon n, we will 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 ASP encodings of the system description of $\mathcal{P}$, the current state, and the goal, together with a small, domain-independent ASP program called the **simple planning module**. The encoding of the system description is identical to the one used for temporal projection; i.e., variable I for steps ranges over integers from 0 to n. (In the original encoding, it was enough for us to consider $n = 1$.) The encoding of the current state is identical to the encoding of the initial state from Chapter 8.

To encode the goal, we first introduce a new relation $goal(I)$ which 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 a rule:

```
goal(I) :- holds(g1,I),
           -holds(g2,I).
```

where $g_1 = \{f : f \in G\}$ and $g_2 = \{f : \neg f \in G\}$.

Now let us describe the domain independent part of the program, i.e. the simple planning module. The first two rules define the success of the search for a plan.

```
%% Planning Rules 1 and 2
success :- goal(I),
           I <= n.
:- not success.
```

The first rule defines success as the existence of a step in the system trajectory which satisfies the relation *goal*. The second states that failure is not acceptable — a plan satisfying the goal should be found.

The second part of the planning module generates sequences of actions of the appropriate length which can possibly be the desired plans. One can imagine that the consequences of the execution of such a sequence in the current state are computed by the encoding of the system description of the problem. If these consequences include *success*, a plan is found. (Actual computation of plans is of course quite different — this is just a possible way to think about the program's use.)

In Lparse syntax, the corresponding generator can be written as a simple choice rule[1]:

[1]Recall that we cannot use defined variables in the head of a choice rule; therefore, we used variable *Action* instead of *A* since the latter was defined by our #domain declaration.

```
%% Planning Rule 3
1{occurs(Action,I): action(Action)}1 :- not goal(I),
                                        I < n.
```

This guarantees that every answer set S of a program containing the planning module will contain exactly one occurrence of the statement of the form $occurs(a, i)$ for every i such that $0 \leq i < n$ and $goal(j) \notin S$ for every $0 \leq j \leq i$. In other words, the planner will produce action sequences containing 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. Note that if n is equal to the length of the shortest plan, then the planner will produce all plans of length n. For n larger than the length of the shortest plan, some of the plans produced will be longer than necessary — they may include irrelevant actions executed before the goal is achieved.

A similar effect can be achieved with DLV as shown below. The choice rule is replaced by several new rules where, as before, `n` stands for the horizon.

```
occurs(A,I) | -occurs(A,I) :- action(A),
                              step(I),
                              not goal(I),
                              I < n.

%% Do not allow concurrent actions:
:- action(A1), action(A2),
   step(I),
   occurs(A1,I),
   occurs(A2,I),
   A1 != A2.

%% An action occurs at each step before
%% the goal is achieved:

something_happened(I) :- action(A),
                         step(I),
                         occurs(A,I).
:- step(I), step(J),
   goal(I),
   J < I,
   not something_happened(J).
```

This completes the construction of our program $plan(\mathcal{P}, n)$. The following proposition establishes the relationship between answer sets of this program and the solutions of $\mathcal{P}$.

Proposition 9.1.1. *Let $\mathcal{P}$ be a classical planning problem with a deterministic system description and let $0 < n$. A sequence of actions $a_0, \ldots, a_k$ where $0 \leq k < n$ is a solution of $\mathcal{P}$ with the horizon n iff there is an answer set S of $plan(\mathcal{P}, n)$ such that*
(i) For any $0 < i \leq k$, $occurs(a_i, i-1) \in S$,
(ii) S contains no other atoms formed by occurs.

As mentioned above, if a horizon n is larger than the shortest plan needed to satisfy our goal, the planner may find plans containing irrelevant, unnecessary actions. To avoid this problem, we can use the planner to look for plans of lengths 1,2, etc., until a plan is found. This can be accomplished by finding answer sets of programs $plan(\mathcal{P}, k)$ with $k = 1, 2, \ldots, n$. If $k = m$ is the smallest number for which this program is consistent, the shortest solutions of the planning problem $\mathcal{P}$ will be given by answer sets of $plan(\mathcal{P}, k)$. If $plan(\mathcal{P}, k)$ is inconsistent for every $1 \leq k \leq n$, then problem $\mathcal{P}$ has no solution. There is no known way of automatically finding a minimal-length plan without multiple calls; however, Section 9.5 describes two simple extensions of ASP that achieve the task.

9.2 Examples of Classical Planning

9.2.1 Planning in the Blocks World

Let us start with considering the blocks world from Section 8.5.2. The ASP encoding of the system description of this domain is given by program `bw.lp` from that section. Our goal is to create a new program, `bwplan.lp`, which will find plans for the blocks world. The main thing we have to do is add the simple planning module defined above to `bw.lp` from Chapter 8. Since our transition system in `bw.lp` was only defined for steps 0 and 1, we will need to set the horizon by changing the definition of step as follows. Replace `step={0,1}` with

```
#const n=8.
step(0..n).
```

As an added convenience, `gringo` (and therefore `clingo`) allows us to override the constant definition from the command line. Therefore, whenever we

wish to run the program with a different horizon, we could simply change our call. For example, the call

```
clingo 0 -c n=9 bwplan.lp
```

would set the horizon to 9.

Further, we will wish to format the output by adding the hide and show statements to display only the positive *occurs* statements.

```
#hide.
#show occurs(A,I).
```

This will extract the plans from the corresponding answer sets. If the program is written in DLV, use option `-pfilter=occurs`.

That's it. We have our blocks world planner, `bwplan.lp`. Try giving it a variety of initial states and goals. For example, you can use the initial state as encoded in Section 8.1, and the goal defined by the following rule:

```
goal(I) :-
   holds(on(b4,t),I), holds(on(b6,t),I),  holds(on(b1,t),I),
   holds(on(b3,b4),I), holds(on(b7, b3),I), holds(on(b2,b6),I),
   holds(on(b0,b1),I), holds(on(b5,b0),I).
```

Let the horizon of our planning problem be $n = 8$. Now we have a planning problem $BW_1(8)$ and its logic programming encoding.

The call,

```
clingo 0 bwplan.lp
```

will find the problem's solutions. For instance one of the answer sets of the program will contain the following sequence:

```
occurs(put(b2,t),0)
occurs(put(b4,b1),1)
occurs(put(b3,b4),2)
occurs(put(b7,b3),3)
occurs(put(b6,t),4)
occurs(put(b2,b6),5)
occurs(put(b0,b1),6)
occurs(put(b5,b0),7)
```

To make the output more readable, we will normally sort the predicates with respect to the second parameter. This may or may not be done by the solver which knows nothing about the significance of this order.

If we try to run the same program with $n = 7$, we discover that the program is inconsistent. Hence, the above sequence is a shortest solution of our program.

Now let's change n to 9 and ask our solver for all possible answer sets. It outputs quite a few. Some of them only have 8 steps. Here's one with 9 steps:

```
occurs(put(b2,t),0)
occurs(put(b7,t),1)
occurs(put(b6,t),2)
occurs(put(b4,t),3)
occurs(put(b3,b4),4)
occurs(put(b0,b1),5)
occurs(put(b5,b0),6)
occurs(put(b7,b3),7)
occurs(put(b2,b6),8)
```

Notice that this plan contains a wasteful action; by putting $b2$ on the table first, the planner misses the opportunity to put it directly on $b6$ at a later time. As we noted earlier, there is nothing in the code that insists that the planner find *only* the shortest plans — just that it not exceed the horizon. Therefore, it finds all plans that meet the conditions — shortest and otherwise.

Program `bwplan.lp` is a typical example of **Answer Set Planning**. It consists of the theory of the blocks world, the description of the initial state, the goal, the horizon, and the planning module. Note that our solution is completely independent of the problem description. We can change the initial state, the goal, and the horizon at will, without changing the rest of the program. Note that just because the theory is independent of the planning module does not mean that we cannot use specific domain knowledge to guide a planner. The separation between domain knowledge and search strategy makes it easy to write domain-specific rules describing actions that we can ignore in our search. This is covered in Section 9.3. As we will see later, the blocks theory is also completely independent of planning — it can be used for multiple purposes.

Answer set planning does not require any specialized planning algorithm. The "planning" query is answered by the same reasoning mechanism used

for other types of queries. *Therefore, the planning program can be easily generalized and improved.*

To illustrate, we begin by giving a few additional examples of planning in the blocks world, followed by examples in a variety of other domains.

Example 9.2.1. *(Multiple Goal States)*
Note that each block of our domain is accounted for in the goal of problem B_1 and there is only one possible goal state. This is certainly not a requirement and it is perfectly fine to have multiple goal states. For example, we may only wish to require that $b3$ be on the table in which case we write:

```
goal(I) :- holds(on(b3,t),I).
```

We let $n = 2$ and call the new program `bwplanb3.lp`. The solutions of the new problem can be found by calling an ASP solver. For example:

```
clingo 0 bwplanb3.lp
```

```
Answer: 1
occurs(put(b2,b4),0) occurs(put(b3,t),1)
Answer: 2
occurs(put(b2,b7),0) occurs(put(b3,t),1)
Answer: 3
occurs(put(b2,t),0) occurs(put(b3,t),1)
```

Example 9.2.2. *(Using Defined Fluents in the Goal)*

Let us now consider an extension of the system description of the blocks-world by a new defined fluent, *occupied*(*block*). The corresponding definition is given by the following state constraint:

$$occupied(B) \textbf{ if } on(B_1, B)$$

We create a new program, `occupied.lp`, which will incorporate this concept into our current blocks-world planner as specified by `bwplan.lp`. First, we expand our planner by the logic programming translation of this law; i.e., we add statements

```
fluent(defined, occupied(B)).
holds(occupied(B),I) :- holds(on(B1,B),I).
```

Note that statement

```
-holds(occupied(B),I) :- not holds(occupied(B),I).
```

is not necessary because we already have the CWA for defined fluents in `bwplan.lp`.

Suppose now that we want blocks $b0$ and $b1$ to be unoccupied, but we do not care about the rest of the blocks. We state the goal as follows:

```
goal(I) :- -holds(occupied(b0),I),
           -holds(occupied(b1),I).
```

Running program `occupied.lp` with horizon 3 and the same initial state as in `bwplan.lp`, we get 45 answer sets corresponding to shortest plans. Here are three examples:

```
Answer: 1
occurs(put(b4,t),0) occurs(put(b2,b7),1) occurs(put(b3,b2),2)
Answer: 2
occurs(put(b4,t),0) occurs(put(b2,b4),1) occurs(put(b3,b2),2)
Answer: 3
occurs(put(b4,t),0) occurs(put(b2,t),1) occurs(put(b3,b2),2)
```

Note that setting the horizon to 2 will correctly produce no answer sets and setting it to 4 will give 837 answer sets, most of which contain a useless action.

This example shows that defined fluents can be useful in stating our goals. In fact, we may wish to define fluents specifically for this purpose.

Example 9.2.3. *(Defining Complex Goals)*
Further extension of the basic blocks-world domain can be obtained by supplying blocks with colors. This can be done by simply expanding our blocks-world $\mathcal{AL}$ system description from the previous example by a new sort, say *color*, with values *white* or *red* and a new static relation

$$is\_colored(B, C)$$

which holds iff block B is of color C. We assume that each block can have at most one color, i.e. the states of our domain should satisfy the following requirement:

$$\neg is\_colored(B, C_1) \textbf{ if } is\_colored(B, C_2), C_1 \neq C_2$$

A possible goal for such a domain could be the general requirement that all towers must have a red block on top of them. To express this we will use

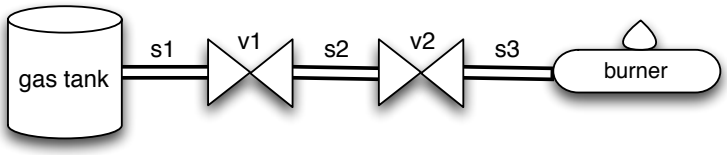


Figure 9.1: Pipeline Configuration

a defined fluent $wrong\_config$ (wrong configuration) which holds iff there is an unoccupied block B which is not red. (Of course a block is unoccupied if and only if it is located on top of some (possibly empty) tower.) This can be specified using fluent 'occupied" from Example 9.2.2:

$$fluent(defined, wrong\_config(B))$$
$$wrong\_config \textbf{ if } \neg occupied(B), \neg is\_colored(B, red)$$

The goal of our problem will be of the form:

```
goal(I) :- -holds(wrong_config, I).
```

All these examples show that our planning methods have a reasonable degree of elaboration tolerance with respect to possible extensions of the initial planning domain and modifications of goals and initial situations. Note also that our translation of system descriptions into ASP programs is modular; i.e., extension of a system description is simply translated into ASP and added to the original program without necessitating further changes.

The next two examples show the application of our planning methodology to very different domains.

9.2.2 Igniting the Burner

Consider the following domain

> A burner is connected to a gas tank through a pipeline. The gas tank is on the left-most end of the pipeline and the burner is on the right-most end. The pipeline is made up of sections connected with each other by valves. The pipe sections can be either pressurized by the tank or unpressurized. Opening a valve causes the section on its right side to be pressurized if the section to its left is pressurized. Moreover, for safety reasons, a valve can be opened only if the next valve in the line is closed. Closing a valve causes the pipe section on its right side to be unpressurized.

We will associate this domain with a planning problem of starting a flame in the burner. We start with describing an $\mathcal{AL}$ representation of this domain. The signature will contain names for sections of the pipeline, $s_1, s_2, \ldots$, and for valves $v_1, v_2, \ldots$. The pipeline will be described by static relations *connected_to_tank*(S), *connected_to_burner*(S) and *connected*(S_1, V, S_2). (The latter holds if sections S_1 and S_2 are connected by valve V and the flow of gas is directed from S_1 to S_2.) Relation *connected* defines the pipeline as the directed graph with source s_0 and sink s_n where s_0 is connected to the tank and s_n is connected to the burner. We will also need inertial fluents: *opened*(V) and *burner_on*, and defined fluent *pressurized*(S). The actions are *open*(V), *close*(V), and *ignite* — open and close the corresponding valves and ignite the burner. The state of the domain and its actions are characterized by the following system description:

pressurized(S) **if** *connected_to_tank*(S).
pressurized$(S2)$ **if** *connected*$(S1, V, S2)$,
opened(V),
pressurized$(S1)$.
$\neg$*burner_on* **if** *connected_to_burner*(S),
$\neg$*pressurized*(S).
open(V) **causes** *opened*(V).
impossible *open*(V) **if** *opened*(V).
impossible *open*$(V1)$ **if** *connected*$(S1, V1, S2)$,
connected$(S2, V2, S3)$,
opened$(V2)$.
close(V) **causes** $\neg$*opened*(V).
impossible *close*(V) **if** $\neg$*opened*(V).
ignite **causes** *burner_on*.
impossible *ignite* **if** *connected_to_burner*(S),
$\neg$*pressurized*(S).

We will use this description in conjunction with the specification of a pipeline which includes the pipeline configuration satisfying the above conditions and the status of the pipeline valves. We also assume that initially the burner is off.

Suppose the pipeline looks like the one modeled in Figure 9.1. Then an example initial situation could be:

$$\{\neg \textit{burner\_on}, \neg \textit{opened}(v1), \textit{opened}(v2)\}.$$

A goal can be defined as

burner_on

Here is the ASP encoding. We call the program `ignite.lp`:

```
%% ignite.lp

section(s1).
section(s2).
section(s3).
#domain section(S).
#domain section(S1).
#domain section(S2).
#domain section(S3).

valve(v1).
valve(v2).
#domain valve(V).
#domain valve(V1).
#domain valve(V2).

connected_to_tank(s1).
connected(s1,v1,s2).
connected(s2,v2,s3).
connected_to_burner(s3).

fluent(inertial, burner_on).
fluent(inertial, opened(V)).
fluent(defined, pressurized(S)).

action(open(V)).
action(close(V)).
action(ignite).
#domain action(A).

%% We can make n as large as we wish.
#const n = 4.
step(0..n).
#domain step(I).

%% ************************************************************
%% AL System Description:
%% ************************************************************
```

```
%% pressurized(S) if connected_to_tank(S).
holds(pressurized(S),I) :- connected_to_tank(S).

%% pressurized(S2) if connected(S1,V,S2),
%%                    opened(V),
%%                    pressurized(S1).
holds(pressurized(S2), I) :- connected(S1,V,S2),
                             holds(opened(V), I),
                             holds(pressurized(S1), I).

%% -burner_on if connected_to_burner(S),
%%               -pressurized(S).
-holds(burner_on, I) :- connected_to_burner(S),
                        -holds(pressurized(S),I).

%% open(V) causes opened(V).
holds(opened(V), I+1) :- occurs(open(V),I),
                         I < n.

%% impossible open(V) if opened(V).
-occurs(open(V),I) :- holds(opened(V),I).

%% impossible open(V) if connected(S1,V1,S2),
%%                       connected(S2,V2,S3),
%%                       opened(V2).
-occurs(open(V),I) :- connected(S1,V1,S2),
                      connected(S2,V2,S3),
                      holds(opened(V2),I).

%% close(V) causes -opened(V).
-holds(opened(V), I+1) :- occurs(close(V), I),
                          I < n.

%% impossible close(V) if -opened(V).
-occurs(close(V), I) :- -holds(opened(V), I).

%% ignite causes burner_on.
holds(burner_on, I+1) :- occurs(ignite, I),
                         I < n.
```

```
%% impossible ignite if connected_to_burner(S),
%%                       -pressurized(S).
-occurs(ignite, I) :- connected_to_burner(S),
                      -holds(pressurized(S), I).

%% CWA for Defined Fluents:
-holds(F,I) :- fluent(defined,F),
               not holds(F,I).

%% General 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) :- not occurs(A,I).

%% ************************************************************
%% Simple Planning Module
%% ************************************************************
success :- goal(I),
           I <= n.
:- not success.

1{occurs(Action,I): action(Action)}1 :- not goal(I),
                                        I < n.

%% ************************************************************
%% Initial Situation:
%% ************************************************************
-holds(burner_on, 0).
-holds(opened(v1),0).
holds(opened(v2),0).
```

```
%% ************************************************************
%% Goal:
%% ************************************************************
goal(I) :- holds(burner_on,I).
%% ************************************************************
%% Output formatting:
%% ************************************************************
#hide.
#show occurs(A,I).
```

The result of invoking `clingo` with `ignite.lp` is:

```
occurs(close(v2),0)
occurs(open(v1),1)
occurs(open(v2),2)
occurs(ignite,3)
```

9.2.3 Missionaries and Cannibals

Another classical planning problem is stated as follows:

> Three missionaries and three cannibals come to a river and find a boat that holds at most two people. If the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten. How can they all cross?

Here is one possible implementation. Our objects are missionaries, cannibals, a boat, and two locations corresponding to the two banks. We will use variables $N, N1, N2, NC, NM, NCSource, NMSource$ to stand for the number of cannibals or missionaries; they are integers in range 0 to 3.

Similarly, variable names starting with NB will stand for the number of boats and can have values of either 0 or 1. We choose to represent the number of missionaries, cannibals and boats at a location with inertial fluents as follows:

$$m(Loc, N)$$

$$c(Loc, N)$$

$$b(Loc, NB)$$

where Loc is location $bank1$ or $bank2$. For example, $m(bank1, 3)$ means that there are three missionaries on $bank1$. Another important inertial fluent is

$$casualties$$

which is true if the cannibals ever outnumber the missionaries on the same bank.

Possible actions in this story are ones having to do with the movements of missionaries and cannibals. We use predicate

$$move(NC, NM, Dest)$$

to represent moving, by boat, NC cannibals and NM missionaries to destination $Dest$. For example, $move(0, 1, bank2)$ means to move one missionary to $bank2$.

To define some of our laws, we will need to know the source of the movement, not just the destination. We know that the source will always be the opposite bank, so we define a static relation:

$$opposite(bank1, bank2)$$

$$opposite(bank2, bank1)$$

The $\mathcal{AL}$ system description consists of the following laws:

- Moving objects increases the number of objects at the destination by the amount moved:

  $move(NC, NM, Dest)$ **causes** $m(Dest, N + NM)$ **if** $m(Dest, N)$
  $move(NC, NM, Dest)$ **causes** $c(Dest, N + NC)$ **if** $c(Dest, N)$
  $move(NC, NM, Dest)$ **causes** $b(Dest, 1)$

- The number of missionaries/cannibals at the opposite bank is 3 – number_on_this_bank. The number of boats at the opposite bank is 1 – number_of_boats_on_this_bank:

  $m(Source, 3 - N)$ **if** $m(Dest, N)$, $opposite(Source, Dest)$
  $c(Source, 3 - N)$ **if** $c(Dest, N)$, $opposite(Source, Dest)$
  $b(Source, 1 - NB)$ **if** $b(Dest, NB)$, $opposite(Source, Dest)$

- There cannot be different numbers of the same type of person at the same location:

  $\neg m(Loc, N1)$ **if** $m(Loc, N2), N1 \neq N2$
  $\neg c(Loc, N1)$ **if** $c(Loc, N2), N1 \neq N2$

- A boat cannot be in and not in a location:

$$\neg b(Loc, NB1) \textbf{ if } b(Loc, NB2), NB1 \neq NB2$$

- A boat cannot be in two places at once:

$$\neg b(Loc1, N) \textbf{ if } b(Loc2, N), Loc1 \neq Loc2$$

- There will be casualties if cannibals outnumber missionaries:

$$\begin{aligned} casualties \textbf{ if } & m(Loc, NM), \\ & c(Loc, NC), \\ & NM > 0, NM < NC \end{aligned}$$

- It is impossible to move more than two people at the same time; it is also impossible to move less than 1 person:

$$\begin{aligned} &\textbf{impossible } move(NC, NM, Dest) \textbf{ if } (NC + NM) > 2 \\ &\textbf{impossible } move(NC, NM, Dest) \textbf{ if } (NM + NC) < 1 \end{aligned}$$

- It is impossible to move objects without a boat at the source:

$$\begin{aligned} \textbf{impossible } move(NC, NM, Dest) \textbf{ if } & opposite(Source, Dest), \\ & b(Source, 0) \end{aligned}$$

- It is impossible to move N objects from a source if there are not at least N objects at the source in the first place:

$$\begin{aligned} \textbf{impossible } move(NC, NM, Dest) \textbf{ if } & opposite(Source, Dest), \\ & m(Source, NMSource), \\ & NMSource < NM \\ \textbf{impossible } move(NC, NM, Dest) \textbf{ if } & opposite(Source, Dest), \\ & c(Source, NCSource), \\ & NCSource < NC \end{aligned}$$

Below we present the ASP implementation.

```
%% crossing.lp

%% ----------
%% Signature:
```

```
%% ----------

%% Steps:
#const n = 11.
step(0..n).
#domain step(I).

location(bank1).
location(bank2).
#domain location(Loc).
#domain location(Loc1).
#domain location(Loc2).
#domain location(Source).
#domain location(Dest).
%% Number of cannibals/missionaries:
num(0..3).
#domain num(N).
#domain num(N1).
#domain num(N2).
#domain num(NC).
#domain num(NM).
#domain num(NCSource).
#domain num(NMSource).
%% Number of Boats:
num_boats(0..1).
#domain num_boats(NB).
#domain num_boats(NB1).
#domain num_boats(NB2).

%% --------
%% Statics:
%% --------

%% opposite bank:
opposite(bank1,bank2).
opposite(bank2,bank1).

%% --------
%% Fluents:
%% --------
```

```
%% number of missionaries at location Loc is N:
fluent(inertial, m(Loc, N)).

%% number of cannibals at location Loc is N:
fluent(inertial, c(Loc, N)).

%% number of boats at location Loc is NB:
fluent(inertial, b(Loc, NB)).

%% true if cannibals outnumber missionaries on the same bank:
fluent(inertial, casualties).

%% --------
%% Actions:
%% --------

%% move NC (a given number of cannibals) and NM (a given number
%% of missionaries) to Dest (a destination):
action(move(NC, NM, Dest)).
#domain action(A).

%%-----------------------------------
%% Encoding of AL System Description:
%%-----------------------------------

%% Moving objects increases the number of objects at the
%% destination by the amount moved.

holds(m(Dest, N+NM), I+1) :- holds(m(Dest,N),I),
                             occurs(move(NC, NM, Dest), I),
                             I < n.

holds(c(Dest, N+NC), I+1) :- holds(c(Dest,N),I),
                             occurs(move(NC, NM, Dest), I),
                             I < n.

holds(b(Dest, 1), I+1) :- occurs(move(NC, NM, Dest), I),
                          I < n.
```

```
%% The number of missionaries/cannibals at the opposite bank
%% is 3 - number_on_this_bank. The number of boats at the
%% opposite bank is 1-number_of_boats_on_this_bank.

holds(m(Source, 3-N),I) :- holds(m(Dest, N),I),
                           opposite(Source,Dest).

holds(c(Source, 3-N),I) :- holds(c(Dest, N),I),
                           opposite(Source,Dest).

holds(b(Source, 1-NB), I) :- holds(b(Dest,NB),I),
                             opposite(Source,Dest).

%% There cannot be different numbers of the same type of person
%% at the same location.
-holds(m(Loc, N1), I) :- holds(m(Loc, N2),I), N1 != N2.
-holds(c(Loc, N1), I) :- holds(c(Loc, N2),I), N1 != N2.

%% A boat can't be in and not in a location
-holds(b(Loc, NB1), I) :- holds(b(Loc, NB2), I), NB1 != NB2.

%% A boat can't be in two places at once.
-holds(b(Loc1, N), I) :- holds(b(Loc2, N),I), Loc1 != Loc2.

%% There will be casualties if cannibals outnumber missionaries:
holds(casualties,I) :- holds(m(Loc, NM),I),
                       holds(c(Loc, NC),I),
                       NM > 0, NM < NC.

%% It is impossible to move more than two people at the same time;
%% it is also impossible to move less than 1 person.
-occurs(move(NC,NM,Dest),I) :- (NC+NM) > 2.
-occurs(move(NC,NM,Dest),I) :- (NM+NC) < 1.

%% It is impossible to move objects without a boat at the source.
-occurs(move(NC,NM,Dest), I) :- opposite(Source,Dest),
                                holds(b(Source,0),I).

%% It is impossible to move N objects from a source if there
%% aren't at least N objects at the source in the first place.
```

```
-occurs(move(NC,NM,Dest), I) :- opposite(Source,Dest),
                                holds(m(Source,NMSource), I),
                                NMSource < NM.
-occurs(move(NC,NM,Dest), I) :- opposite(Source,Dest),
                                holds(c(Source,NCSource),I),
                                NCSource < NC.

%%---------------
%% 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) :- not occurs(A,I).

%% ------------------
%% Initial Situation:
%% ------------------

holds(m(bank1,3), 0).
holds(c(bank1,3), 0).
holds(b(bank1,1), 0).
-holds(casualties,0).

%% -----
%% Goal:
%% -----
```

```
goal(I) :-
   -holds(casualties,I),
    holds(m(bank2,3),I),
    holds(c(bank2,3),I).

%% ----------------
%% Planning Module:
%% ----------------

success :- goal(I),
           I <= n.
:- not success.

1{occurs(Action,I): action(Action)}1 :- not goal(I),
                                        I < n.

#hide.
#show occurs(A,I).
```

We called the program `crossing.lp` and invoked it with
`clingo 0 crossing.lp`
`clingo` came up with four answer sets, one of which we show below:

```
occurs(move(1,1,bank2),0)
occurs(move(0,1,bank1),1)
occurs(move(2,0,bank2),2)
occurs(move(1,0,bank1),3)
occurs(move(0,2,bank2),4)
occurs(move(1,1,bank1),5)
occurs(move(0,2,bank2),6)
occurs(move(1,0,bank1),7)
occurs(move(2,0,bank2),8)
occurs(move(0,1,bank1),9)
occurs(move(1,1,bank2),10)
```

9.3 Heuristics

The efficiency of ASP planners can be substantially improved by expanding a planning module by domain dependent heuristics represented by ASP

rules. As an example consider our blocks world planning problem. Note that plans produced by planning module in `bwplan.lp` may contain actions of the form $put(B, L)$ even when B is already located at L. Such an action can be executed by the robot's arm by lifting B up and putting it back on L or by simply doing nothing. In any case the action is completely unnecessary and can be eliminated from considerations of the planner. This can be done by the following heuristic rule:

```
:- holds(on(B,L),I),
   occurs(put(B,L),I).
```

The additional information guarantees that the program will not generate plans containing this type of useless action.

The planning module can be expanded by another useful heuristic which tells the planner to only consider moving the blocks that are out of place. Notice that this heuristic is defined naturally in terms of subgoals because our towers are defined in terms of individual block placement. However, in our current encoding, this information is hidden in the rule defining relation $goal(I)$. To make it explicit we introduce a new relation

$$subgoal(fluent, boolean)$$

and expand our program by the following:

```
subgoal(on(b4,t),true).
subgoal(on(b6,t),true).
subgoal(on(b1,t),true).
subgoal(on(b3,b4),true).
subgoal(on(b7,b3),true).
subgoal(on(b2,b6),true).
subgoal(on(b0,b1),true).
subgoal(on(b5,b0),true).
```

This idea can be generalized for all heuristics that are based on knowledge of subgoal interaction.

Now the heuristic can be defined by the following rules:

```
in_place(B,I) :- subgoal(on(B,B1),true),
                 holds(on(B,B1),I),
                 in_place(B1,I).
in_place(B,I) :- subgoal(on(B,t),true),
                 holds(on(B,t),I).
```

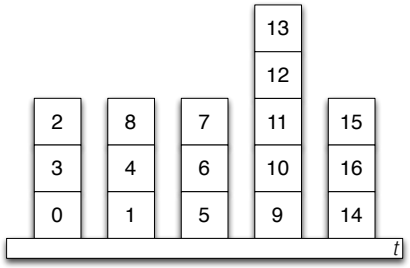


Figure 9.2: Initial Configuration

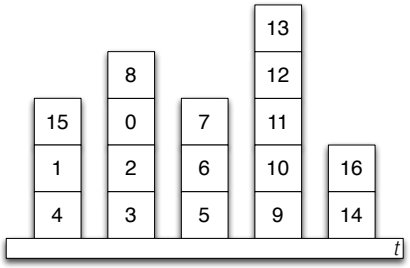


Figure 9.3: Goal Configuration

```
in_place(t,I).

:- holds(in_place(B,I)),
   occurs(put(B,L),I).
```

Again the heuristic will allow to eliminate some "non-optimal" plans. More importantly, it may have a substantial positive effect on the efficiency of the planning program. This is especially true if achieving the agent's goal does not require disassembling some of the towers located on the table. To see the effect of our two heuristics let us look at the following example.

Consider a blocks world with 17 blocks, $b0, \ldots, b16$, an initial block configuration represented by Figure 9.2 and a goal represented by Figure 9.3. The initial configuration will be represented by the following collection of facts:

$$holds(on(b0, t), 0). \quad holds(on(b3, b0), 0). \quad \ldots$$

The goal configuration will be given by rule:

$$goal(I) \leftarrow holds(on(b4, t), I), holds(on(b1, b4), I), \ldots$$

and a collection of subgoals:

$$subgoal(on(b4, t), true). \quad subgoal(on(b1, b4), true). \quad \ldots$$

It may be instructive to run this input with and without two heuristics described above for $n = 7$ (no solution), $n = 8$ (shortest solutions) and $n = 9$. Of course the result will depend on the solver you use but, most likely, you will be able to see that addition of the heuristics improves performance of the program. (On our computer we have an approximately 5 times increase

in efficiency of the planner tested with `clingo`[2].) There is, however, only a slight improvement in the quality of plans. For $n = 8$ both planners (with and without heuristics) find five best plans of length 9. If $n = 9$ then the planner without heuristics finds 2041 plans while one with the heuristics finds 2014 plans. Only 27 non-optimal plans are eliminated.

Additional heuristics may help to further improve efficiency and to eliminate a much larger number of non-optimal plans. Consider, for instance, another domain dependent heuristic which gives priority to actions that increase the number of blocks placed in the right position. This can be written as follows:

```
good_move(B,L,I) :-
                subgoal(on(B,L),true),
                in_place(L,I),
                -occupied(L,I),
                -occupied(B,I).

occupied(B,I) :- holds(on(B1,B),I).
-occupied(t,I).
-occupied(B,I) :- not occupied(B,I).
```

The first rule suggests that, if possible, it is good to move block B on its required position L. The last three simply define relation *occupied*.

Note that it may be tempting to express our heuristic by a rule requiring an agent to execute a good move. This can be done by the rule

```
occurs(put(B,L),I) :- good_move(B,L,I).
```

Unfortunately, this may lead to inconsistency. Remember that the domain has only one robotic arm and hence can execute only one action at a time. There are, however, situations with more than one good move.

We need to find another representation. The following, slightly more complex, rules will do the job.

```
exists_good_move(I) :- good_move(B,L,I).

:- exists_good_move(I),
   occurs(put(B,L),I),
   not good_move(B,L,I).
```

[2]Note that we normally test performance of our programs using `clingo`, which is often more efficient than our publically-available version of DLV. This is not necessarily the case for the commercial version of DLV.

Instead of requiring good moves to be executed, these rules prohibit the execution of "bad" moves.

Let us now try a new planner containing all three heuristics on instances of the blocks world problem from the above example. This time you will see not only further improvement in efficiency but also in the number of non-optimal plans eliminated by the heuristics. Instead of 2014 plans found by the planner which uses the first two heuristics for $n = 9$ the new planner finds 42 plans. With the increase of n the advantages of using all three heuristics becomes even more obvious.

It is difficult to overestimate the importance of heuristic information for our ability to solve difficult search problems. There is of course a substantial amount of work related to heuristics. Among other things researchers are trying to discover criteria allowing us to estimate the usefulness of a heuristics and to find algorithms to automatically derive heuristics which are good for a given problem. Much of this work is done for planners based on specialized planning algorithms implemented in procedural planners. As previous examples show heuristics can also be expressed declaratively but much more work is needed to fully understand how to evaluate and automatically learn heuristics in the context of declarative planning methods.

9.4 Concurrent Planning

If our domain allows simultaneous execution of several actions, we may want to look for so called concurrent plans in which more than one action can be executed at each step. To adapt our $plan(\mathcal{P}, n)$ to this situation, all we need to do is to change the planner's "generator" rule as follows:

```
1{occurs(Action,I): action(Action)}m :- not goal(I),
                                        I < n.
```

Here m is the maximum number of actions which can be performed simultaneously. That's it. Answer sets of this new planning module used together with the problem description will produce concurrent plans for achieving the problem's goal.

To better understand the behavior of concurrent planners, let us again look at the blocks world but now assume that the domain contains two robotic arms that are able to operate independently. We'll construct a new program, `twoarmplan.lp`, from `bwplan.lp` by changing the generator rule as described above; i.e., we simply set $m = 2$. As discussed in Section 8.5.3,

we will need to prohibit the program from putting something on a moving block. To achieve this, we add rule

```
-occurs(put(B1,L),I) | -occurs(put(B2,B1),I)
```

to `twoarmsplan.lp`, just as we did in Section 8.5.3.

We set the horizon to 5 and run `clingo` as follows:

```
clingo --shift twoarmsplan.lp
```

The program returns one of the possible plans, say:

```
occurs(put(b7,t),0)
occurs(put(b4,t),0)
occurs(put(b2,t),1)
occurs(put(b6,t),1)
occurs(put(b2,b6),2)
occurs(put(b3,b4),2)
occurs(put(b0,b1),3)
occurs(put(b7,b3),3)
occurs(put(b5,b0),4)
```

The plan contains nine actions executed in four steps. The goal cannot be reached in three steps (i.e. the program with $n = 4$ is inconsistent). Note, however, that this plan is not optimal; i.e. there are plans containing a smaller number of actions. For example the plan

```
occurs(put(b4,t),0)
occurs(put(b2,t),0)
occurs(put(b3,b4),1)
occurs(put(b0,b1),2)
occurs(put(b7,b3),2)
occurs(put(b6,t),3)
occurs(put(b5,b0),4)
occurs(put(b2,b6),4)
```

consist of 8 actions. In the next section we show how to find optimal plans for this and similar problems.

9.5 (*) Finding Minimal Plans

It is, of course, desirable not to be forced to make multiple calls to a solver or to be able to make very good guesses about the horizon of planning problems

in order to find minimal plans. Unfortunately, there is no known simple way to reduce this to computing answer sets of ASP programs. In this section we discuss how this can be done using two extensions of ASP — CR-Prolog introduced in Section 5.5 and ASP with minimality statements implemented on top of many ASP systems such as Smodels and `clingo`.

To use CR-Prolog to find minimal solutions to planning problems, we use the following rules instead of the Simple Planning Module:

```
#domain action(A).
#domain action(A1).
#domain action(A2).

success :- goal(I),
           I <= n.
:- not success.

r1(A,I):occurs(A,I) +-.

something_happened(I) :- occurs(A,I).
:- not something_happened(I),
   something_happened(I+1).

-occurs(A2,I) :- occurs(A1,I),
                 A1 != A2.
```

The first two rules should be familiar. The next rule simply says that during the planning process the agent may consider occurrences of its actions if they are needed to resolve a contradiction (i.e. to achieve "success"). The next two rules guarantee that the agent does not plan to procrastinate, i.e. to remain idle at some time steps and continue actions afterward. The last rule is used to limit our planner to finding plans which allow the execution of at most one action per step.

The **Simple Planning Module of CR-Prolog** will then consist of the above rules and the cardinality-based preference relation of CR-Prolog. It will allow us to find shortest one action per step solutions of the planning problem with horizon n without resorting to multiple calls of ASP solvers. To obtain the concurrent version of the planner, we simply remove the planner's last rule. The new planner will find solutions of a planning problem which involve the minimal number of actions.

For example, let's replace the simple planning module of `bwplan.lp` with the CR-Prolog simple planning module to create program `crbwplan.lp`.

Recall that running `bwplan.lp` with n=9 gave us plans with useless actions. To run `crbwplan.lp` with horizon 9, use the following command:

```
crmodels -m 0 -c n=9 --min-card --smodels clasp crbwplan.lp
```

`-m 0` means "find all models"

`-c n=9` means "set program constant n to 9"

`--min-card` means "use the cardinality-based preference relation"

`--smodels clasp` means "use clasp as the solver"

You will see that `crmodels` correctly finds the plans with 8 actions, not 9.

Similarly, we can create `crtwoarmsplan.lp` from `twoarmsplan.lp` from the previous section. In this case, we replace the planning module with the CR-Prolog simple planning module without the last rule. Running `crmodels` to find minimal cardinality plans gives us only 2 models, as opposed to the 298 models for `twoarmsplan.lp`, even when the horizons are minimal! Note also that there is another funny thing that happens because of concurrency. If we use `crmodels` to run `crtwoarms.lp` with the horizon set to 6, we get 20 models. Each one has only 8 actions, but some have 5 steps.

We can also compute minimal plans by using a special form of the minimize statement of Lparse and `gringo`. Syntactically, the statement has the form

$$\#minimize\{q(X_1, \ldots, X_n) : s_1(X_1) : \cdots : s_n(X_n)\}$$

where s_is are sorts of the variables occurring in atom $q(X_1, \ldots, X_n)$. The statement, which can be viewed as a directive to an ASP solver, instructs it to compute only those answer sets of the program which contain the smallest number of occurrences of atoms formed by predicate symbol q. To use this for planning, we simply add the statement:

```
#minimize{occurs(Action, K) : action(Action) : step(K)}.
```

to our programs. To see all optimizations, we need to call `clingo` with the command option `--opt-all`. To see how this affects the computations, try running `bwplan.lp` with and without the minimize statement as follows:

```
clingo 0 -c n=9 bwplan.lp --opt-all
```

Summary

In this chapter, we described declarative methodology for solving classical planning problems. The main idea is

1. Use an action language (in our case $\mathcal{AL}$) to represent information about an agent and its domain. x

2. Automatically translate this representation into a program of Answer Set Prolog.

3. Expand the program by the description of the initial state and the goal given by a collection of logic programming facts.

4. Use answer set solvers to compute the answer sets of the resulting program combined with a comparatively small program called the planning module. The maximum number of steps in plans computed by this program is parametrized by non-negative integer n.

5. A collection of facts formed by relation *occurs* which belong to such an answer set corresponds to a plan for achieving the goal in n steps.

The methodology was demonstrated by a number of examples.

The chapter is of interest because of at least two reasons. First, it explains how to declaratively implement one of the most important reasoning steps in the agent loop — searching for plans to achieve the agent's goal. Second, it can be viewed as another typical example of Answer Set Programming. Intuitively the task is solved by generating sequences of actions and testing if their execution satisfies the goal. This basic generate-and-test procedure is independent of the domain. Of course, real generate-and-test is done by answer set solvers and is much smarter than the "blind" generate-and-test of our theoretical models. We also discuss how additional knowledge given in terms of logic programming rules containing heuristic information can further improve efficiency of the search.

Our solution has the typical advantages of Answer Set Programming. It has a reasonably high level of elaboration tolerance. We do not need to alter an existing knowledge base of an agent to add planning capability. Very small changes allow us to go from one-action-at-a-time planning to planning allowing concurrent actions. The resulting programs are provenly correct, i.e. shown to find plans which guarantee success (assuming of course that the world does not change in some unexpected way to interfere with our plans). Existing answer set solvers are sufficiently efficient to provide acceptable

solutions for a substantial number of non-trivial planning problems. In many cases they successfully compete with specialized planners. In other cases no procedural solutions are known. This is especially true in domains where planning requires a large amount of knowledge.

There are, of course, a number of remaining problems. First, ASP-based planners may be inefficient if the required plans are really long. This happens because grounding for programs with a large number n of steps can be too costly. There is extensive ongoing work on efficient grounding in the ASP community which may help to alleviate this problem. Second, ASP solvers may produce plans with a number of redundant, unnecessary actions. A short discussion in the last section of this chapter describes several ways of finding optimal plans that do not contain such actions.

Finally, it is worth mentioning that the chapter only deals with classical planning. There are many different types of planning undergoing extensive study in AI. For instance, if the initial information of the agent is incomplete or some actions in the domain are non-deterministic, the limited approach given in this chapter is not enough. In this situation the transition diagram of our domain may have multiple trajectories which start at possible initial states and are labeled by the same actions. Some of them will lead to the goal while others will not. Our method allows us to find paths leading to the goal but is not sufficient to guarantee that the execution of the corresponding actions will always do so. A sequence of actions such that *all paths* in the diagram which start in possible initial states and are labeled by actions of the sequence lead to states satisfying the goal is called a *conformant* plan. The problem of finding conformant plans is computationally more difficult than simple planning but, when successfully solved, can be very useful. There are also planning and scheduling problems, problem of finding plans which succeed with a high degree of probability, etc. There are substantial advances in these areas. Both procedural and declarative methods for solving such problems (including those based on ASP) have been developed, but much more remains to be done.

Planning is a fascinating and important part of human reasoning and, of course, it is not surprising that it is not yet fully understood and automated. We hope, however, that this chapter gives some insight into a number of problems related to planning and outlines a feasible solution to the simplest (but non-trivial) form of this problem.

References and Further Reading

The origins of declarative planning discussed in this chapter can be found in [58]. In this paper Henry A. Kautz and Bart Selman show how the classical planning problem can be reduced to checking satisfiability of a propositional formula. The use of ASP in planning can be traced to [106] and [35]. A good exposition can be found in [67]. For information about more-complex forms of planning see, for instance, [54], [110], and [92].

Exercises

1. Replace rules

   ```
   success :- goal(I).
   :- not success.
   ```

   in program `bwplan.lp` by the single statement

   ```
   :- not goal(I).
   ```

   What happens? Why?

2. Add the planning module to the briefcase program, `bc.lp`, from Section 8.5.1 and set the horizon. Run the program to find a plan to unlock the briefcase from an initial situation in which both clasps are locked.

3. Create a new program, `colorplan.lp`, that extends `bwplan.lp` by incorporating logic programming translations of the laws defined in Examples 9.2.2 and 9.2.3. Test this program with the following colors of blocks in the initial situation: blocks 0, 3, 4, and 5 are red; the rest are white. Use the new goal and find a minimal plan. (Adjust the horizon accordingly.)

4. Modify the basic system description presented in Example 9.2.3 and use it to write a program that finds a minimal plan to have at least one tower consisting of only red blocks. The program must work with an arbitrary initial configuration (although you can adjust the horizon accordingly).

5. Expand the basic blocks world system description $\mathcal{D}_{bw}$ by information on which blocks are heavy and which are light. Define a tower of blocks to be "uniform" if it consists entirely of one type of block, either all heavy or all light. Use the new system description to write an ASP program to find a minimal plan to create configurations of only uniform towers. The program must work with an arbitrary initial configuration (although you can adjust the horizon accordingly).

6. Give a simple $\mathcal{AL}$ action theory of shooting and use it to write an ASP program which finds a plan for shooting a turkey. You may assume that the theory has actions *load* and *shoot* and fluents *loaded* and *alive*.

7. Give an $\mathcal{AL}$ action theory and use it to create an ASP program to solve the following problem:

   > A farmer needs to get a chicken, some seed, and a fox safely across a river. He has a boat that will carry him and one "item" across the river. The chicken cannot be left with the seed or with the fox without the farmer's presence. How can the farmer get everything across in tact? Note: Don't get cute — there is no bridge, the river's not dry or frozen, there is no cage he can build, etc. The farmer must use the boat to ferry one thing/creature at a time.

8. Create a new system description by extending that given in Example 9.2.3 by adding a new action, $paint(B, C)$, which changes the color of block B to color C. Painting actions can be executed concurrently, but it is impossible to paint a moving block. Use the new theory to create program with colors of the blocks in the initial situation as follows: blocks 0, 3, 4, and 5 are red; the rest are white. Use the "towers must have a red block on top" goal. What is the shortest plan that the program comes up with if action "paint" is allowed? What is the shortest plan if action "paint" does not exist?

9. Convert the program from Exercise 7 in Chapter 8 to a planner that can find a plan for Jonathan to play the Wii.

10. Consider the farmers' market story from Exercise 6 in Chapter 8. Add the knowledge that customers will not buy a product from a vendor if they already have that product. Let us assume that Evaline is the first customer of the day and she knows what every vendor sells. For

each of the following goals, write a program to find a plan to achieve it.

(a) Evaline wants to buy something from every vendor.

(b) She wants to buy one kind of fruit and two different kinds of vegetables.

(c) Expand the problem's knowledge base by a new action $make\_coleslaw(P)$ which is executable if person P has carrots and cabbage. Evaline wants to make coleslaw.

(d) She wants to buy apples, carrots and lettuce from the vendors closest to the entrance which carry these products. Amy has the closest stand followed by Bruce, Carrie, and Don.

Chapter 10

Diagnostic Agents

In this chapter we discuss how to build agents capable of finding explanations of unexpected observations. To do that we divide actions of our domain into two disjoint classes: **agents actions** and **exogenous actions**. As expected the former are those performed by the agent associated with the domain. The latter are those performed by nature or by other agents[1]. As usual we will make some simplifying assumptions. We assume that:

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 occurring in its environment.

Note that the second assumption is defeasible — some exogenous actions can remain unobserved. These assumptions hold in many realistic domains and are suitable for a broad class of applications. In other domains, however, the effects of actions and the truth-values of observations can only be known with a substantial degree of uncertainty which cannot be ignored in the modeling process. We will comment on such situations in Chapter 11 which deals with probabilistic reasoning.

In our setting 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. contradicts the agent's expectations.

[1]If our agent is part of a multi-agent system, it is convenient to view actions of other agents as exogenous, just as we would view the occurrence of any other naturally-occurring action.

- An **explanation** of a symptom is a collection of unobserved past occurrences of exogenous actions which may account for the unexpected observations.

  This notion of explanation is closely connected with our second simplifying assumption. Recall that while assumptions about the agent's ability to perform and observe, as well as its knowledge of causal laws are not defeasible, the completeness of its observations of exogenous actions is defeasible. Accordingly, the agent realizes that it may miss some of the exogenous actions and attributes its wrong prediction to such an omission. A collection of missing occurrences of exogenous actions which 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 latter.

We illustrate this intuition by the following example.

Example 10.0.1. *(A Diagnostic Problem)*
Consider an agent controlling a simple electrical system from Figure 10.1

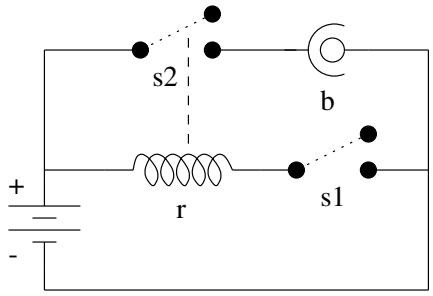


Figure 10.1: $\mathcal{AC}$

consisting of a circuit $\mathcal{AC}$. We assume that switches s_1 and s_2 are mechanical components which cannot become damaged. Relay r is a magnetic coil. If not damaged, it is activated when s_1 is closed, causing s_2 to close. An undamaged bulb b emits light if s_2 is closed. The agent is aware of two exogenous actions relevant to its work: *break*, which causes the circuit bulb to become faulty, and *surge*, which damages the relay and also the bulb if the latter is not protected.

Suppose now that at the beginning of the agent's activity the system is in the state depicted by Figure 10.1, the bulb is protected, both the bulb and

the relay are know to be OK, and the agent closes the switch s_1. The agent expects that this action will activate relay r, and that, as a consequence, switch s_2 will become closed and bulb b will emit light. Assume now that the agent is surprised by an unexpected observation — the light is not lit. There are three natural explanations for this phenomenon: the bulb broke, the relay broke, or both of them broke. In other words one or both of the actions *break* and *surge* occurred in parallel with the agent's closing the switch. It seems that the commonsense reasoner usually ignores the latter explanation and only concentrates on the first two. This is probably due to some notion of minimal or best explanation which usually depends on ordering explanations as determined by the domain. In our case the first two explanations are minimal with respect to set-theoretic and cardinality-based orderings of explanations.

If, in addition to the initial observations, the agent was to observe that the bulb was OK, then the only possible minimal explanation of our unexpected observation would be the occurrence of *surge*. If the bulb was observed to be broken, then the only possible minimal explanation would be the occurrence of *break*. Of course, if initially the bulb was not protected, then both minimal explanations would still be valid. To find out the correct one, the agent would need to perform more "testing" actions, e.g. replace the bulb, etc.

In what follows we make this intuition precise. We start with presenting the syntax and semantics of the system's recorded history, and show how it can be used to predict the current state of the system and discover and explain unexpected observations.

10.1 Recording the History of a Domain

As mentioned above we assumed that, in addition to a description of the transition diagram representing possible trajectories of the system, the knowledge base of a diagnostic agent will contain the system's **recorded history** — observations made by the agent together with a record of its own actions.

Recorded history defines a collection of paths in the diagram which, from the standpoint of the agent, can be interpreted as the system's possible pasts. 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. The following definitions describe syntax and semantics of recorded history of a dynamic system $\mathcal{SD}$ up to the current step n.

Definition 10.1.1. *(Recorded History — Syntax)*
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 10.1.2. *(Recorded History — Semantics)*
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* $0 \leq i < n$

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.*

Definition 10.1.3. *(Entailment)*

- *A fluent literal* l **holds** *in a model* M *of* Γ_{n-1} *at step* $i \leq n$ *(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}*,* $M \models h(l, i)$*.*

Example 10.1.1. *(Recorded History in the Briefcase Domain)*
Let's look back at our briefcase example from Section 8.5.1. The system description, D_{bc}, contains the agent's knowledge about the domain.

$$\begin{array}{l} toggle(C) \ \textbf{causes} \ up(C) \ \textbf{if} \ \neg up(C) \\ toggle(C) \ \textbf{causes} \ \neg up(C) \ \textbf{if} \ up(C) \\ open \ \textbf{if} \ up(c_1), up(c_2) \end{array}$$

Suppose that, initially, the clasp 1 was fastened and the agent unfastened it. The corresponding recorded history is:

$$\Gamma_0 \left\{ \begin{array}{l} obs(up(c_1), false, 0). \\ hpd(toggle(c_1), 0). \end{array} \right.$$

There are two models of Γ_0 that satisfy this history. In both models, our action, a_0, is $toggle(c_1)$, but our states, σ_0 and σ_1 are different. Path $\langle \sigma_0, toggle(c_1), \sigma_1 \rangle$ is a model of Γ_0 if

$$M_1 \begin{cases} \sigma_0 = \{\neg up(c_1), \neg up(c_2), \neg open\} \\ \sigma_1 = \{up(c_1), \neg up(c_2), \neg open\} \end{cases}$$

or

$$M_2 \begin{cases} \sigma_0 = \{\neg up(c_1), up(c_2), \neg open\} \\ \sigma_1 = \{up(c_1), up(c_2), open\} \end{cases}$$

Since Γ_0 has models, it is consistent with respect to the transition diagram for the briefcase domain in Figure 8.8. Although we have a consistent history, our knowledge is not complete. There are two possible trajectories consistent with the agent's recorded history Γ_0. At the end of one, the briefcase is open; at the end of another, it is not. An intelligent agent would need more information to come up with a conclusion about the state of the briefcase. But, since $\Gamma_0 \models holds(up(c_1), 1)$, the agent knows that currently the first clasp is unlocked.

Consider another example where

$$\Gamma_0 \begin{cases} obs(up(c_1), true, 0) \\ obs(up(c_2), true, 0) \\ hpd(toggle(c_1), 0) \\ obs(open, true, 1) \end{cases}$$

This history is not consistent with the transition diagram in Figure 8.8 because it has no model. Note that there is no path in our diagram that we can follow in this situation. This makes sense since the briefcase is open iff both clasps are up, but the first clasp must be down at step 1 as the result of toggle.

10.2 Defining Explanations

Now let us consider an agent which completed the execution of its $(n-1)^{th}$ action. Let us denote the recorded history of the system up to this point by Γ_{n-1}. In accordance with our agent loop, the agent will now observe the values of a collection of fluents at the current step n. Let us denote these observations by O^n. The pair $\mathcal{C} = \langle \Gamma_{n-1}, O^n \rangle$ will be often referred to as the **system configuration**. If new observations are consistent with the

agent's view of the world, i.e. if $\mathcal{C}$ is consistent[2], then O^n simply becomes part of the recorded history. Otherwise, the agent needs to start seeking the explanation(s) of the mismatch. As mentioned earlier, the only possible explanation for unexpected observations would be that *some exogenous action(s) occurred that the agent did not observe.*

This intuition is captured by the following definition:

Definition 10.2.1. *(Possible Explanation)*

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

Example 10.2.1. *(Diagnosing the Circuit-1)*
To illustrate these definitions let us go back to the diagnostic problem from Example 10.0.1. To model the dynamic system from this example, we first need to define its transition diagram. This will be done by the following system description, $\mathcal{D}_{ec}$:

Signature

Components:

$$comp(r)$$
$$comp(b)$$

Switches:

$$switch(s_1)$$
$$switch(s_2)$$

Fluents:

$$fluent(inertial, prot(b))$$
$$fluent(inertial, closed(SW)) \leftarrow switch(SW)$$
$$fluent(inertial, ab(X)) \leftarrow comp(X)$$
$$fluent(defined, active(r))$$
$$fluent(defined, on(b))$$

[2]Note that syntactically $\mathcal{C}$ can be viewed as the recorded history of the system but, unlike recorded history, it is not yet stored in the agent's memory. We will, however, slightly abuse our terminology and talk about inconsistency of system configuration $\mathcal{C}$ understood as the absence of a model of $\mathcal{C}$ viewed as a recorded history.

Actions:

$$action(agent, close(s_1))$$
$$action(exogenous, break)$$
$$action(exogenous, surge)$$

Laws

The causal laws, state constraints and executability conditions describing the normal functioning of our system are encoded as follows:

$close(s_1)$ **causes** $closed(s_1)$
$active(r)$ **if** $closed(s_1), \neg ab(r)$
$closed(s_2)$ **if** $active(r)$
$on(b)$ **if** $closed(s_2), \neg ab(b)$
impossible $close(s_1)$ **if** $closed(s_1)$

The information about the system's malfunctioning is given by:

$break$ **causes** $ab(b)$
$surge$ **causes** $ab(r)$
$surge$ **causes** $ab(b)$ **if** $\neg prot(b)$

This concludes our system description.

Now consider a history, Γ_0 of the system.

$$\Gamma_0 \begin{cases} hpd(close(s_1), 0) \\ obs(closed(s_1), false, 0) \\ obs(closed(s_2), false, 0) \\ obs(ab(b), false, 0) \\ obs(ab(r), false, 0) \\ obs(prot(b), true, 0) \end{cases}$$

It is easy to see that the path $\langle \sigma_0, close(s_1), \sigma_1 \rangle$ where

$$\sigma_0 = \{prot(b)\}^3$$
$$\sigma_1 = \{prot(b), closed(s_1), active(r), closed(s_2), on(b)\}$$

is the only model of Γ_0 and hence

$$\Gamma_0 \models h(on(b), 1).$$

[3]Here, for simplicity, we show only positive atoms in the description of states.

In other words, the agent expects the bulb to be lit.

To illustrate the notion of possible explanation, let us assume that now the agent observes that the bulb is not lit; i.e., its prediction differs from reality. In our terminology it means that the configuration

$$\mathcal{C} = \langle \Gamma_0, obs(on(b), false, 1) \rangle$$

is a symptom. As discussed informally in Example 10.0.1 the symptom is expected to have three possible explanations:

$$\begin{array}{l} \mathcal{E}_1 = \{occurs(surge, 0)\}, \\ \mathcal{E}_2 = \{occurs(break, 0)\}, \\ \mathcal{E}_3 = \{occurs(surge, 0), occurs(break, 0)\}. \end{array}$$

Two of them are minimal with respect to set-theoretic and cardinality-based orderings of possible explanations. Our formal analysis leads to the same conclusion. Actions *break* and *surge* are the only exogenous actions available in our language, and $\mathcal{E}_1, \mathcal{E}_2$, and $\mathcal{E}_3$ are the only sets such that $\mathcal{C} \cup \mathcal{E}_i$ is consistent (for every $1 \leq i \leq 3$). Hence $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ are the only possible explanations of the symptom. If, however, in the initial situation the bulb were not protected, $\mathcal{E}_2$ would be the only one possible explanation.

It is possible, in fact probable, that the system we would be modeling would be much more complex than the one we have here. Therefore, it is logical to assume that there may be many, possibly irrelevant, exogenous actions in our system. If we were to expand our circuit example by another exogenous action, *make_coffee*, the addition of this action alone would add three new possible explanations:

$$\begin{array}{l} \mathcal{E}_4 = \{occurs(surge, 0),\ occurs(make\_coffee, 0)\}, \\ \mathcal{E}_5 = \{occurs(break, 0),\ occurs(make\_coffee, 0)\}, \\ \mathcal{E}_6 = \{occurs(surge, 0),\ occurs(break, 0), occurs(make\_coffee, 0)\}. \end{array}$$

Notice that, although we might want an explanation that includes both *break* and *surge*, we are probably not at all interested in knowing that some other agent made coffee.[4]

It is clear that no rational agent will be interested in finding all possible explanations of a symptom. Normally different explanations can be compared with each other using various criteria. This can be represented by an ordering relation between explanations. $\mathcal{E}_1$ can be viewed to be *better*

[4]Of course, in the great scheme of things, we may not be able to dismiss this so lightly; it is possible that plugging in the coffee pot might have caused the surge in the first place.

than $\mathcal{E}_2$ if $\mathcal{E}_1$ is a proper subset of $\mathcal{E}_2$ or if the cardinality of $\mathcal{E}_1$ is less then the cardinality of $\mathcal{E}_2$ or if $\mathcal{E}_1$ contains actions which are more likely to occur or are more relevant to the symptom than those in $\mathcal{E}_2$, etc. The following definition will make this idea precise.

Definition 10.2.2. *(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 $\mathcal{E}_2$). 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 $\mathcal{E}_0 < \mathcal{E}$.*

If the partial linear order on which we base our definition of best explanation is either cardinality-based or subset-based, the best explanations for the circuit problem would be $\{occurs(break, 0)\}$ and $\{occurs(surge, 0)\}$. In some situations one may want to also allow explanation $\{occurs(break, 0), occurs(surge, 0)\}$, but certainly no "best" explanation of our symptom will contain information about occurrences of *make_coffee*. This result can be achieved by considering explanations consisting of actions "relevant" to the symptom. These actions can be defined as ones that, in some way may, directly or indirectly, cause the corresponding fluent (in our case $on(b)$) to become false. Since no law in our system description links making coffee to problems with the circuit, this action can be considered irrelevant. Our best explanations would then be $\{occurs(break, 0)\}$, $\{occurs(surge, 0)\}$ and $\{occurs(break, 0), occurs(surge, 0)\}$ since, by our definition, they are better than the ones containing action *make_coffee*.

10.3 Computing Explanations

Let us now consider a system with current configuration

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

The agent associated with the system just performed its $(n-1)^{th}$ 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.* As in the case of planning, the problem can be reduced to reasoning with answer sets. To do that we construct program **all_clear** (with parameters $\mathcal{SD}$ and $\mathcal{C}$) consisting of the encoding $\Pi(\mathcal{SD})$ of system

description $\mathcal{SD}$, configuration $\mathcal{C}$, and the following axioms:

$$holds(F,0)\ or\ \neg holds(F,0) \leftarrow fluent(inertial,F).$$
$$occurs(A,I) \leftarrow hpd(A,I).$$
$$\leftarrow obs(F,true,I), \neg holds(F,I).$$
$$\leftarrow obs(F,false,I), holds(F,I).$$

with I ranging over $[0,n]$. If program $all\_clear(\mathcal{SD},\mathcal{C})$ is consistent, then the agent's expectations are consistent with observations. Otherwise, diagnostics will be required.

Each of these axioms deserves some explanation. We'll begin by briefly introducing them and then elaborating on each one in the examples that follow. The first axiom, sometimes called the **Awareness Axiom**, guarantees that the agent takes into consideration all the fluents of the system. Recall that this is not a tautology. Under the answer set semantics the disjunction is epistemic, and the statement simply asserts that, during the construction of agent's beliefs, the value of the fluent in question cannot be unknown; i.e., any answer set of the program must contain either $holds(F,0)$ or $\neg holds(F,0)$.

The second axiom establishes the relationship between two similar relations — *occurs* and *hpd*. The latter is used to record actions that were actually observed to have happened, while the former holds even if the corresponding action is only hypothetical (as, for instance, in planning). As you can see, *occurs* is a superset of *hpd*. The rule ensures that in the process of hypothetical reasoning, we take into account actions that actually happened.

The last two rules, often called the **Reality Check Axioms**, guarantee that the agent's expectations agree with its observations. If they do not, we want the program to realize that there is an inconsistency between what it believes should logically be true versus what it has observed. Note that, despite the apparent similarity between the pair of relations *hpd* and *occurs* and that of *obs* and *holds*, the actual axioms representing these relationships are rather different. The reason for this will be discussed in Section 10.5 below.

The following proposition reduces detection of symptoms to ASP reasoning.

Proposition 10.3.1. *(Symptom Checking)*
A configuration $\mathcal{C}$ is a symptom iff program all_clear$(\mathcal{SD},\mathcal{C})$ *is inconsistent, i.e. has no answer set.*

Example 10.3.1. *(Briefcase)*
Let us again consider the briefcase domain from Section 8.5.1. One can easily check that for configuration $\mathcal{C}_0 = \langle \Gamma_0, obs(open, true, 1)\rangle$ where Γ_0 is

$$\Gamma_0 \begin{cases} obs(up(c_1), false, 0) \\ obs(up(c_2), true, 0) \\ hpd(toggle(c_1), 0) \end{cases}$$

the program $all\_clear(\mathcal{D}_{bc}, \mathcal{C}_0)$ is consistent, and hence $\mathcal{C}_0$ is not a symptom. 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.

Now let us consider the problem of computing explanations of a symptom $\mathcal{C} = \langle \Gamma_{n-1}, O^n\rangle$. Again, this problem can be reduced to computing answer sets of an ASP program. The program, called **diagnose** (with parameters $\mathcal{SD}$ and $\mathcal{C}$) consists of:

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

  $$occurs(A, I) \; or \; \neg occurs(A, I) \leftarrow 0 \leq I < n, action(exogenous, A)$$

  or, alternatively,

  $$\{occurs(A, I) : action(exogenous, A)\} \leftarrow 0 \leq I < n.$$

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

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

The following proposition reduces computation of a possible explanation of a symptom to extracting atoms formed by relation *expl* from an answer set of $diagnose(\mathcal{SD}, \mathcal{C})$.

Proposition 10.3.2. *(Computing Possible Explanations)*
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

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

Example 10.3.2. *(Diagnosing the Circuit-2)*
To illustrate the proposition let us again go back to the electrical circuit domain and history Γ_0 from Example 10.2.1. Initially both switches are open, the bulb and the resistor are normal, the bulb is protected, and the agent closes switch s_1. In Example 10.2.1 we showed that a configuration

$$\mathcal{C}=\langle \Gamma_0, obs(on(b), false, 1)\rangle$$

is a symptom with three possible explanations:

$$\begin{array}{l}\mathcal{E}_1=\{occurs(surge, 0)\},\\ \mathcal{E}_2=\{occurs(break, 0)\},\\ \mathcal{E}_3=\{occurs(surge, 0), occurs(break, 0)\}.\end{array}$$

One can easily check that, using Proposition 10.3.2, these explanations can be extracted from answer sets of program $diagnose(\mathcal{D}_{ec}, C)$. Below is the complete program, $diagnose(\mathcal{D}_{ec}, C)$; we call it `circuit.lp`.

```
%% ------------------------------------------------------
%% circuit.lp
%% ------------------------------------------------------

%% ----------
%% Signature:
%% ----------

%% Components:
comp(r).  %% relay
comp(b).  %% bulb

%% Switches:
switch(s1).
switch(s2).

%% Fluents:

fluent(inertial, prot(b)).
fluent(inertial, closed(SW)) :- switch(SW).
fluent(inertial, ab(C)) :- comp(C).
fluent(defined, active(r)).
fluent(defined, on(b)).
```

```
%% Actions:

action(agent, close(s1)).
action(exogenous, break).
action(exogenous, surge).

action(X) :- action(agent, X).
action(X) :- action(exogenous,X).
#domain action(A).

%% Steps:

#const n = 1.
step(0..n).
#domain step(I).

%% -------------------
%% System Description:
%% -------------------

%% Causal laws:

%% close(s1) causes closed(s1)
holds(closed(s1),I+1) :- occurs(close(s1),I),
                         I < n.

%% break causes ab(b)
holds(ab(b),I+1) :- occurs(break,I),
                    I < n.

%% surge causes ab(r)
holds(ab(r),I+1) :- occurs(surge,I),
                    I < n.

%% surge causes ab(b) if -prot(b)
holds(ab(b),I+1) :- occurs(surge,I),
                    -holds(prot(b),I),
                    I < n.
```

```
%% State constraints:

%% active(r) if closed(s1), -ab(r)
holds(active(r),I) :- holds(closed(s1),I),
                      -holds(ab(r),I).

%% closed(s2) if active(r)
holds(closed(s2),I) :- holds(active(r),I).

%% on(b) if closed(s2), -ab(b)
holds(on(b),I) :- holds(closed(s2),I),
                  -holds(ab(b),I).


%% Executability conditions:

%% impossible close(s1) if closed(s1)
-occurs(close(s1), I) :- holds(closed(s1),I).


%% CWA for Defined Fluents:

-holds(F,I) :- fluent(defined,F),
               not holds(F,I).

%% General 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) :- not occurs(A,I).

%% --------
%% History:
%% --------

obs(closed(s1),false,0).
obs(closed(s2),false,0).
obs(ab(b),false,0).
obs(ab(r),false,0).
obs(prot(b),true,0).

hpd(close(s1),0).

obs(on(b),false,1).

%% -------
%% Axioms:
%% -------

%% Full Awareness Axiom:
holds(F,0) | -holds(F,0) :- fluent(inertial, F).

%% Take what actually happened into account:
occurs(A,I) :- hpd(A,I).

%% Reality Check:
:- obs(F,true,I), -holds(F,I).
:- obs(F,false,I), holds(F,I).


%% ------------------------
%% Explanation Generation:
%% ------------------------

{occurs(Action,K) : action(exogenous,Action)} :- step(K),
                                                 K >= 0, K < n.


expl(A,I) :- action(exogenous,A),
```

```
            occurs(A,I),
            not hpd(A,I).

#hide.
#show expl(A,I).
```

Note that, in the presence of complete information for the initial state, it is often convenient to use a CWA to specify the inertial fluents that are false:

```
obs(F, false, 0) :- fluent(inertial,F),
                    not obs(F,true,0).
```

If we had added this line of code, then our recorded history would simply have been:

```
obs(prot(b),true,0).
hpd(close(s1),0).
obs(on(b),false,1).
```

Defined fluents are taken care of by their CWA.

As expected, invoking this program with `clingo 0 --shift circuit.lp` yields:

```
Answer: 1
expl(break,0)
Answer: 2
expl(surge,0)
Answer: 3
expl(break,0) expl(surge,0)
```

Note that the last explanation is not minimal and may or may not be viewed as desirable. At the end of the previous section we discussed an extension of our system by a new action, *make_coffee*, and showed that this substantially increased the number and length of explanations. All these explanations are found by our method. Possibly the simplest way to limit this number would be to put restrictions on the number of exogenous actions which may be unnoticed by the agent. This can be done by the addition of a simple constraint

```
%% The agent could miss at most one exogenous action.
:- action(exogenous,X1),
   action(exogenous,X2),
```

```
   X1 != X2,
   occurs(X1,I),
   not hpd(X1,I),
   occurs(X2,I1),
   not hpd(X2,I1).
```

This constraint eliminates explanation $\{expl(break, 0), expl(surge, 0)\}$ as well as all explanations containing action *make_coffee*. If the designer of the program wishes to preserve explanation $\{expl(break, 0), expl(surge, 0)\}$ but still wants to eliminate those which contain irrelevant actions, he can replace the above constraint by

```
:- action(exogenous,X),
   occurs(X,I),
   not hpd(X,I),
   not relevant(X,on(b)).

relevant(break,on(b)).
relevant(surge,on(b)).
```

where $on(b)$ is the fluent whose unexpected value we are trying to explain.

The next section describes other ways of computing best explanations by using existing extensions of ASP. These methods are very similar to those used to find minimal plans in Section 9.5.

10.4 (*) Finding Minimal Explanations

So far we only discussed the way to compute possible explanations of unexpected observations. As seen in the coffee example from Section 10.2, there are normally too many such explanations, often containing completely irrelevant actions. So it would be natural to limit ourselves to computing minimal possible explanations (with respect to cardinality, set-theoretic, or some other type of explanation ordering). Unfortunately, as with the problem of finding minimal plans, there is no known simple way to reduce this to computing answer sets of ASP programs. Therefore, as in Section 9.5 of the chapter on planning, we will rely on CR-Prolog or the `minimize` statement.

To reduce the problem of computing minimal explanations to finding answer sets of programs of CR-Prolog, we can simply replace the explanation generation rule above by cr-rule

$$occurs(A, I) \stackrel{+}{\leftarrow} \begin{array}{l} 0 \leq I < n, \\ action(exogenous, A). \end{array}$$

The rule says that an observed exogenous action could possibly have occurred in the past, but this is a rare event which should be ignored by the agent whenever possible. Accordingly, if a current configuration is not a symptom, these rules will be ignored. However, if it is a symptom, then a minimal collection of such rules will be used to restore consistency and provide the symptom's diagnosis. (Recall that minimality can refer to set-theoretic or cardinality ordering depending on the options used in the invocation of the CR-Prolog inference engine.) Try adding exogenous action *make_coffee* to `circuit.lp`, changing the explanation generation rule, and running the new program with CR-Prolog.

The `minimize` statement of Lparse which came in so handy for finding minimal plans in Section 9.5, can also be used to compute minimal explanations. Let us consider a program $min\_diagnose(\mathcal{SD}, \mathcal{C})$ obtained by expanding $diagnose(\mathcal{SD}, \mathcal{C})$ with

$$minimize\{occurs(A, S) : action(exogenous, A) : step(S)\}.$$

It is clear that the resulting program is going to compute only the diagnosis of $\mathcal{C}$ that is minimal with respect to the cardinality ordering of explanations. Try adding this `minimize` statement to the circuit program that includes exogenous action *make_coffee*. You will need to use the `--opt-all` option with `clasp` to direct it to return all possible optimizations.

10.5 Importance of New Predicates *hpd* and *obs*

The attentive reader may wonder why our language of recorded history (and consequently the program *diagnose*) uses new predicate symbols *obs* and *hpd* instead of the old *holds* and *occurs* used in the previous chapter. Of course, part of the explanation lays in their intuitive meaning. In accordance with assumptions about our agent's ability made at the beginning of this chapter, $obs(f, true, i)$ guarantees the truth of f at step i of the corresponding trajectory, while $holds(f, i)$ can also stand for expressing our hypothesis about the value of f at i. (This is exactly how *holds* was used in planning.) Similarly for *hpd* and *occurs*. One can still wonder why this distinction is important. What, if anything, will be lost if we stick with our old vocabulary?

To answer this question for *obs* and *holds*, it is enough to notice that our definition of a symptom depends on the agent's ability to distinguish between observations and hypotheses and, hence, the distinction does not look too surprising. One can still ask, however, why the Reality Check

(which is based on the distinction between observation and prediction) is not written in a way similar to the axiom for *hpd* and *occurs* as follows:

$$holds(F, I) \leftarrow obs(F, true, I). \quad (*)$$

$$\neg holds(F, I) \leftarrow obs(F, false, I). \quad (**)$$

To see the difficulty let us consider the following history of some domain

$$obs(f, true, 0)$$

$$hpd(a, 0)$$

where action a does not change the value of inertial fluent f. What would be the value of f at 1? From the observation of f, the Awareness Axiom, and the original Reality Check, we have that $holds(f, 0)$. From the Inertia Axiom, the axiom relating *hpd* and *occurs*, and the independence of f from a, we will have $holds(f, 1)$. What would happen now if we were to expand our history by

$$obs(f, false, 1)?$$

Clearly, this observation will contradict our (default) prediction $holds(f, 1)$ and hence the Reality Check will, as expected, cause a contradiction.

Let's now see how this reasoning will change if we were to replace the Reality Check by rules (*) and (**) mentioned earlier which can be viewed as expansions of the agent's definition of *holds*. Suppose there is no observation at step 1, yet f is still true at 0. As before, inertia leads us to believe $holds(f, 1)$. However, the addition of $obs(f, false, 1)$ together with the new axiom (**) allows the reasoner to defeat the Inertia Axiom and conclude $\neg holds(f, 1)$. This new power of prediction allows the agent to conclude too much — f mysteriously becomes false. This clearly contradicts our general assumption about dynamic domains from Chapter 8 which insists that change is the result of action. Therefore, we defined the Reality Check in terms of constraints and used the Awareness Axiom to resolve any possible incompleteness of the initial state.

Summary

In this chapter, we described declarative methodology for solving the basic diagnostics problem — finding possible explanations for discrepancies between what an agent expects and what it observes. As in the case of classical planning, we utilize a mathematical model which views a dynamic

domain as a transition system represented by a system description of $\mathcal{AL}$. The description contains axioms specifying normal behavior of the system together with those describing actions which can cause the system's malfunctioning. This, however, is not enough. In addition the agent must maintain the recorded history of actions and observations which describes past trajectories of the system possible from the standpoint of the agent. Given this information the agent can determine a symptom — a new observation which is not compatible with any of these trajectories. The symptom is explained by assuming the existence of past occurrences of exogenous actions which escaped the agent's attention and remain unobserved. The precise definition of a recorded history and its models, as well as the notions of symptom and possible explanation that were presented in this chapter, capture this intuition. To compute possible explanations of symptoms, we expanded our translation of a system descriptions of $\mathcal{AL}$ into logic programs by several additional axioms including the Awareness and Reality Check axioms. The new translation allowed us to introduce the basic diagnostics module which is very similar to the basic planning module discussed in the previous chapter. (Instead of hypothesizing about future actions by the agent, the diagnostics module makes hypotheses about past exogenous actions.) The chapter continues with a short discussion of ways to limit our computation to explanations which do not contain irrelevant actions. As in the case of planning, we show how this can be done in extensions of Answer Set Prolog by consistency restoring rules and by minimization constructs. The chapter ends by comments explaining several subtle points related to the language used to record the history of the domain.

The ability to find explanations of unexpected events is another important reasoning task which has been reduced to the computation of answer sets. Even though much more work is needed to fully understand diagnostics reasoning, the material presented in the chapter can already serve as the basis for declarative design of intelligent agents. It is especially important to notice that both planning and diagnostic reasoning use the same knowledge of the domain[5].

Exercises

1. (a) Give causal laws of $\mathcal{AL}$ for action $receive(X, N, P)$ — person X receives N units of produce P. Use a fluent $has(X, N, P)$ which

[5]Even though this may not be surprising to the readers of this book, historically substantially different knowledge bases were used for each task.

holds iff person X has N units of produce P.

(b) Describe a recorded history Γ_1 in which Rey, who initially has 3 apples, receives two more of them.

(c) Show logic program $all\_clear(\mathcal{SD}, \mathcal{C})$ for system description $\mathcal{SD}$ from (a) and configuration $\mathcal{C} = \langle \Gamma_1, \emptyset \rangle$.

(d) Check if $\mathcal{C}$ is a symptom.

2. You are given the following 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.

(a) Represent the story by a system description of $\mathcal{AL}$ and a recorded history.

(b) Translate this representation into ASP and check if the resulting configuration is a symptom. What knowledge does the agent have about the quality of Joanna's baking soda?

3. You are given the following story:
Arnold needs to get the book he left in his room. The room is locked and dark, but Arnold has the key and can turn on the light.

(a) Represent this domain in ASP and write a program that finds a plan that allows Arnold to get the book. Hint: Do not forget to tell the planner that one cannot go into a room if one is already in it, etc.

(b) Suppose that *after Arnold turned on the light switch, the room remained dark.* Add a history to your program that records that the steps in Arnold's plan have been executed, and that the room was observed to still be dark. Assume that there are two relevant exogenous actions, $break(switch)$ and $break(power\_line)$.

Replace your planning module with an explanation module that comes up with a minimal explanation of this unexpected observation.

4. Consider the farmers' market story from Exercise 10 in Chapter 9. Suppose that Fred, Georgina, and Harry are the first customers to arrive at the farmers' market. Fred wants to buy apples, pears, and lettuce. He knows what each of the vendors sells, so he decides to buy apples from Amy first, then lettuce from Bruce, and finally pears from

Don. However, when he arrives at Don's booth he learns that Don is out of pears. Write an ASP program that models this domain and its history and comes up with a diagnosis. Note that, since Fred is the one doing the shopping, his actions are considered agent actions, while the actions of other customers are exogenous. Natural concurrent actions should be allowed, but obvious restrictions should also apply. To limit possible answer sets, also add the assumption that two customers cannot buy from the same vendor at the same time.

5. You are given the following story:
Millie is a teenager. She felt fine when she was at school but when she came home she observed that she had a fever and a severe, coated sore throat. She knows that, in teenagers, these are symptoms of both strep throat and mononucleosis. She concludes that she has one of these diseases.

   (a) Represent Millie's medical knowledge in $\mathcal{AL}$ and her knowledge about her actions and observations as a recorded history. Use exogenous actions *infect(strep, Person)* or *infect(mono, Person)* which cause a person to have strep or mono, respectively. The symptoms of these diseases can be viewed as indirect effects of these actions.

   (b) Translate your representation into ASP, add the explanation module and rules limiting the number of unobserved exogenous actions to model Millie's reasoning.

   (c) *A day later, Millie develops jaundice, which is a symptom of mono, but not strep. She concludes that she has mononucleosis.* Add the new knowledge to you program and run it to find the new explanation. Use action *wait* to represent that Millie waited a day before observing her new symptom.

Chapter 11

Probabilistic Reasoning

The knowledge representation languages and reasoning algorithms discussed in previous chapters allowed an agent to reason with various forms of incomplete knowledge. Defaults gave agents the ability to make assumptions about completeness of their knowledge and to withdraw these assumptions if they contradicted new information. Incompleteness of the agent's knowledge was manifested by multiple answer sets, some of which contained proposition p while others contained its negation. This was the case for, say, the program consisting of the Awareness Axiom,

$$p(a) \ or \ \neg p(a)$$

for $p(a)$. Another form of incompleteness, exhibited by program

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

corresponded to the case when the agent associated with the program was simply not aware of $p(a)$. Technically, the program has one answer set containing neither $p(a)$ nor $\neg p(a)$. In both these cases, propositions could only have three truth values: *true*, *false*, and *unknown*. In this chapter we discuss an augmentation of our knowledge representation languages with constructs for representing a finer gradation of the strength of an agent's beliefs. The resulting language preserves all the power of ASP, including the ability to represent defaults, non-monotonically update the knowledge base with new information, define rules recursively, etc., while allowing an agent to do sophisticated probabilistic reasoning.

11.1 Classical Probabilistic Models

Mathematicians and computer scientists developed a number of mathematical models capturing various aspects of quantifying the beliefs of a rational agent. The oldest and most developed collection of such models can be found in Probability Theory — a branch of mathematics with a several hundred year history of development and applications. Even though Probability Theory is a well-developed branch of mathematics, the methodology of its use for knowledge representation, and even the intuitive meaning of basic notions of the theory, are still matters of contention. In this book we view probabilistic reasoning as *commonsense reasoning about the degree of an agent's beliefs in the likelihood of different events*. In what follows we clarify the meaning of this statement. Let us start with a simple example.

Example 11.1.1. *(Lost in the Jungle)*

> Imagine yourself lost in a dense jungle. A group of natives has found you and offered to help you survive, provided you can pass their test. They tell you they have an Urn of Decision from which you must choose a stone at random. (The urn is sufficiently wide for you to easily get access to every stone, but you are blindfolded so you cannot cheat.) You are told that the urn contains nine white stones and one black stone. Now you must choose a color. If the stone you draw matches the color you chose, the tribe will help you; otherwise, you can take your chances alone in the jungle. (The reasoning of the tribe is that they do not wish to help the exceptionally stupid, or the exceptionally unlucky.)

It does not take knowledge of probability theory to realize that you will have a much better chance of obtaining their help if you choose white for the color of the stone. All that is required is common sense. Your line of reasoning may be as follows: "Suppose I choose white. What would be my chances of getting help? They are the same as the chances of drawing a white stone from the urn. There are nine white stones out of a possible ten. Therefore, my chances of picking a white stone and obtaining help are $\frac{9}{10}$." The number $\frac{9}{10}$ can be viewed as the degree of belief that help will be obtained if you select white. To double-check yourself (since your life may depend on this), you might consider what would happen if you choose black. In this case, to get help you need to draw the black stone. The chances of that are one out of ten ($\frac{1}{10}$). Clearly, you should choose white.

To mathematically study this type of reasoning, one may use the classical notion of **probabilistic model**. Such models consist of

- a finite set Ω whose elements are referred to as **possible worlds**. Intuitively possible worlds correspond to possible outcomes of random experiments we attempt to perform (like drawing a stone from the urn);
- a function μ from possible worlds of Ω to the set of real numbers such that:

$$\text{for all } W \in \Omega \ \mu(W) \geq 0 \text{ and}$$

$$\sum_{W \in \Omega} \mu(W) = 1.$$

Function μ will be referred to as a **probabilistic measure**. Intuitively $\mu(W)$ quantifies the agent's degree of belief in the likelihood of the outcomes of random experiments represented by W.

A function

$$P : 2^{\Omega} \rightarrow [0, 1]$$

such that

$$P(E) = \sum_{W \in E} \mu(W)$$

for any $E \subseteq \Omega$ is called a **probability function** induced by μ; it characterizes the degree of belief in the likelihood of the actual outcome of the agent's experiment belonging to E.

Of course this simple definition only works for situations in which there is only a finite number of possible worlds. Modern probability theory generalizes this notion to the infinite case and discovers a lot of non-trivial properties of probability and other similar measures. But even the simple definition described above allows us to obtain non-trivial properties of probability and to successfully use the resulting calculus to reason about the likelihood of a particular outcome of random events.

In logic-based probability theory, possible worlds are often identified with logical interpretations, and a set E of possible worlds is often represented by a formula F such that $W \in E$ iff W is a model of F. In this case the probability function may be defined on propositions

$$P(F) =_{def} P(\{W : W \in \Omega \text{ and } W \text{ is a model of } F\}).$$

Now let us go back to our traveler saying: "suppose I choose white" and performing the mental experiment of randomly drawing a stone from the urn. To understand a mathematical model that predicts his chances of getting help, we need to come up with a collection Ω of possible worlds

which correspond to possible outcomes of this random experiment. There are, of course, many ways of doing this. We start with a fairly detailed representation which includes individual stones. To be able to refer to stones, we enumerate them by integers from 1 to 10 starting with the black stone. The possible world describing the effect of the traveler drawing stone number 1 from the urn will look like this:

$$W_1 = \{select\_color = white, draw = 1, \neg help\}.$$

Drawing the second stone will result in possible world

$$W_2 = \{select\_color = white, draw = 2, help\}$$

etc. We will have 10 possible worlds, nine of which contain *help*. To define a probabilistic measure μ on these possible worlds, we use the so-called **Principle of Indifference** — a commonsense rule which states that *possible outcomes of a random experiment are assumed to be equally probable if we have no reason to prefer one of them to any other*. This rule suggest that $\mu(W) = 0.1$ for any possible world $W \in \Omega$. According to our definition of probability function P, the probability that the outcome of the experiment contains *help* is 0.9. A similar argument for the case in which the traveler selects black gives 0.1. The probabilistic model allowed us to produce the expected result.

The reader of course noticed that the most difficult part of the above argument was setting up a probabilistic model of our domain — especially the selection of possible worlds. This observation raises a typical Computer Science question: "*How can possible worlds of a probabilistic model be found and represented?*" In what follows we discuss one of the possible ways of doing this. We will simply try to encode the knowledge presented to us in a story by a program in knowledge representation language **P-log** [16] — an extension of ASP and/or CR-Prolog that allows us to combine logical and probabilistic knowledge. Answer sets of a program of the new language will be identified with possible worlds of the domain.

11.2 The Jungle Story in P-log

We now show how to encode the jungle story in P-log. (We call the resulting program Π_{jungle}.) The syntax of P-log is rather similar to that of ASP. But, unlike standard ASP, the signature of a P-log program is sorted — each function symbol comes with sorts for its parameters and its range. Π_{jungle}

has two sorts, *stones* and *colors*, consisting of objects of our domain. The sorts will be defined by the following P-log statements:

$$stones = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.$$

$$colors = \{black, white\}.$$

For the sake of making it easy to talk about the stones, we assume that the first one is black and the others are white. (It really does not matter which one is black.) This is exactly what we did in setting up our first model. To record this we introduce a function *color* mapping stones into colors:

$$color : stones \rightarrow colors.$$

(The usual mathematical notation is used for the function declarations.) The values of the function are given by the following P-log rules.

$$\begin{array}{l} color(1) = black. \\ color(X) = white \quad \leftarrow \quad X \neq 1. \end{array}$$

Note that the only difference between rules of P-log and ASP is the form of the atoms.

To proceed, let us call the random process of selecting one of these stones from the urn *draw*. In P-log this will correspond to the following declaration:

$$\begin{array}{l} draw : stones. \\ random(draw). \end{array}$$

The first statement says that *draw* is a zero-arity function which takes its values from the sort *stones*. The second statement, referred to as a **random selection rule** states that *normally*, in a (mental or physical) experiment conducted by the reasoner, the values for *draw* are selected at random. Random processes of P-log are often referred to as **random attributes**. Finally we need to declare two zero-arity functions

$$select\_color : colors$$

and

$$help : boolean$$

where *boolean* is a predefined sort with the usual values *true* and *false*. The tribal laws will then be represented by the rules:

$$\begin{array}{lll} help & \leftarrow & draw = X, \\ & & color(X) = C, \\ & & select\_color = C. \\ \neg help & \leftarrow & draw = X, \\ & & color(X) = C, \\ & & select\_color \neq C. \end{array}$$

Here $help$ and $\neg help$ are used as shorthands for $help = true$ and $help = false$, respectively. Also, $select\_color \neq C$ is understood as shorthand for $\neg(select\_color = C)$.

This concludes the construction of our first P-log program, Π_{jungle}. It represents the traveler's knowledge about his situation. Recall that informally the traveler starts by asking himself the following question: *"Suppose I choose white. What would my chances of getting help be?"* To answer this question the reasoner first expands the program Π_{jungle} by the statement

$$select\_color = white.$$

The resulting program, $\Pi_{jungle}(white)$, contains one random attribute, $draw$.

Each possible outcome of random selection for $draw$ defines one possible world. The precise definition of possible worlds of the program will be given in the next section but intuitively one should be able to compute them. If the result of our random selection were 1, then this world would be

$$W_1 = \{draw = 1, select\_color = white, \neg help\}^1$$

(Since $color(1) = black$ and $select\_color = white$ are facts of the program, the result follows immediately from the definition of $help$.) If the result of our random selection were 2 then the world determined by this selection would be

$$W_2 = \{draw = 2, select\_color = white, help\}.$$

Similarly for stones from 3 to 10. Possible worlds defined by our program are exactly those introduced in our previous solution. The new representation, however, is more elaboration tolerant and is closer to the informal

[1] Actually this possible world also contains other atoms which do not depend on the value of $draw$ such as sorts, statements of the form $color(1) = black$, $color(2) = white$, etc., but they are the same in all the possible worlds and we will not show them in our representation.

description of the story. The semantics of P-log will use the Indifference Principle to automatically compute the probabilistic measure of every possible world and hence the probabilities of the corresponding events. Since in this case all worlds are equally plausible, the ratio of possible worlds in which arbitrary statement F is true to the number of all possible worlds will give the probability of F. Hence the probability of *help* defined by the program $\Pi_{jungle}(white)$ is $\frac{9}{10}$.

Now the reasoner will consider a program $\Pi_{jungle}(black)$ obtained from Π_{jungle} by adding the statement

$$select\_color = black.$$

A similar argument shows that probability of getting help in this situation is $\frac{1}{10}$. Clearly, the first choice is preferable.

11.3 Syntax and Semantics of P-log

In this section we use the jungle program to informally explain the syntax and semantics of P-log. Since P-log is a sorted language, its programs contain definitions of sorts and declarations of functions. The atoms of P-log are properly-typed expressions of the form $a(\bar{t}) = y$, where y is a constant from the range of a function symbol a or a variable. A term of the form $a(\bar{t})$ is called an **attribute**. The negation of an atom, $\neg(a(\bar{t}) = y)$, is written as $a(\bar{t}) \neq y$. As usual, atoms and their negations are referred to as literals. In addition to declarations and the regular ASP rules formed from P-log literals, the P-log programmer may declare some of the attributes to be random.

It is not difficult to show that program Π_{jungle}, as well as any other P-log program Π, can be translated into a regular ASP program, say $\tau(\Pi)$, with the same logical meaning. In fact the logical semantics of Π will be defined via the semantics of $\tau(\Pi)$.

Definition 11.3.1. *($\tau(\Pi)$)*
For every attribute $a(\bar{t})$ with $range(a) = \{y_1, \ldots, y_n\}$, the mapping τ

- *represents the sort information by a corresponding set of atoms; e.g.*

  $s = \{1, 2\}$ is turned into facts $s(1)$ and $s(2)$;

- *replaces every occurrence of an atom*

$$a(\bar{t}) = y$$

by

$$a(\bar{t}, y),$$

and expands the program by rules of the form

$$\neg a(\bar{t}, Y_2) \leftarrow a(\bar{t}, Y_1), Y_1 \neq Y_2;$$

- *replaces every occurrence of* $a(\bar{t}, true)$ *and* $a(\bar{t}, false)$ *by* $a(\bar{t})$ *and* $\neg a(\bar{t})$ *respectively, and removes double negation* $\neg\neg$*, which might have been introduced by this operation;*

- *replaces every rule of the form*

$$random(a(\bar{t})) \leftarrow body$$

by

$$a(\bar{t}) = y_1 \ or \ \ldots \ or \ a(\bar{t}) = y_n \leftarrow body, \ not \ intervene(a(\bar{t})) \qquad (11.1)$$

where intervene *is a new predicate symbol;*

The translation is justified by the intuitive reading of $random(a(\bar{t}))$ *which says that,* under normal circumstances, during the construction of a possible set of beliefs, the reasoner associated with the program must randomly select exactly one value of $a(\bar{t})$ from a's range.

Actually, P-log allows more-general random selection rules which have the form:

$$random(a(\bar{t}) : \{X : p(X)\}) \leftarrow body.$$

The set $\{X : p(X)\}$ *is often referred to as the* **dynamic range** *of* $a(\bar{t})$*. The rule limits the selection of the value of* $a(\bar{t})$ *to elements of* a*'s range which satisfy property* p*. For each such rule,* τ *creates an additional ASP rule:*

$$\leftarrow a(\bar{t}, Y), \ not \ p(Y), \ not \ intervene(a(\bar{t})). \qquad (11.2)$$

- *grounds the resulting program by replacing variables with elements of the corresponding sorts.*

This completes the construction of $\tau(\Pi)$*.*

Last, we would like to mention that P-log's random selection rules may be preceded by terms used as rule names. Actually, because of technical reasons, such names are required by some P-log solvers. They will not, however, be used in this book.

Definition 11.3.2. *Collections of atoms from answer sets of* $\tau(\Pi)$ *are called* **possible worlds** *of* Π. *(The new term is used to simply stay close to the traditional terminology of probability theory.)*

Declarations and rules of the form described so far form the logical part of a P-log program Π. This part defines sets of beliefs of a rational reasoner associated with Π, or, in the language of probability, possible worlds of Π.

In addition, program Π may contain so called *pr*-atoms[2] describing the likelihood of particular random events, and constructs describing observations and actions of the reasoner associated with the program. The syntax and semantics of these constructs will be defined a little later.

Meanwhile we will proceed with defining the probabilistic semantics of a P-log program. To do that we first define a probabilistic measure on its possible worlds. This measure, a real number from the interval $[0, 1]$, represents the degree of a reasoner's belief that a possible world W matches a true state of the world. Zero means that the agent believes that the possible world does not correspond to the true state; one corresponds to the certainty that it does. *The probability of a set of possible worlds is the sum of the probabilistic measures of its elements.* This definition is easily expanded to define the probability of propositions[3].

Definition 11.3.3. *The* **probability of a proposition** *is the sum of the probabilistic measures of possible worlds in which this proposition is true.*

Our next task is to explain the definition of probabilistic measure on possible worlds of a P-log program. Our jungle example shows how this can be done using the Indifference Principle for a program with one random selection rule. The following example demonstrates how the Indifference Principle is used to define probabilistic measure for programs with multiple random selection rules.

Example 11.3.1. *(Dice)*

> Mike and John each own a die. Each die is rolled once. We would like to estimate the chance that the sum of the rolls is high, i.e. greater than 6.

To do that we construct a P-log program Π_{dice} encoding our knowledge of the domain. (The program will contain some knowledge which is not

[2]Here *pr* stands for *probabilistic*.

[3]Here by proposition we mean ground atoms or formulas obtained from these atoms using connectives $\neg$, $\wedge$, and *or*.

necessary for making the desired prediction, but it will be useful for further modifications of the example.) Π_{dice} will have a signature Σ containing the sort *dice* consisting of the names of the two dice, d_1 and d_2; a (random) attribute *roll* mapping each die to the value it indicates when thrown, which is an integer from 1 to 6; an attribute *owner* mapping each die to a person; relation *high* which holds when the sum of the rolls of the two dice is greater than six. The corresponding declarations will look like this:

$$
\begin{aligned}
&dice = \{d_1, d_2\}.\\
&score = \{1, 2, 3, 4, 5, 6\}.\\
&person = \{mike, john\}.\\
&roll : dice \rightarrow score.\\
&random(roll(D)).\\
&owner : dice \rightarrow person.\\
&high : boolean.
\end{aligned}
$$

The regular part of the program will consists of the following rules:

$$
\begin{aligned}
&owner(d_1) = mike.\\
&owner(d_2) = john.\\
&high \leftarrow roll(d_1) = Y_1,\\
&\qquad\quad roll(d_2) = Y_2,\\
&\qquad\quad (Y_1 + Y_2) > 6.\\
&\neg high \leftarrow roll(d_1) = Y_1,\\
&\qquad\quad\ \ roll(d_2) = Y_2,\\
&\qquad\quad\ \ (Y_1 + Y_2) \leq 6.
\end{aligned}
$$

The translation, $\tau(\Pi_{dice})$, of our program into ASP looks like this:

$$
\begin{array}{l}
dice(d_1).\\
dice(d_2).\\
dice(d_3).\\
score(1..6).\\
person(mike).\\
person(john).\\
roll(D,1)\ or\ \ldots\ or\ roll(D,6) \leftarrow\ not\ intervene(roll(D)).\\
owner(d_1,mike).\\
owner(d_2,john).\\
high \leftarrow roll(d_1,Y_1),\\
\qquad\quad roll(d_2,Y_2),\\
\qquad\quad (Y_1+Y_2)>6.\\
\neg high \leftarrow roll(d_1,Y_1),\\
\qquad\quad\ \ roll(d_2,Y_2),\\
\qquad\quad\ \ (Y_1+Y_2)\leq 6.
\end{array}
$$

Notice that, in accordance with the definition of τ, function declarations are simply omitted.

By computing answer sets of $\tau(\Pi_{dice})$ we obtain 36 possible worlds of our program — each world corresponding to a possible selection of values for random attributes $roll(d_1)$ and $roll(d_2)$. The first selection has six possible outcomes which, by the principle of indifference, are equally likely. So is the second selection. *The mechanisms controlling the way the agent selects the values of* $roll(d_1)$ *and* $roll(d_2)$ *during the construction of his beliefs are independent from each other.* This independence justifies the definition of the probabilistic measure of a possible world containing $roll(d_1) = i$ and $roll(d_2) = j$ as the product of the agent's degrees of belief in $roll(d_1) = i$ and $roll(d_2) = j$ [4]. Hence the measure of a possible world containing $roll(d_1) = i$ and $roll(d_2) = j$ for every possible i and j will be $\frac{1}{6}\times\frac{1}{6} = \frac{1}{36}$. The probability $P_{\Pi_{dice}}(high)$ is the sum of the measures of the possible worlds which satisfy $high$. Since $high$ holds in 21 worlds, the probability $P_{\Pi_{dice}}(high)$ of $high$ being true is $\frac{7}{12}$. If the reasoner associated with Π_{dice} had to bet on the outcome of the game, betting on $high$ would be better.

[4] In probability theory two events A and B are called independent if the occurrence of one does not affect the probability of another. Mathematically, this intuition is captured by the following definition: events A and B are independent (with respect to probability function P) if $P(A \wedge B) = P(A) \times P(B)$. For example, an event of d_1 showing a 5 is independent of d_2 showing a 5 while an event of the sum of scores on both dice showing showing a 5 is dependent on an event of d_1 showing a 5.

Note that while in the jungle example only one random selection rule was used to construct a possible world of the program, in the dice example there are two such rules, which requires the application of the product rule. Otherwise, the probabilistic argument of the dice example is similar to that of the jungle example.

Suppose now that we learned from a reliable source that while the die owned by John is fair, the die owned by Mike is biased. The informer conducted multiple statistical experiments which allowed him to conclude that, on average, Mike's die will roll a 6 in 1 out of 4 rolls. To incorporate this type of knowledge in our program, we need some device allowing us to express the numeric probabilities of possible values of random attributes. This is expressed in P-log through **causal probability statements**, or *pr*-**atoms**. A *pr*-atom takes the form

$$pr_r(a(\bar{t}) = y|_c\ B) = v \tag{11.3}$$

where $a(\bar{t})$ is a random attribute, B a conjunction of literals, r is the name of a random selection rule used to generate the values of $a(\bar{t})$, $v \in [0, 1]$, and y is a possible value of $a(\bar{t})$. If the rule generating the values of $a(\bar{t})$ is uniquely determined by the program then it's name, r, can be omitted. The "causal stroke" '$|_c$' and the "rule body" B may also be omitted in case B is empty.

The statement has a rather sophisticated informal reading. First it says that *if the value of* $a(\bar{t})$ *is generated by rule* r*, and* B *holds, then the probability of the selection of* y *for the value of* $a(\bar{t})$ *is* v*.* In addition, it states the potential *existence of a direct causal relationship between* B *and the possible value of* $a(\bar{t})$. We will refer to relations expressed by causal probability statements of P-log as **causal probabilities**. In this book we will only briefly discuss causal probability and its properties. For an in-depth study of this relation, defined in terms of acyclic directed graphs and probability tables, one can consult [89]. The remarkable book contains a wealth of information about causality, probability, and their applications to various types of causal and probabilistic reasoning.

Example 11.3.2. *(Biased Dice)*
Information about the fairness of Mike's die after the bias was discovered will be expressed by *pr*-atom

$$pr(roll(D) = 6\ |_c\ owner(D) = mike) = \frac{1}{4}.$$

Let us now demonstrate how these statements can be used to define the probabilistic measures of possible worlds of the resulting program Π_{biased}.

Clearly programs Π_{dice} and Π_{biased} have the same possible worlds. First consider a possible world

$$W_1 = \{roll(d_1) = 6, roll(d_2) = 1, high = true, \dots\}$$

(Since each world is uniquely determined by choices of values for the program's random attributes, $roll(d_1)$ and $roll(d_2)$, we show only these choices and the value of $high$.) Die d_1 belongs to Mike and hence, by the causal probability statement above, the likelihood of this die showing 6 as the result of the corresponding selection is $\frac{1}{4}$. By the principle of indifference the likelihood of d_2 showing 1 is $\frac{1}{6}$. As explained before, the measure of W_1 is the product of our degrees of belief in $roll(d_1) = 6$ and $roll(d_2) = 1$. Hence the measure of W_1 is $\frac{1}{24}$. Similarly for other worlds in which $roll(d_1) = 6$. What about the measure of each world in which Mike's die shows 5 or less? Since we have no information about the likelihood of these five possible outcomes of Mike's die, by the principle of indifference we have that our degree of belief in each such outcome is $\frac{(1-\frac{1}{4})}{5} = \frac{3}{20}$. So the measure of each such world is $\frac{3}{20} \times \frac{1}{6} = \frac{1}{40}$. By summing up measures of the worlds satisfying $high$ we will get $P_{\Pi_{biased}}(high) = \frac{5}{8}$.

To better understand the intuition behind our definition of probabilistic measure, it may be useful to consider an intelligent agent associated with the program in the process of constructing possible worlds. Suppose the agent has already constructed a part V of a (not yet completely constructed) possible world W, and suppose that V satisfies the precondition of some random selection rule r defining the value of some random attribute $a(\bar{t})$. The agent can continue construction by considering a mental experiment associated with r — a random selection of the possible value of $a(\bar{t})$. If y is a possible outcome of this experiment, then the agent may continue construction by adding the atom $a(\bar{t}) = y$ to V. To define the probabilistic measure μ of the possible world W under construction, we need to know the likelihood of y being the outcome of r, which we call the causal probability of the atom $a(t) = y$ in W. This information can be obtained from a pr-atom $pr(a(\bar{t}) = y \mid_c B) = v$ of our program with B satisfied by W or computed using the principle of indifference. In the latter case we need to consider the collection R of possible outcomes of experiment r. For example if $y \in R$, there is no pr-atom assigning probability to outcomes of R, and $|R| = n$, then the causal probability of $a(\bar{t}) = y$ in W will be $\frac{1}{n}$. To compute the probabilistic measure of W, one needs to consider values $y_1, \dots, y_n$ of random attributes $a_1(\bar{t}_1), \dots, a_n(\bar{t}_n)$ which determine a possible world W. The **unnormalized probabilistic measure**, $\hat{\mu}$ of W, will be defined as

the product of causal probabilities of atoms $a_1(\bar{t}_1), \ldots, a_n(\bar{t}_n)$ in W. The probabilistic measure $\mu(W)$ of a possible world W is the unnormalized probabilistic measure of W divided by the sum of the unnormalized probabilistic measures of all possible worlds of Π, i.e.,

$$\mu(W) =_{def} \frac{\hat{\mu}(W)}{\sum_{W_i \in \Omega} \hat{\mu}(W_i)}$$

For this definition to be correct program, Π must satisfy a number of natural conditions. There should be possible worlds; i.e., program $\tau(\Pi)$ should be consistent. In the process of construction of a possible world, a random attribute $a(\bar{t})$ should be defined by a unique random selection rule. There are a few others. It is not difficult to show that if these conditions are satisfied then the function μ defined above is indeed a probabilistic measure.

Finally, we introduce the two remaining syntactic constructs of P-log — statements recording observations of results of random experiments

$$obs(a(\bar{t}) = y) \tag{11.4}$$

$$obs(a(\bar{t}) \neq y) \tag{11.5}$$

and deliberate interventions in these experiments setting the value of the corresponding attribute to some given y,

$$do(a(\bar{t}) = y). \tag{11.6}$$

Here $a(\bar{t})$ is a random attribute, and y is a possible value of $a(\bar{t})$. For instance, statement $obs(roll(d_1) = 6)$ says that the random experiment consisting in rolling the first die shows 6; $do(roll(d_1) = 6)$ says that, instead of throwing the die at random, it was deliberately put on the table showing 6. To give formal semantics of these statements we need to expand our translation $\tau(\Pi)$. This is done as follows. First, for every atom of the form (11.4), and (11.5) and (11.6), we write, respectively:

$$obs(a(\bar{t}, y))$$

$$\neg obs(a(\bar{t}, y))$$

$$do(a(\bar{t}, y)).$$

Next, we add the following rules:

$$\leftarrow obs(a(\bar{t}, y)), \neg a(\bar{t}, y) \tag{11.7}$$

$$\leftarrow \neg obs(a(\bar{t}, y)), a(\bar{t}, y) \tag{11.8}$$

$$a(\bar{t}, y) \leftarrow do(a(\bar{t}, y)) \tag{11.9}$$

$$intervene(a(\bar{t})) \leftarrow do(a(\bar{t}, y)) \tag{11.10}$$

The first two rules eliminate possible worlds of the program failing to satisfy the observation. The third rule makes sure that interventions affect their intervened-upon variables in the expected way. The fourth rule defines the relation *intervene* which, for each intervention, cancels the randomness of the corresponding attribute (see rule 11.1 above). This completes the definition of $\tau(\Pi)$ for programs with observations and deliberate interventions.

As before, possible worlds of Π are defined as collections of atoms from answer sets of $\tau(\Pi)$. The definition of probability does not change either, but one should pay careful attention to the process of normalization and possible change of an attribute from random to deterministic.

We illustrate the semantics of the new statements by revisiting our dice example.

Example 11.3.3. *(Dice Revisited)*
Consider a P-log program from Example 11.3.1 expanded by an observation that John rolled a 3:

$$obs(roll(d_2) = 3).$$

It is not difficult to see that the resulting program $\Pi_{dice} \cup \{obs(roll(d_2) = 3)\}$, has six possible worlds obtained from possible worlds of Π_{dice} by removing those containing $roll(d_2) = y$ where $y \neq 3$. For each such world W, the unnormalized probabilistic measure is

$$\hat{\mu}(W) = \frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$$

After normalization we have

$$\mu(W) = \frac{1/36}{6/36} = \frac{1}{6}$$

and hence, since $high$ is true in 3 worlds, we have

$$P_{\Pi_{dice}}(high) = \frac{3}{6} = \frac{1}{2}$$

(We hope the reader noticed the importance of normalization.)

Conditioning on observations has been extensively studied in classical probability theory. If P is a probabilistic measure, then the conditional

probability $P(A|B)$ is defined as $P(A \wedge B)/P(B)$, provided $P(B)$ is not 0. Intuitively, $P(A|B)$ is understood as the probability of a formula A with respect to the background theory and a set B of all of the agent's additional observations of the world. The new evidence B simply eliminates the possible worlds which do not satisfy B. It can be shown that, under reasonable conditions on program Π, the same formula can be used to compute the corresponding P-log probability, i.e. we have

$$P_{\Pi}(A|B) = P_{\Pi \cup obs(B)}(A)$$

Let us now consider program $\Pi_{dice} \cup \{do(roll(d_2) = 3)\}$ which is identical to $\Pi_{dice} \cup \{obs(roll(d_2) = 3)\}$ except that the observation is replaced by the deliberate action of putting John's die on the table showing three dots. Possible worlds of the new program will be the same as that of $\Pi_{dice} \cup \{obs(roll(d_2) = 3)\}$ but now, by rule (11.10), $intervene(roll(d_2))$ will be true and hence attribute $roll(d_2)$ will not be selected at random by rule (11.1). Instead it will be deterministically assigned value 3, and will not be used in the definition of probabilistic measure. For each such world W normalized probabilistic measure

$$\mu(W) = \frac{1}{6}$$

and hence

$$P_{\Pi_{dice} \cup \{do(roll(d_2)=3)\}}(high) = \frac{1}{2}.$$

Even though the computations of the corresponding probabilities were different, they led to the same results. This is not always the case. The substantial differences between the results of updating your knowledge base by observations and interventions will be discussed in later examples.

There is one more concept left to demonstrate and that is the effect of the dynamic range on possible worlds. When an attribute has a dynamic range, it means that the sample from which its values are drawn can change with time. For example, if we draw a card from a deck and then draw another card without replacing the first, our sample has changed.

Example 11.3.4. *(Aces in Succession)*
Suppose we are interested in the probability that two consecutive draws from a deck of 52 cards result in both cards being aces. We begin by representing the relevant knowledge. One possible way to do that is to enumerate our cards

$$card = \{1 \ldots 52\}.$$

Without loss of generality we can assume that the first four of them are aces.

$$ace = \{1, 2, 3, 4\}.$$

We also need two tries and random attribute *draw* that maps each try into the card selected by this draw:

$$try = \{1, 2\}.$$
$$draw : try \rightarrow card.$$

If we were to simply draw a card from the deck and then put it back, we could define *draw* as random ($random(draw(T))$). However, since the card we drew is not returned to the deck, the second selection will be made from a smaller deck. To encode this we use a dynamic range and define the randomness of *draw* as follows:

$$draw(T) : \{C : available(C, T)\}.$$

We define $available(C, T)$ to be the set of cards without the one we drew in the previous try:

$$\begin{array}{lcl} available(C, 1) & \leftarrow & card(C). \\ available(C, T+1) & \leftarrow & available(C, T), \\ & & draw(T) \neq C. \end{array}$$

Finally we'll add

$$\begin{array}{lcl} two\_aces & \leftarrow & draw(1) = Y1, \\ & & draw(2) = Y2, \\ & & 1 \leq Y1 \leq 4, \\ & & 1 \leq Y2 \leq 4. \end{array}$$

Note that because of the dynamic range of our selection the two cards chosen by the two draws can not be the same, possible worlds of the program are of the form

$$W_k = \{draw(1) = c_1, draw(2) = c_2, \dots\}$$

where $c_1 \neq c_2$. There are 52 possible outcomes of the first draw and 51 possible outcomes of the second. Thus, the program has 52×51 possible worlds — each world corresponding to outcomes of the two successive draws. By the principle of indifference the unnormalized probabilistic measure of each such world is equal to $\frac{1}{52} \times \frac{1}{51} = \frac{1}{2652}$. One can easily check that, as expected, the non-normalized measure is equal to the normalized one. To

compute the number of possible worlds containing *two_aces*, let us notice that the number of aces which may be selected by the first draw is 4 while the number of aces which may be selected by the second draw is 3. Thus, there are 12 possible worlds containing *two_aces*, and the probability of this event is $12 \times \frac{1}{2652} = \frac{12}{2652} = \frac{1}{221}$.

11.4 Representing Knowledge in P-log

The previous examples addressed rather simple probabilistic problems, which could be easily solved without building a knowledge base of the corresponding domain. In this section we will consider several examples where the use of P-log allows to substantially clarify the modeling process. The first example, known as the Monty Hall Problem, is a non-trivial puzzle which is frequently solved incorrectly even by people with some knowledge of probability theory. Moreover, it has caused a substantial amount of debate about the correctness of given solutions.

11.4.1 The Monty Hall Problem

The Monty Hall Problem gets its name from the TV game show hosted by Monty Hall.

> A player is given the opportunity to select one of three closed doors, behind one of which there is a prize. Behind the other two doors are empty rooms. Once the player has made a selection, Monty is obligated to open one of the remaining closed doors which does not contain the prize, showing that the room behind it is empty. He then asks the player if he would like to switch his selection to the other unopened door, or stay with his original choice. Here is the problem: does it matter if he switches?

The answer is YES. In fact switching doubles the player's chance to win. This problem is quite interesting, because the answer is felt by many people — often including mathematicians — to be counter-intuitive. These people almost immediately come up with a (wrong) negative answer and are not easily persuaded that they made a mistake. We believe that part of the reason for the difficulty is some disconnect between modeling probabilistic and non-probabilistic knowledge about the problem. In P-log this disconnect disappears which leads to a natural, correct, and explainable solution. In other words, the standard probability formalisms lack the ability to explicitly

represent certain non-probabilistic knowledge that is needed in solving this problem. In the absence of this explicit knowledge, wrong conclusions are made. This example is meant to show how P-log can be used to avoid this problem by allowing us to specify relevant knowledge explicitly. Technically this is done by using a random attribute *open* with the dynamic range defined by regular logic programming rules.

The domain contains the set of three doors and three zero-arity attributes: *selected*, *open* and *prize*. This will be represented by the following P-log declarations (the numbers are not part of the declaration; we number statements so that we can refer back to them):

(1) $doors = \{1, 2, 3\}$.
(2) $open, selected, prize : doors$.

The regular part contains rules that state that Monty can open any door to a room which is not selected by the player and which does not contain the prize.

(3) $\neg can\_open(D) \leftarrow selected = D$.
(4) $\neg can\_open(D) \leftarrow prize = D$.
(5) $can\_open(D) \leftarrow not\ \neg can\_open(D)$.

The first two rules are self-explanatory. The last rule, which uses both classical and default negations, is a typical ASP representation of the closed world assumption — Monty can open any door except those which are explicitly prohibited.

Assuming that both, Monty and the player act randomly the probabilistic information about the three attributes of doors can be now expressed as follows:

(6) $random(prize)$.
(7) $random(selected)$.
(8) $random(open : \{X : can\_open(X)\})$.

Notice that rule (8) guarantees that Monty selects only those doors which can be opened according to rules (3)–(5). The knowledge expressed by these rules (which can be extracted from the specification of the problem) is often not explicitly represented in probabilistic formalisms leading reasoners (who usually do not realize this) to insist that their wrong answer is actually correct.

P-log program Π_{monty0} consisting of logical rules (1)–(8) represents our general knowledge of the problem domain. To proceed with our particular story, let us encode the results of random events which occurred before the player was asked if he wanted to change his selection. Without loss of

generality we can assume that the player has already selected door 1 and that Monty opened door 2 revealing that it did not contain the prize. This is recorded by the following statements:

$$obs(selected = 1).\ \ obs(open = 2).\ \ obs(prize \neq 2).$$

Let us denote the resulting P-log program by Π_{monty1}. The program contains complete knowledge of the player relevant to his behavior in the game. To make an informed decision about switching, the player should compute the probability of the prize being behind door 1 and the prize being behind door 3. To do that, he must consider the possible worlds of Π_{monty1} and find their measures. Once the measures are found, he must sum up the measures of the worlds in with the prize is behind door 1 and do the same for worlds where the prize is behind door 3.

Because of the observations, Π_{monty1} has two possible worlds:

$$W_1 = \{selected = 1, prize = 1, open = 2, can\_open(2), can\_open(3)\}.$$

$$W_2 = \{selected = 1, prize = 3, open = 2, can\_open(2)\}.$$

In W_1 the player would lose if he switched; in W_2 he would win. Note that the possible worlds contain information not only about where the prize is, but which doors Monty can open. This is the key to correct calculation!

Now the player is ready to compute the probabilistic measures of W_1 and W_2. To do that he needs to compute the likelihood of random events within each world. In this case, they are which door is selected, where the prize is, and which door is opened by Monty. By the principle of indifference the causal probability of selecting particular values of *prize* and *selected* during the construction of both possible worlds is $\frac{1}{3}$ each. The situation, however, is different for the random attribute *open*. Recall that Monty can only open a door satisfying relation *can_open*. There are two such doors in W_1, hence the causal probability of selecting the second door in W_1 is $\frac{1}{2}$. But since Monty can open only one door in W_2, the causal probability of his choosing door 2 in W_2 is 1. The probabilistic measure of a possible world is the product of likelihoods of the random events it is comprised of. It follows that

$$\hat{\mu}(W_1) = \tfrac{1}{3} \times \tfrac{1}{3} \times \tfrac{1}{2} = \tfrac{1}{18}$$

$$\hat{\mu}(W_2) = \tfrac{1}{3} \times \tfrac{1}{3} \times 1 = \tfrac{1}{9}$$

Normalization gives us:

$$\mu(W_1) = \frac{1/18}{1/18 + 1/9} = \frac{1}{3}$$

$$\mu(W_2) = \frac{1/9}{1/18 + 1/9} = \frac{2}{3}.$$

Finally, since $prize = 1$ is true in only W_1,

$$P_{\Pi_{monty1}}(prize = 1) = \mu(W_1) = \frac{1}{3}.$$

Similarly for $prize = 3$:

$$P_{\Pi_{monty1}}(prize = 3) = \mu(W_2) = \frac{2}{3}.$$

Changing doors doubles the player's chance to win.

Now consider a situation when the player assumes (either consciously or without consciously realizing it) that Monty could have opened any one of the doors not selected by the player (including the one that contains the prize). Then the corresponding program will have a new definition of *can_open*. Rules (3)–(5) will be replaced by

$$\neg can\_open(D) \leftarrow \ selected = D.$$
$$can\_open(D) \leftarrow \ not\ \neg can\_open(D).$$

The resulting program Π_{monty2} will also have two possible worlds containing $prize = 1$ and $prize = 3$ respectively, each with unnormalized probabilistic measure of $\frac{1}{18}$ and, therefore, $P_{\Pi_{monty2}}(prize = 1) = \frac{1}{2}$ and $P_{\Pi_{monty2}}(prize = 3) = \frac{1}{2}$. In that case, changing the door will not increase the probability of getting the prize. Since the rules are explicitly written, it makes the discussion about correctness of the answer much easier.

Program Π_{monty1} has no explicit probabilistic information and so the possible results of each random selection are assumed to be equally likely. If we learn, for example, that given a choice between opening doors 2 and 3, Monty opens door 2 four times out of five, we can incorporate this information by the following statement:

$$(9) \quad pr(open = 2 \mid_c can\_open(2), can\_open(3)) = \frac{4}{5}$$

A computation similar to the one above shows that changing doors still increases the player's chances of winning. (In fact changing doors is advisable as long as each of the available doors can be opened with some positive probability.)

The next example demonstrates the causal character of probabilistic statements of P-log.

11.4.2 Death of a Rat?

Consider the following program Π_{rat} representing knowledge about whether a certain rat will eat arsenic today, and whether it will die today.

$$
\begin{array}{l}
arsenic, death : boolean. \\
random(arsenic). \\
random(death). \\
pr(arsenic) = 0.4. \\
pr(death \mid_c arsenic) = 0.8. \\
pr(death \mid_c \neg arsenic) = 0.01.
\end{array}
$$

The above program tells us that the rat eating arsenic and the rat dying are viewed as random events and that the rat is more likely to die if it eats arsenic. Not only that, the intuitive semantics of the *pr*-atoms expresses that the rat's consumption of arsenic carries information about the cause of its death (as opposed to, say, the rat's death being informative about the causes of its eating arsenic).

An intuitive consequence of this reading is that seeing the rat die raises our suspicion that it has eaten arsenic, while killing the rat (say, with a pistol) does not affect our degree of belief that arsenic has been consumed. The following computations show that this is reflected in the probabilities computed under our semantics.

The possible worlds of the above program, with their unnormalized probabilistic measures, are as follows (we show only *arsenic* and *death* literals):

$$
\begin{array}{lll}
W_1: & \{arsenic, death\}. & \hat{\mu}(W_1) = 0.4 \times 0.8 = 0.32 \\
W_2: & \{arsenic, \neg death\}. & \hat{\mu}(W_2) = 0.4 \times 0.2 = 0.08 \\
W_3: & \{\neg arsenic, death\}. & \hat{\mu}(W_3) = 0.6 \times 0.01 = 0.006 \\
W_4: & \{\neg arsenic, \neg death\}. & \hat{\mu}(W_4) = 0.6 \times 0.99 = 0.594
\end{array}
$$

Since the unnormalized probabilistic measures add up to 1, they are the same as the normalized measures. Hence,

$$P_{\Pi_{rat}}(arsenic) = \mu(W_1) + \mu(W_2) = 0.32 + 0.08 = 0.4$$

To compute the probability of $arsenic$ after the observation of *death*, we consider the program created from Π_{rat} and the statement $obs(death)$. The resulting program has two possible worlds, W_1 and W_3, with unnormalized probabilistic measures as above. Normalization yields

$$P_{\Pi_{rat} \cup \{obs(death)\}}(arsenic) = \frac{0.32}{0.32 + 0.006} = 0.982$$

Notice that the observation of death raised our degree of belief that the rat had eaten arsenic.

To compute the effect of deliberately killing the rat on the agent's belief in *arsenic*, we augment the original program with the literal *do(death)*. The resulting program, has two answer sets, W_1 and W_3. However, the action of deliberate kiling defeats the randomness of death so that W_1 has unnormalized probabilistic measure 0.4 and W_3 has unnormalized probabilistic measure 0.6. These sum up to one so the measures are also 0.4 and 0.6 respectively, and we get

$$P_{\Pi_{rat} \cup \{do(death)\}}(arsenic) = 0.4$$

Note that this is identical to the initial probability $P_{\Pi_{rat}}(arsenic)$ computed above.

The example shows that, in contrast to the case when the effect was passively observed, deliberately bringing about the effect did not change our degree of belief about the propositions relevant to its cause. Propositions relevant to a cause, on the other hand, give equal evidence for the attendant effects whether they are forced to happen or passively observed. For example, if we feed the rat arsenic, this increases its chance of death, just as if we had observed the rat eating the arsenic on its own. The conditional probabilities computed under our semantics bear this out. Similarly to the above, we can compute

$$P_{\Pi_{rat}}(death) = 0.38$$

$$P_{\Pi_{rat} \cup \{do(arsenic)\}}(death) = 0.8$$

$$P_{\Pi_{rat} \cup \{obs(arsenic)\}}(death) = 0.8$$

11.4.3 The Spider Bite

Here is another example of subtle probabilistic reasoning which can be done in P-log with the help of interventions. The story we would like to formalize is as follows:

> In Stan's home town there are two kinds of poisonous spider — the creeper and the spinner. Bites from the two are equally common in Stan's area, though spinner bites are more common on

a worldwide basis. An experimental anti-venom has been developed to treat bites from either kind of spider, but its effectiveness is questionable.

One morning Stan wakes to find he has a bite on his ankle, and drives to the emergency room. A doctor examines the bite, and concludes it is a bite from either a creeper or a spinner. In deciding whether to administer the anti-venom, the doctor examines the data he has on bites from the two kinds of spiders: out of 416 people bitten by the creeper worldwide, 312 received the anti-venom and 104 did not. Among those who received the anti-venom, 187 survived; while 73 survived who did not receive anti-venom. The spinner is more deadly and tends to inhabit areas where the treatment is less available. Of 924 people bitten by the spinner, 168 received the anti-venom, 34 of whom survived. Of the 756 spinner bite victims who did not receive the experimental treatment, only 227 survived. Should Stan take the anti-venom treatment?

Let us formalize relevant parts of the story in a P-log program Π_{spider} from the point of view of the doctor. We use boolean attributes $survive$ — a random patient survived, $anti-venom$ — a random patient was administered anti-venom, and attribute $spider$ where $spider = creeper$ indicates that a random person was bitten by $creeper$. Similarly for $spinner$. From the standpoint of the doctor all three attributes (including $antivenom$) are random. Hence we have:

$$survive, antivenom \ : \ boolean.$$
$$spider \ : \ \{creeper, spinner\}.$$

$$random(spider).$$
$$random(survive).$$
$$random(antivenom).$$

Since bites from the two spiders are equally common in the area the doctor assumes that

$$pr(spider = creeper) = 0.5.$$

Statistical information available in the story allows the doctor to estimate the corresponding probabilities:

$$pr(antivenom|spider = creeper) = 312/416 = 0.75$$
$$pr(antivenom|spider = spinner) = 168/924 = 0.18$$
$$pr(survive \mid_c spider = creeper, antivenom) = 187/312 = 0.6.$$
$$pr(survive \mid_c spider = creeper, \neg antivenom) = 73/104 = 0.7.$$
$$pr(survive \mid_c spider = spinner, antivenom) = 34/168 = 0.2.$$
$$pr(survive \mid_c spider = spinner, \neg antivenom) = 227/756 = 0.3.$$

How should doctor decide if giving the anti-venom to Stan is beneficial? A natural answer is to compare the result of the deliberate action of taking the anti-venom on Stan's chance of survival with that of not taking the anti-venom. To do that let us first compute $P_{\Pi_{spider} \cup \{do(antivenom)\}}(survive)$. The program $\Pi_{spider} \cup \{do(antivenom)\}$ has the following possible worlds:

$$W_1 = \{spider = creeper, antivenom, survive\}$$
$$W_2 = \{spider = creeper, antivenom, \neg survive\}$$
$$W_3 = \{spider = spinner, antivenom, survive\}$$
$$W_4 = \{spider = spinner, antivenom, \neg survive\}$$

To compute the probabilistic measures of the possible worlds let us first notice that, because of the intervention, each possible world contains two random attributes, *spider* and *survive*. Accordingly,

$$\mu(W_1) = 0.5 \times 0.6 = 0.3$$
$$\mu(W_2) = 0.5 \times 0.4 = 0.2$$
$$\mu(W_3) = 0.5 \times 0.2 = 0.1$$
$$\mu(W_4) = 0.5 \times 0.8 = 0.4$$

(Note that, since the measures sum up to one, unnormalized and normalized measures coincide.)

It follows that the probability of survival after the treatment is

$$P_{\Pi_{spider} \cup \{do(antivenom)\}}(survive) = 0.4$$

Now let us compute $P_{\Pi_{spider} \cup \{do(\neg antivenom)\}}(survive)$. Possible worlds of this program are

$$W_5 = \{spider = creeper, \neg antivenom, survive\}$$
$$W_6 = \{spider = creeper, \neg antivenom, \neg survive\}$$
$$W_7 = \{spider = spinner, \neg antivenom, survive\}$$
$$W_8 = \{spider = spinner, \neg antivenom, \neg survive\}$$

The measures are

$$\mu(W_5) = 0.5 \times 0.7 = 0.35$$
$$\mu(W_6) = 0.5 \times 0.3 = 0.15$$
$$\mu(W_7) = 0.5 \times 0.3 = 0.15$$
$$\mu(W_8) = 0.5 \times 0.7 = 0.35$$

and the probability of survival without the treatment is

$$P_{\Pi_{spider} \cup \{do(\neg antivenom)\}}(survive) = 0.5$$

Thus, to maximize Stan's chance of survival, it is better not to administer the anti-venom.

But what would happen if, instead of conditioning on intervention, the doctor decided to use what he believes to be a more traditional approach and condition on observations, i.e. compare $P_{\Pi_{spider} \cup \{obs(antivenom)\}}(survive)$ and $P_{\Pi_{spider} \cup \{obs(\neg antivenom)\}}(survive)$? The program $P_{\Pi_{spider} \cup \{obs(antivenom)\}}$ would have the possible worlds $W_1, \ldots W_4$ defined above, but the measures would be different. In this case, *antivenom* would be a random attribute, and its likelihood should be taken into account.

The unnormalized probabilistic measures induced by $P_{\Pi_{spider} \cup \{obs(antivenom)\}}$ are

$$\hat{\mu}(W_1) = 0.5 \times 0.75 \times 0.6 = 0.225$$
$$\hat{\mu}(W_2) = 0.5 \times 0.75 \times 0.4 = 0.15$$
$$\hat{\mu}(W_3) = 0.5 \times 0.18 \times 0.2 = 0.018$$
$$\hat{\mu}(W_4) = 0.5 \times 0.18 \times 0.8 = 0.072$$

measures are

$$\mu(W_1) = 0.225/0.465 = 0.484$$
$$\mu(W_2) = 0.15/0.465 = 0.322$$
$$\mu(W_3) = 0.18/0.465 = 0.039$$
$$\mu(W_4) = 0.8/0.465 = 0.155$$

and probability

$$P_{\Pi_{spider} \cup \{obs(antivenom)\}}(survive) = \mu(W_1) + \mu(W_3) = 0.523$$

Similarly, we can compute $P_{\Pi_{spider} \cup \{obs(\neg antivenom)\}}(survive)$

$$\hat{\mu}(W_5) = 0.5 \times 0.25 \times 0.7 = 0.088$$
$$\hat{\mu}(W_6) = 0.5 \times 0.25 \times 0.3 = 0.038$$
$$\hat{\mu}(W_7) = 0.5 \times 0.82 \times 0.3 = 0.123$$
$$\hat{\mu}(W_8) = 0.5 \times 0.82 \times 0.7 = 0.287$$

$P_{\Pi_{spider} \cup \{obs(\neg antivenom)\}}(survive) = \mu(W_5) + \mu(W_7) = 0.164 + 0.229 = 0.393$

The different computation suggests that taking anti-venom would increase Stan's chance of survival, i.e. it would lead to the different (and wrong) conclusion. We hope that this example helps to see how conditioning on interventions — a possibility not available in classical probability — helps to avoid an unpleasant and costly mistake caused by confusion between observations and interventions.

11.4.4 The Bayesian Squirrel

In this section we consider an example from [57] used to illustrate the notion of Bayesian learning. One common type of learning problem consists of selecting from a set of models of a random phenomenon by observing repeated occurrences of the phenomenon. The Bayesian approach to this problem is to begin with a "prior density" on the set of candidate models and update it in light of our observations.

As an example, Hilborn and Mangel describe the Bayesian squirrel. The squirrel has hidden its acorns in one of two patches, say Patch 1 and Patch 2, but can not remember which. The squirrel is 80% certain the food is hidden in Patch 1. Also, it knows there is a 20% chance of finding food per

day when it is looking in the right patch (and, of course, a 0 chance if it's looking in the wrong patch).

To represent this knowledge in P-log program Π_{sq} we introduce sorts

$$patch = \{p1, p2\}.$$
$$day = \{1 \dots n\}.$$

(where n is some constant, say, 5) and attributes

$$hidden\_in : patch.$$
$$found : patch \times day \rightarrow boolean.$$
$$look : day \rightarrow patch.$$

Attribute *hidden_in* is always random. Hence we include

$$random(hidden\_in).$$

Attribute $found$, however, is random only if the squirrel is looking for food in the right patch, i.e. we have

$$random(found(P, D)) \leftarrow \quad hidden\_in = P,$$
$$look(D) = P.$$

The value of attribute $look(D)$ is decided by the squirrel's deliberation. Hence this attribute is not random.
The regular part of the program consists of the closed world assumption for $found$:

$$\neg found(P, D) \leftarrow \ not\ found(P, D).$$

Probabilistic information of the story is given by statements:

$$pr(hidden\_in = p1) = 0.8.$$
$$pr(found(P, D)) = 0.2.$$

This knowledge, in conjunction with the description of the squirrel's activity, can be used to compute probabilities of possible outcomes of the next search for food.

Consider for instance program $\Pi_{sq1} = \Pi_{sq} \cup \{look(1) = p_1\}$ which represents that the squirrel decided to look for food in Patch 1 on the first day. The program has three possible worlds

$$W_1^1 = \{look(1) = p_1, hidden\_in = p_1, found(p_1, 1), \dots\}$$
$$W_2^1 = \{look(1) = p_1, hidden\_in = p_1, \neg found(p_1, 1), \dots\}$$
$$W_3^1 = \{look(1) = p_1, hidden\_in = p_2, \neg found(p_1, 1), \dots\}$$

with probabilistic measures

$$\mu(W_1^1) = 0.16$$
$$\mu(W_2^1) = 0.64$$
$$\mu(W_3^1) = 0.2$$

As expected

$$P_{\Pi_{sq1}}(hidden\_in = p_1) = 0.16 + 0.64 = 0.8$$

and

$$P_{\Pi_{sq1}}(found(p_1, 1)) = 0.16.$$

Suppose now that the squirrel failed to find its food during the first day, and decided to continue its search in the first patch the next morning. Failure to find food on the first day should decrease the squirrel's degree of belief that the food is hidden in patch one, and consequently decrease its degree of belief that it will find food by looking in the first patch again. This is reflected in the following computation:

Let $\Pi_{sq2} = \Pi_{sq1} \cup \{obs(\neg found(p_1, 1)), look(2) = p_1\}$.

The possible worlds of Π_{sq2} are:

$$W_1^2 = \{look(1) = p_1, \neg found(p_1, 1), hidden\_in = p_1, look(2) = p_1, found(p_1, 2) \dots\}$$
$$W_2^2 = \{look(1) = p_1, \neg found(p_1, 1), hidden\_in = p_1, look(2) = p_1, \neg found(p_1, 2) \dots\}$$
$$W_3^2 = \{look(1) = p_1, \neg found(p_1, 1), hidden\_in = p_2, look(2) = p_1, \neg found(p_1, 2) \dots\}$$

Their probabilistic measures are

$$\mu(W_1^2) = .128/.84 = .152$$
$$\mu(W_2^2) = .512/.84 = .61$$
$$\mu(W_3^2) = .2/.84 = .238$$

Consequently,

$$P_{\Pi_{sq2}}(hidden\_in = p_1) = 0.762$$

and

$$P_{\Pi_{sq2}}(found(p_1, 2)) = 0.152$$

and so on.

After a number of unsuccessful attempts to find food in the first patch, the squirrel can come to the conclusion that food is probably hidden in the second patch and change its search strategy accordingly. Notice that each new experiment changes the squirrel's probabilistic model in a non-monotonic way. That is, the set of possible worlds resulting from each successive experiment is not merely a subset of the possible worlds of the previous model. The program, however, is changed only by the addition of new actions and observations. Distinctive features of P-log such as the ability to represent observations and actions, as well as conditional randomness, play an important role in allowing the squirrel to learn new probabilistic models from experience.

11.5 (*) P-log + CR-Prolog and the Wandering Robot

There is no reason why P-log must be limited to standard ASP. In fact, its semantics are naturally defined on top of CR-Prolog with the only difference being that possible worlds correspond to CR-Prolog answer sets instead of ASP ones. In this section we give an example of a P-log program with a cr-rule.

> A robot is moving around in three rooms, say $r1, r2, r3$. These rooms have doors that can be open or closed. The robot works pretty well, but in a rare case it might have a malfunction and end up in any of the three rooms that are open. If it does malfunction, the probability of the robot ending up in the intended room is $\frac{1}{2}$.

The formalization requires the initial and final moments of time

$$time = \{0, 1\}$$

and three rooms

$$room = \{r_0, r_1, r_2\}.$$

There are two attributes we will refer to as actions:

$$enter : room \rightarrow boolean.$$

$$break : boolean.$$

A robot that performs action $enter(R)$ *attempts* to enter room R and succeeds unless exogenous action *break* occurs at time 0; *break* corresponds to robot's malfunctioning. Since the value of *enter* is determined by the deliberating robot the attribute is not random. We know that *break* normally does not happen and have no information on the likelyhood of this rare event *break* is not random either.

Next we need some attributes that describe the domain's state. Attribute *open* refers to a room's door and attribute *in* is true if the robot is in a room at a given point in time.

$$open : room \rightarrow boolean.$$

$$in : time \rightarrow room.$$

The values of *open* are assumed to be given, hence it is not random; *in* is random only if the robot is malfunctioning.

The following rules govern changes in the domain:

1. If no malfunctioning occurs the robot gets to where it is going:

$$in(1) = R \leftarrow enter(R), \neg break.$$

2. A malfunction can cause the robot to end up in any of the rooms:

$$random(in(1) : \{X : open(X)\}) \leftarrow enter(R),\ break.$$

3. If there is a malfunction, the probability of the robot winding up in the room it intended to enter is $\frac{1}{2}$:

$$pr(in(1) = R \mid_c enter(R), break) = 1/2$$

4. For simplicity we assume that, if not otherwise specified, doors are open in the initial situation:

$$open(R) \leftarrow\ not\ \neg open(R).$$

5. We assume that normally the robor does not malfunction:

$$\neg break \leftarrow\ not\ break$$

However, if no other consistent state of the world exists, we must assume that the robot malfunctioned:

$$break \stackrel{+}{\leftarrow}.$$

(For the sake of brevity, we assume that the program will be used in conjunction with collections of atoms that make sense. For example, we do not allow a robot to be in two rooms at once, to try to enter more than one room at a time, to try to enter a room whose door is closed, and to start out in a closed room.)

Let's consider some inputs and the corresponding possible worlds of the program. Let's add the line

$$enter(0)$$

to our program. In this case, there is a unique possible world which contains $\neg break$ and $in(1) = r_0$. Since we do not consider malfunctions unless we are told about them or there is some sort of inconsistency, we have this expected result. Now let's tell the program that a malfunction occurred by adding line:

$$break.$$

The program with the two new inputs has three possible worlds:

$$W_0 = \{in(1) = r_0, \dots\}$$

$$W_1 = \{in(1) = r_1, \dots\}$$

$$W_2 = \{in(1) = r_2, \dots\}$$

According to P-log semantics, the first world is assigned the probability of $\frac{1}{2}$, while the second two get $\frac{1}{4}$. Notice that the addition of $break$ changed the reasoner's degree of belief in the robot's being in r_0 from 1 to $\frac{1}{2}$ — a feat not possible in classical Bayesian updating.

Let's return to the original program and assume no given knowledge of a break occurring. Given CR-Prolog semantics, we can have inputs such as

$$enter(r_1).$$
$$obs(in(1) = r_2).$$

without contradiction because the program will simply fire the CR-rule and deduce $break$. (Note, that we have $obs(in(1) = r_2)$ instead of $in(1) = r_2$ because the value of $in(1)$ can be selected randomly.) Consider another situation. Suppose we learn that the robot tried to enter r_0 but failed to do so. This knowledge can be recorded by adding the following lines to the original program:

$$enter(r_0).$$
$$obs(in(1) \neq r_0).$$

Now there are two possible worlds:

$$W_0 = \{in(1) = r_1, break, \dots\}$$

$$W_1 = \{in(1) = r_2, break, \dots\}$$

each with probability $\frac{1}{2}$, which conforms to intuition. We hope this example demonstrates the utility of combining CR-Prolog and P-log and the simplicity with which this can be done.

Summary

In this chapter we defined the syntax and semantics of knowledge representation language P-log capable of combining logical and probabilistic reasoning. The logical part of the language is based on ASP and its extensions. The probabilistic part adopts the idea of causal probability from Pearl. The interpretation of the probability of an event as a measure of the degree of a rational agent's belief in the truth of that event allows for a natural combination of the two formalisms.

A number of examples shows how the language can be used to formalize subtle forms of reasoning including both probability and logic. There is a substantial number of other attempts to combine logical and probabilistic reasoning. There is, however, a collection of properties of P-log which are not seen together in other languages:

- P-log probabilities are defined with respect to an explicitly stated knowledge base. In many cases this greatly facilitates creation of probabilistic models.

- In addition to logical non-monotonicity, P-log is "probabilistically non-monotonic" — addition of new information can add new possible worlds and substantially change the original probabilistic model, allowing for Bayesian learning.

- Possible knowledge base updates include defaults, rules introducing new terms, observations, and deliberate actions in the sense of Pearl.

The language is comparatively new and is currently the subject of extensive investigation. In particular, we can expect substantial progress in the automation of P-log reasoning.

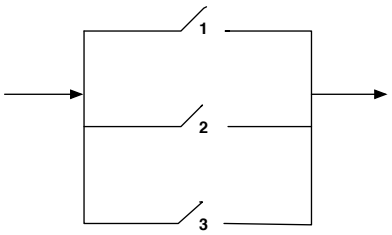


Figure 11.1: Parallel System

Exercises

1. A parallel system consists of 3 components, and it functions if at least one component works. The probability that each component works is independent of the others.

   (a) Write a P-log program to describe this system. Use the notion of probabilistic measure defined by this program to compute the probability that the system functions.

   (b) How would you use P-log to write that the first component was observed to be broken?

   (c) How would you use P-log to write that the first component was broken on purpose?

   (d) Suppose the system consists of components a, b, and c. Part a works $\frac{1}{2}$ of the time, part b works $\frac{1}{3}$ of the time, and part c works $\frac{2}{5}$ of the time. Write statements that would allow the program to take this new information into account. Compute the probability that the system functions.

   (e) Change the program to describe a serial system. Compute the probability that the system functions. (A serial system requires all components to work.)

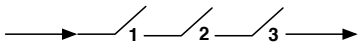


Figure 11.2: Serial System

2. Write a P-log program to represent the information about cards from Example 11.3.4 that does not require the use of a dynamic range. *Hint:* Use attribute

$$draw\_ace : try \rightarrow boolean.$$

and define pr-atoms for $draw\_ace(1)$ and $draw\_ace(2)$.

3. At a party names of dance partners are drawn at random from two hats. The first hat contains the names of all the boys and the second — those of the girls. There are 3 names in each hat. Write a P-log program to compute the odds that boy 1 and girl 1 will end up as dance partners.

4. Prove that for any disjoint subsets E_1 and E_2 of Ω, $P(E_1 \cup E_2) = P(E_1) + P(E_2)$.

Chapter 12

The Prolog Programming Language

We complete the book by a short discussion of an inference mechanism which is very different from the one presented in Chapter 7. It is only applicable to *nlp* programs[1] and, although sound with respect to answer set semantics of *nlp*, it is not complete, and might even not terminate; however, it has a number of advantages. In particular, it is applicable to *nlp*s with infinite answer sets and it does not require grounding. The algorithm is implemented in programming language Prolog. The language, introduced in the late 1970s, is still one of the most popular universal programming languages based on logic. Syntactically, Prolog can be viewed as an extension of the language of *nlp*s by a number of non-logical features. Its inference mechanism is based on on two important algorithms called *unification* and *resolution*, implemented in standard Prolog interpreters. **Unification** is an algorithm for matching atoms; **resolution** uses unification to answer queries of the form "find X such that $q(X)$ is the consequence of an *nlp* Π."

We will end this chapter with several examples of the use of Prolog for finding declarative solutions to non-trivial programming problems. Procedural solutions to these problems are longer and much more complex.

[1] Recall from Chapter 3 that an *nlp* is a logic program without classical negation and disjunction, i.e., a program consisting of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \ not\ a_{m+1}, \ldots, \ not\ a_n$$

where as are atoms.

12.1 The Prolog Interpreter

We start with defining unification and **SLD resolution** — an algorithm used by Prolog interpreters to answer queries to definite programs (i.e., *nlp* programs without default negation). Then we will look at an example of computing answers to queries to an *nlp* program with default negation using **SLDNF resolution.**[2](The accurate description of the algorithm is non-trivial but we believe that the example is enough to illustrate the process for its practical understanding.)

12.1.1 Unification

The unification algorithm forms the core of resolution. Given a collection of atoms it checks if these atoms can be made identical by substitutions of terms for variables. For instance, atoms $p(X, a)$ and $p(b, Y)$ can be made identical by substitution $\alpha = [X{=}b, Y{=}a]$ which replaces X by b and Y by a; atoms $p(a)$ and $p(b)$ cannot be made identical by any substitution. Unexpectedly this seemingly simple procedure becomes non-trivial in more complex examples. Before giving a general unification algorithm we will need the following definitions.

Definition 12.1.1. *A* **substitution** *is a finite mapping from variables to terms. We will only consider substitutions α such that for any variable X, X does not occur in $\alpha(X)$.*

A substitution α defines a mapping on arbitrary expressions: for any expression A, $\alpha(A)$ is defined as the result of simultaneously applying α to all occurrences of variables in A. For instance, if $\alpha = [X{=}b, Y{=}a]$ and $A = p(X, Y, f(X))$ then $\alpha(A) = p(b, a, f(b))$.

Definition 12.1.2. *A substitution α is called a* **unifier** *of expressions A and B if $\alpha(A) = \alpha(B)$.*

Definition 12.1.3. *A substitution α is called a* **most general unifier (mgu)** *of A and B if:*

1. *$\alpha(A) = \alpha(B)$ (i.e. α is a unifier)*
2. *α is more general than any other unifier β of A and B; i.e. there is a substitution γ such that for any variable X from the domain of α, $\beta(X) = \gamma(\alpha(X))$.*

[2]The NF in SLDNF resolution stands for negation as failure — an alternative name for default negation.

It is possible to prove that most general unifiers can only differ by the names of variables, but we will not do this here for the sake of brevity. Due to this result, we sometimes say "the" mgu instead of "an" mgu.

Here are a few examples to illustrate the definitions.

1. Mappings $\beta = [X{=}a, Y{=}a]$ and $\alpha = [X{=}Y]$ are unifiers of expressions $f(X)$ and $f(Y)$. Mapping β says that you can unify $f(X)$ and $f(Y)$ by replacing X by a and Y by a. Mapping α says that you can unify these expressions by simply replacing X in $f(X)$ by Y; it requires one less step. It is easy to see that α is more general than β since $\beta(X) = \gamma(\alpha(X))$ for $\gamma = [Y{=}a]$. In fact (as we show later), α is a most general unifier.

2. Mapping $[X{=}a, Y{=}f(b)]$ is an mgu of expressions $f(X, f(b))$ and $f(a, Y)$.

3. Expressions $f(X)$ and $f(f(X))$ are not unifiable.

Theorem 3. *(Unification Theorem)*
(Herbrand, Robinson, Martelli and Montanari)
There exists an algorithm which for any two atoms produces their most general unifier if they are unifiable and otherwise reports nonexistence of a unifier.

Proof: Notice that atoms are unifiable only if they start with the same predicate symbols. Therefore it suffices to describe a unification method for atoms $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$, i.e. to find a unifier for a set of equations

$$S_0 = \{t_1{=}s_1,\ \ldots,\ t_n{=}s_n\}.$$

This can be done by the following algorithm.

For the purpose of this algorithm, treat constants as function symbols of arity 0. Non-deterministically choose from the set of equations of a form below and perform the associated action:

	Equation	**Action**
(1)	$f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$	replace by the set $\{t_i = s_i : i \in [1..n]\}$
(2)	$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$	stop with failure
(3)	$X = X$	delete the equation
(4)	$t = X$ where t is not a variable	replace by $X = t$
(5)	$X = t$ where t is not X, and X has another occurrence in the set of equations	if X occurs in t then stop with failure else replace X by t in every other equation

Let $X_1, \ldots, X_n$ be the set of variables occurring in S_0. We will show that the algorithm stops with failure or produces a substitution of the form $[X_1{=}t_1, \ldots, X_n{=}t_n]$. The key is to show that the algorithm always terminates.

Consider a trace of the algorithm consisting of sets $S_0, S_1, S_2, \ldots$ where S_i is obtained from S_{i-1} by an application of one of the above rules. Notice that a successful application of rule (5) to variable X_i reduces the number of occurrences of X_i to one, and that application of no other rule increases this number. Since S_0 contains a finite number of variables and since rule (5) is not applicable to a set with no multiple occurrences of variables, we can conclude that rule (5) can only occur in a trace a finite number of times. Let S_n be the set obtained by the last application of rule (5). Since an application of rule (1) to a set S decreases the number of occurrences of function symbols in S and no other rule except (5) increases this number, there can only be a finite number of applications of rule (1) in a trace after S_n. A similar argument can be used for other rules and hence any trace of the algorithm is finite. From the definition of the algorithm's rules it is clear that for every rule r, substitution α is a unifier of a set E of equations iff α is a unifier of the result of applying r to E. This implies that if the two atoms are unifiable, the last collection of equations is their most general unifier. As mentioned above such a unifier is unique modulo renaming of variables.

Let's apply the algorithm to our previous examples.

Example 12.1.1.
• Unify $p(f(X))$ and $p(f(Y))$.
We record the trace of the algorithm by listing equations in each set formed by the iterations.

$$S_0 \ \left\{ \ f(X) = f(Y)$$

Rule (1) gives us:

$$S_1 \ \left\{ \ X = Y$$

No other rules are applicable. Hence, S_1 is an mgu of $p(f(X))$ and $p(f(Y))$.

Example 12.1.2.
• Unify $p(f(X, f(b)))$ and $p(f(a, Y))$.

$$S_0 \ \left\{ \ f(X, f(b)) = f(a, Y)$$

Rule (1) gives us:

$$S_1 \ \left\{ \begin{array}{l} X = a \\ f(b) = Y \end{array} \right.$$

Rule (4) requires that we flip the second equation:

$$S_2 \begin{cases} X = a \\ Y = f(b) \end{cases}$$

S_2 is the mgu.

Example 12.1.3.
• Unify $p(f(X))$ and $p(f(f(X)))$.
The algorithm forms equation $X{=}f(X)$ by rule (1) and stops with failure by rule (5).

Below are a few more examples:

Example 12.1.4.
• Unify $p(a)$ and $p(b)$.
Applying rule (1) of the algorithm forms equation $a = b$. Since the algorithm treats constants as function symbols of arity 0, it stops with failure by rule (2).

Example 12.1.5.
• Unify $p(X, f(f(a, b), X))$ and $p(g(c), f(Y, Z))$.
(The equation selected for the next step is indicated by an arrow.)

$$S_0 \begin{cases} X = g(c) \\ f(f(a, b), X) = f(Y, Z) \quad \longleftarrow \end{cases}$$

Rule (1) gives us the following set of equations:

$$S_1 \begin{cases} X = g(c) \quad \longleftarrow \\ f(a, b) = Y \\ X = Z \end{cases}$$

Rule (5) applied to $X = g(c)$ tells us to replace other occurrences of X by $g(c)$:

$$S_2 \begin{cases} X = g(c) \\ f(a, b) = Y \\ g(c) = Z \quad \longleftarrow \end{cases}$$

Rule (4) makes $g(c) = Z$ into $Z = g(c)$:

$$S_3 \begin{cases} X = g(c) \\ f(a, b) = Y \quad \longleftarrow \\ Z = g(c) \end{cases}$$

and $f(a, b) = Y$ into $Y = f(a, b)$:

$$S_4 \begin{cases} X = g(c) \\ Y = f(a, b) \\ Z = g(c) \end{cases}$$

The equations in S_4 describe the mgu.

12.1.2 SLD Resolution

The Prolog interpreter uses resolution-based algorithms as its method for finding answers to queries to a logic program. In this section we define **SLD resolution** — an algorithm used by Prolog interpreters to answer queries to definite programs (i.e. *nlp* programs without default negation).

Then we will look at an example of computing answers to queries to an *nlp* program with default negation using **SLDNF resolution.**[3] (The accurate description of the algorithm is non-trivial but we believe that the example is enough to illustrate the process for its practical understanding.)

We start with basic definitions:

Definition 12.1.4. *Let $Q = [q_0, \ldots, q_m]$ where the qs are atoms be a* **query** *to a definite program Π. An* **answer** *to Q by Π is a substitution α such that*

$$\Pi \models \forall\ \alpha(Q)$$

i.e. every ground instance of $\alpha(Q)$ belongs to the answer set of Π.

For example, consider Π_1:

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

and query $Q = [p(X), q(X)]$. An answer to Q by Π is $[X{=}a]$. You can see that the ground instance of $\alpha(Q)$ is $\{p(a), q(a)\}$ and that it belongs to the answer set of Π.

Now we will describe the SLD resolution method for answering queries to definite programs. For the purpose of our resolution algorithm we represent the original query Q by a rule

$$yes(X_1, \ldots, X_k) \leftarrow Q \tag{12.1}$$

[3]The NF in SLDNF resolution stands for negation as failure — an alternative name for default negation.

where $X_1, \ldots, X_k$ is the list of variables of Q and yes is a new predicate symbol not appearing in the program. (In our further discussion we will refer to rules containing the yes predicate symbol in the head as **query rules**.) It is easy to see that α is an answer to a query Q by Π_1 iff α is an answer to a query $yes(X_1 \ldots, X_k)$ by $\Pi_1 \cup (12.1)$.

A solution α to a query Q is then found by a series of consecutive transformations of the initial query rule which leads to a single atom $yes(t_1, \ldots, t_k)$ where $\alpha = [X_1{=}t_1, \ldots, X_k{=}t_k]$.

For example, consider program Π_2 consisting of three rules:

$$p(X, Y) \leftarrow q(X, b), r(f(X), Y). \qquad (\Pi_2{:}1)$$

$$q(a, b). \qquad (\Pi_2{:}2)$$

$$r(f(a), b). \qquad (\Pi_2{:}3)$$

and query $Q = [p(X, Y)]$. The initial query rule, denoted by r_1, will have the form

$$yes(X, Y) \leftarrow p(X, Y). \qquad (r_1)$$

Rule (Π_2:1) will be used to transform r_1 into another query rule r_2 of the form

$$yes(X, Y) \leftarrow q(X, b), r(f(X), Y). \qquad (r_2)$$

The new rule is obtained by replacing $p(X, Y)$ by the body of (Π_2:1). Intuitively it is clear that if α is a solution to query $yes(X, Y)$ with respect to program $\Pi_2 \cup \{r_2\}$ then it is a solution to the same query with respect to program $\Pi_2 \cup \{r_1\}$. The second rule of Π_2 will be used to transform r_2 into the next query rule r_3

$$yes(a, Y) \leftarrow r(f(a), Y). \qquad (r_3)$$

This time the transformation is slightly more involved. It consists of two steps. During the first step query $q(X, b)$ is unified with $q(a, b)$ — the head of rule (Π_2:2). The corresponding substitution is applied to the whole rule transforming it into $yes(a, Y) \leftarrow q(a, b), r(f(a), Y)$. The second step replaces $q(a, b)$ by the body of rule (Π_2:2) — in our case the empty set. Finally similar transformation which uses the last rule (Π_2:3) is applied to transform r_3 into r_4:

$$yes(a, b). \qquad (r_4)$$

Clearly $\alpha = [X{=}a, Y{=}b]$ is a solution of Q with respect to original program Π_2.

In general, each step of this process is accomplished by the following rule called the Resolution Rule:

Definition 12.1.5. *(Resolution Rule)*
Consider a query rule r_q

$$yes(t_1, \ldots, t_k) \leftarrow q_0, \Delta$$

where q_0 *is the first predicate in the body and* Δ *represents the rest of the query predicates, and a program rule* r_p

$$p_0 \leftarrow \Gamma$$

such that q_0 *and* p_0 *are unifiable and* r_q *and* r_p *do not have common variables.*

Rule r

$$\alpha(yes(t_1, \ldots, t_k) \leftarrow \Gamma, \Delta).$$

where α *is the mgu of* p_0 *and* q_0 *is called the* **resolvent** *of rules* r_q *and* r_p*. A triple* $\langle r_q, r_p, r\rangle$ *is referred to as the* **SLD resolution inference rule** *with premises* r_q, r_p *and conclusion* r*.*

It can be shown that if α is a solution to query $yes(X, Y)$ with respect to program $\Pi \cup \{r_q\}$ then it is a solution to the same query with respect to program $\Pi \cup \{r\}$.

The following is an example of the application of the resolution inference rule. Consider a query rule r_q,

$$yes(X, Y) \leftarrow p(X, Y), s(X)$$

and a program Π_3 consisting of a single rule r_p,

$$p(a, Y_1) \leftarrow r(f(a), Y_1).$$

The first rule, r_q, has the form

$$yes(X, Y) \leftarrow q_0, \Delta$$

where q_0 is $p(X, Y)$ and Δ is $s(X)$. The second, r_p, is of the form

$$p_0 \leftarrow \Gamma$$

where p_0 is $p(a, Y_1)$ and Γ is $r(f(a), Y_1)$. Clearly, the two rules have no common variables and $\alpha = [X{=}a, Y{=}Y_1]$ is an mgu of q_0 and p_0. Hence the resolvent of r_q and r_p is r:

$$yes(a, Y_1) \leftarrow r(f(a), Y_1), s(a).$$

Now we will define the notion of SLD derivation which serves as the basis of the resolution algorithm.

A rule r_2 obtained from a rule r_1 by renaming its variables is called a **variant** of r_1.

Definition 12.1.6. *(SLD derivation)*
An **SLD derivation** *of Q from Π is a sequence $r_0, \ldots, r_n$ of query rules such that*

- *$r_0 = \{yes(X_1, \ldots, X_k) \leftarrow Q\}$ where $X_1, \ldots, X_k$ is the list of variables in Q.*
- *r_i is obtained by resolving r_{i-1} with a variant r of some rule of Π such that r and r_{i-1} have no common variables.*
- *r_n has the empty body.*

The following theorems explain the importance of SLD derivation.

Theorem 4. *Soundness of SLD derivation*
If $yes(t_1, \ldots, t_k)$ is the last rule in the SLD derivation of $Q(X_1, \ldots, X_k)$ from Π, then the substitution $[X_1{=}t_1, \ldots, X_k{=}t_k]$ is an answer to Q by Π.

Theorem 5. *Completeness of SLD derivation*
If a ground query $Q(c_1, \ldots, c_k)$ is true in the answer set of Π, then there is an SLD derivation of $Q(X_1, \ldots, X_k)$ from Π with the last rule $yes(t_1, \ldots, t_k)$ such that $Q(c_1, \ldots, c_k)$ is a ground instantiation of $Q(t_1, \ldots, t_k)$.

Clearly a sequence of rules r_q, r_p, r given above is an SLD derivation of query $[p(X, Y), s(X)]$ from program Π_3. Similarly, the sequence r_1, r_2, r_3 of rules constructed earlier to illustrate the derivation of query $p(X, Y)$ from program Π_2. There is a subtle point we need to make here. Even though r_1, r_2, r_3 is an SLD derivation, our explanation of its construction is not, strictly speaking, accurate. A careful reader will notice that, in this construction, something similar to the resolution rule was applied to rules (Π_2:1) and r_1 which have common variables — this is prohibited by the definition of SLD derivation. Of course the problem can be easily remedied by renaming variables of one of the rules. So why does the definition includes such a requirement? The next example gives an answer to this question and shows that such renaming is indeed necessary.

Consider a program Π_4 that consists of the rule

$$p(f(X)).$$

Suppose we have query $p(X)$. If we ignore the requirement and attempt to apply the resolution to the program's rule and the first query rule

$$yes(X) \leftarrow p(X). \qquad (r_1)$$

we fail. (This, of course, happens because $p(X)$ and $p(f(X))$ are not unifiable.) However, $p(X)$ and $p(f(Y))$ are unifiable. Resolving query rule r_1 with variant $p(f(Y))$ of the program rule produces resolvent r_2

$$yes(f(Y)). \qquad (r_2)$$

Since this rule has an empty body, r_1, r_2 is our SLD derivation. The answer to query $p(X)$ is substitution $[X{=}f(Y)]$. Without renaming variables we would fail to produce this answer.

12.1.3 Searching for SLD Derivation — Prolog Control

An algorithm implemented in the interpreter for definite Prolog programs can be viewed as a depth-first search for an SLD derivation of Q from Π in the space of "candidate derivations." In this search a "candidate rule" r_i of this derivation is always obtained by resolving the candidate rule r_{i-1} with the topmost available rule of Π. For example, let Π be a program consisting of the following three rules:

$$p(a). \qquad (1)$$

$$p(b). \qquad (2)$$

$$q(b). \qquad (3)$$

To answer query $Q(X) = [p(X), q(X)]$ the Prolog interpreter will create the first query rule r_0

$$yes(X) \leftarrow p(X), q(X) \qquad (r_0)$$

and resolve it with topmost rule (1) of the program. The body, $q(a)$, of the resulting query rule

$$yes(a) \leftarrow q(a) \qquad (r_1)$$

is, however, not unifiable with any rule of the program. Hence, no further resolution is possible. Since the body of (r_1) is not empty, the algorithm **backtracks**. Now (r_0) is resolved with rule (2) since it is the first untried rule of the program. The resulting query rule

$$yes(b) \leftarrow q(b) \qquad (r_2)$$

is further resolved with rule (3) which obtains

$$yes(b). \qquad (r_3)$$

Clearly, $(r_0), (r_2), (r_3)$ is an SLD derivation and the answer to our query is

$$X = b.$$

The use of depth-first search allows the Prolog interpreter to find answers without using a large amount of memory (as opposed to the use of breadth-first search). It is, however, responsible for non-termination of programs which causes a substantial problems to beginning Prolog programmers. Consider for instance a program

$$\Pi_1 \left\{ \begin{array}{l} p \leftarrow p. \\ p. \end{array} \right.$$

and a query p. Clearly the answer to the query according to Prolog semantics is *yes*. However, it is not difficult to see that the search mechanism implemented in the Prolog interpreter will not be able to produce this answer. After producing the query rule

$$yes \leftarrow p$$

the interpreter will resolve it with the topmost possible rule $p \leftarrow p$ of the program. The result will be again

$$yes \leftarrow p.$$

The interpreter will not be able to find its way out of this loop.

There of course is an SLD derivation of a rule

$$yes$$

which can be obtained by resolving the query rule with the second rule of the program but this derivation will never be found.

The same query to the semantically equivalent program

$$\Pi_2 \left\{ \begin{array}{l} p. \\ p \leftarrow p. \end{array} \right.$$

will return *yes* but will go into the loop if the interpreter is forced to backtrack.

The limited control strategy of Prolog does not allow it to be viewed as a fully declarative language. If it were, termination properties of its programs would not be dependent on the rules' order or other semantically equivalent transformations.

There are modifications of control strategies implemented in Prolog interpreter which try to alleviate this problem. For instance, the interpreter of the XSB logic programming system tables information about the querying-answering process. Given a program Π_1 above and a query p the XSB interpreter will resolve the query rule with the first rule of the program and record this fact in a table. If the backtracking requires resolution of the same query rule (as in our example) XSB will consult the table, discover that this query rule has already been resolved with the first rule of Π_1 and will not attempt to repeat the process. Instead it will resolve the query rule with the second rule p of Π_1 obtaining the desired answer *yes*.

If the program contains variables and negation the tabling mechanism is deeply non-trivial but even in this case XSB manages to avoid a large number of potential loops. It is important to realize however that the complete elimination of loops is impossible. This follows immediately from the fact that Prolog is a universal programming language and can therefore represent any algorithm. Hence, according to the famous undecidability result by Turing it is impossible to design an algorithm capable of detecting all the loops in an arbitrary program.

12.1.4 SLDNF Resolution

In this section we will discuss the Prolog interpreter for programs containing default negation *not*. Let us first expand the definition of a query from the section on SLD resolution. Now by a **query** we will mean a sequence $Q = [q_0, \ldots, q_m]$ where qs are atoms or atoms preceded by default negation *not*. To answer queries for *nlp* programs with *not* the Prolog interpreter implements a version of an extension of SLD resolution. The precise definition of this algorithm, called SLDNF resolution, is somewhat complex so we will explain it by way of example.

Let us consider a program Π consisting of the four rules below:

$$p(a). \tag{Π:1}$$

$$p(b). \tag{Π:2}$$

$$q(a). \tag{Π:3}$$

$$r(X) \leftarrow p(X),\ not\ q(X). \tag{Π:4}$$

and a query $r(X)$ represented as a rule:

$$yes(X) \leftarrow r(X). \qquad (r_0)$$

The Resolution Rule from Definition 12.1.5 can be naturally expanded to rules in which atoms in the body can be preceded by *not*. SLDNF resolution will use this rule to resolve query r_0 with program rule (Π:4). The resulting query rule,

$$yes(X) \leftarrow p(X),\ not\ q(X) \qquad (r_1)$$

will be resolved with program rule (Π:1) to obtain the next query rule

$$yes(a) \leftarrow\ not\ q(a). \qquad (r_2)$$

The first atom in the body of this rule is preceded by default negation, and hence Resolution Rule is not applicable. To proceed we need a new inference rule called **Negation as Finite Failure (NAF)**: *A query rule*

$$yes(t_1, \ldots, t_k) \leftarrow\ not\ q, \Delta$$

where q is a ground atom derives a query rule

$$yes(t_1, \ldots, t_k) \leftarrow \Delta$$

if SLDNF resolution can consider all possible ways to prove q and demonstrate that all of them fail.[4]

If we go back to our example we will see that SLDNF's attempt to prove $q(a)$ immediately succeeds. Rule r_2 can be used by neither Resolution Rule nor *NAF* and hence the algorithm is forced to backtrack to rule (r_1). By resolving (r_1) and (Π:2) it finds another candidate rule:

$$yes(b) \leftarrow\ not\ q(b). \qquad (r_3)$$

It is easy to see that the query $q(b)$ cannot be resolved with anything else, and cannot, therefore, be proven by SLDNF resolution. At this point the resolution algorithm uses *NAF* and removes *not* $q(b)$ from the body of (r_3) producing the rule

$$yes(b) \qquad (r_4)$$

[4]This description is admittedly vague. Since *NAF* is used in the definition of SLDNF resolution, the description is circular and requires clarification. It is exactly this problem that makes the precise description of SLDNF resolution non-trivial. In most cases, however, this circularity does not cause any problem.

which contains the answer to the initial query.

Notice that the algorithm only works if the corresponding negated query is ground. Otherwise the interpreter is said to **flounder** and its actual behavior depends on the implementation.

It can be shown that *if the algorithm does not flounder then it is sound with respect to the answer set semantics.* It is also shown to be complete for a comparatively large class of programs. Later we will spend more time studying the completeness and termination properties of SLDNF resolution.

12.2 Programming in Prolog

In this section we briefly discuss data representation in Prolog and give examples of using the language for solving several non-trivial programming problems. It is not our purpose to teach Prolog programming for which many good books have already been written. Rather, we want to give readers a feel for the language and the types of solutions its use can produce. We use actual Prolog programming systems (such as SICStus, SWI, GNU, etc.) that include several "meta-level" constructs which do not fit into the syntax of *nlp* rules we discussed so far. We give intuitive explanations of these Prolog features without going deeply into the precise definition of the semantics so as not to digress from the main goal with a lengthy discussion. As examples of Prolog use, we present solutions to two classical problems. The first one, referred to as the *parts inventory problem*, consist of finding the type and quantity of basic parts needed to assemble a given number of products (in our case, bicycles). The second is that of *computing derivatives of polynomials.*

Note that the built-in operator `\+` is used in many versions of Prolog to denote default negation.

12.2.1 The Basics of Prolog Programming

We start with a simple program stating that two terms, a and b, satisfy property p.

```
% First Prolog Program
p(a).
p(b).
```

Let us assume that the program is stored in file `first.pl`.

Let's run the program in Prolog. You can use any Prolog system mentioned above. Once you've installed the system, invoke it from the command

line. (Make sure you're in the directory where your program resides.) You should see a question mark prompt. For instance, if you type

```
swipl
```

for SWI Prolog, you will see the prompt

```
?-
```

You will need to tell the interpreter to load the program. This is called "consulting" the file. If you use extension `.pl` in naming your file, you can omit it when consulting and just say:

```
?- [first].
```

Prolog should respond with a message telling you that it has compiled your program. (If you do not use `.pl` but some other extension, say `.ext`, you will have to say `['first.ext']`.

Next, type in the following query to Prolog and hit return. It will respond with `true` and return you to a Prolog prompt.

```
?- p(a).
true
?-
```

The query $p(X)$ will produce the following:

```
?- p(X).
X = a
```

At this point you have a choice. If you hit the return/enter key, Prolog will say `true`, return to the Prolog prompt and wait for further input. Sometimes, however, you will want to force the interpreter to backtrack to see if there are any more possible answers to the query. If you type in a semicolon, you will do just that. If there is another answer, Prolog will return the first one it finds and, again, wait to see if you want more. In our case we will have

```
?- p(X);
X = a ;
X = b ;
false
```

Answer `false` indicates that there is no other correct answer to the query. To exit the Prolog environment, type `halt.`

Consider now program

```
% Second Prolog Program
p(a).
p(f(X)) :-
    p(X).
```

stored in file `second.pl`. If you consult this program and ask it query $p(X)$, Prolog will return `X = a`. If prompted to backtrack, it will return `p(f(a))`, `p(f(f(a)))`, etc. Since the answer set of the program contains infinite set of answers to our query it will be up to you to stop the process.

For the next example consider a directed acyclic graph represented by relation $edge(X, Y)$ and relation $connected(X, Y)$ which holds if there is a path in the graph from node X to node Y.

```
% Graph Connectivity
edge(a,b).
edge(a,c).
edge(b,d).
connected(X,Y) :-
    edge(X,Y).
connected(X,Y) :-
    edge(X,Z),
    connected(Z,Y).
```

The program will respond to query $connected(X, Y)$ as follows:

```
X = a,
Y = b ;

X = a,
Y = c ;

X = b,
Y = d ;

X = a,
Y = d ;

true
```

correctly returning all the pairs satisfying our relation. Note, however, that if we replace the last rule of the program by equivalent rule

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

and force Prolog to backtrack, it will quickly run out of memory. (In our experiments, Prolog found the four answers but, instead of answering *false*, it returned an error message.) This happens because, due to the control strategy of the Prolog interpreter, it will go into an infinite loop. As we mentioned in Section 12.1.3 Prolog programmers cannot fully rely on the declarative meaning of Prolog statements and need to firmly keep in mind the basic strategy of Prolog control.

In addition to terms Prolog has another useful datatype called the **list**. A list of objects can be formed by using square brackets. An expression $[a, b, c]$ defines a list of three terms, a, b, and c; [ ] denotes the empty list. A special expression $[F|T]$ denotes a list with head F and tail T. Here F is the first element of the list and T is the list consisting of the rest of the elements. For instance, list $[a, b, c]$ can be written as $[a|[b, c]]$; list $[a]$ as $[a|[]]$, etc. This definition is reflected in the unification algorithm. For instance, list $[F|T]$ is unified with a list $[a, b, c]$ by substitution $F{=}a, T{=}[b, c]$. The head and tail notation for lists is very convenient for writing recursive definitions of list properties and relations. For example, the following program tests the membership relation in lists:

```
% Program in1.pl
is_in(X,[X|_]).
is_in(X,[Y|T]) :-
    is_in(X,T),
    diff(X,Y).
eq(X,X).
diff(X,Y) :-
    \+ eq(X,Y).  % i.e., not eq(X,Y)
```

In the first rule '_' stands for an arbitrary term and is often called an "anonymous variable." (Any variable except X can be used instead but it is easier to write '_'.)

The definition can be used to check if a ground term t belongs to a ground list l. For instance, to check membership of a in $[a, b, c]$ we ask a query $is\_in(b, [a, b, c])$ which, as expected, will be answered by `true`; query $is\_in(d, [a, b, c])$ will be answered by `false`.

In addition to checking if an element belongs to a list, the definition of *is_in* can be used to find list members. This can be done by a query of the form $is\_in(X, list)$ where *list* is ground. For example,

```
?- is_in(X,[a,b,a]).
X = a ;
X = b ;
false.
```

Sometimes it is useful to use a more complex non-ground term as the first parameter. For instance, to answer query $is\_in([X, b], [[a, b], c, [d, b]])$ Prolog does the necessary matching and responds as follows:

```
?- is_in([X,b],[[a,b],c,[d,b]]).
X = a ;
X = d ;
false.
```

The examples show that the program is capable of answering queries whose parameters are arbitrary term and ground term, respectively. (In fact it can be proven that such queries are always correctly answered by the program.) But such good behavior is not guaranteed if the second parameter is not ground.

For instance a query $is\_in(a, X)$ will be answered by `X = [a|_A]?` read as "any list which starts with a'. This is, of course, a correct answer but, if prompted for another answer, the program will go into an infinite loop (which, most likely, will be indicated by a message mentioning "insufficient memory").

To avoid using a Prolog definition for answering unintended queries, Prolog programmers are strongly advised to mention the type of expected queries in documentation. In what follows we will use symbol + if the corresponding parameter must be ground, − if it contains variables, and ? if the parameter is arbitrary, i.e. if it may or may not contain variables. For instance, the above program will be preceded by a comment line

```
% is_in(?,+).
```

which indicates that the second parameter of the query formed by *is_in* must be ground. Sometimes a mapping of parameters into $\{+, -, ?\}$ is called the **mode** of a definition.

One might ask why we did not use a definition of *is_in* given by the following rules:

```
% Program in2.pl
is_in(X,[X|_]).
is_in(X,[Y|T]) :-
    is_in(X,T).
```

This will work too but there would be slight differences between the two definitions. For instance, let us run Prolog on the new program with query $is\_in(X, [a, b, a])$

```
?- is_in(X,[a,b,a]).
X = a ;
X = b ;
X = a ;
false.
```

The last answer is redundant and was not produced by the first program. To understand the difference let us trace the execution of both programs. For simplicity we drop the extension *pl* and refer to the programs as `in1` and `in2`. On the first call to program `in2`, Prolog instantiates X with a using the first rule. Similarly for `in1`. When asked to backtrack in `in2`, Prolog uses the second rule of `in2`, instantiates Y with a and T with $[b, a]$ and calls *is_in(X,[b,a])*. Backtracking in `in1` is almost identical except the last call is *is_in(X,[b,a]), diff(X,a)*. Both programs return second answer $X = b$. One more backtrack shows us the difference: `in2` backtracks to *is_in(X,[b,a])* and returns the third answer $X = a$; `in1` backtracks to *is_in(X,[b,a]),diff(X,a)*. As in `in2` the call to $is\_in(X, [b, a])$ returns $X = a$, but *diff(a,a)* fails and `in1` exits without returning a redundant answer.

This happens because the two rules in the definition of `in2` are not mutually exclusive; the corresponding rules of `in1` are. It is a good programming practice to keep rules in the definition of a relation mutually exclusive to avoid unintended consequences.

Note also that, although it would be logical to write

```
is_in(X,[Y|T]) :-
    diff(X,Y),  % causes floundering!
    is_in(X,T).
```

we cannot, because this would cause floundering as mentioned in Section 12.1.4. (Recall that `\+` is used in the definition of *diff*.) This undesirable behavior of the interpreter can be avoided if we put the is_in predicate first because it would serve to ground the variable and *diff* would no longer be called

incorrectly. Relation *is_in* is similar to relation *member* which is predefined in some Prologs and is part of a list library in others.

There are many other useful relations on lists that have been implemented as part of the Prolog systems one way or another; e.g.

$length(L, N)$ iff N is the length of list L.

$append(L_1, L_2, L_3)$ iff L_3 is the result of appending L_1 and L_2.

Others can be defined by the programmer; e.g.
relation $rm\_dupl(L_1, L_2)$ with mode $rm\_dupl(+, -)$ finds a list L_2 which is obtained from a ground list L_1 by removing duplicate elements. The relation is defined by the following rules:

```
% rm_dupl(+,-)
rm_dupl([],[]).
rm_dupl([X|T], L2) :-
    is_in(X,T),
    rm_dupl(T,L2).
rm_dupl([X|T],[X|T1]) :-
    \+ is_in(X,T),
    rm_dupl(T,T1).
```

The program will answer query $rm\_dupl([a, a, b, c, b], X)$ as follows:

```
?- rm_dupl([a,a,b,c,b],X).
X = [a, c, b] ;
false
```

In other words the above definition allows us to find only one answer to the above query. Another valid answer, [a,b,c], will not be found. If one desires to get all the answers, one can modify the above definition using relation *permutation*, which is available in the list libraries of most Prologs, as follows

```
% rm_dupl1(+,?)
rm_dupl1(L1,L2) :-
    rm_dupl(L1,L),
    permutation(L,L2).
```

Lists in Prolog can be created with the help of several important 'meta-level' relations. Relation *findall(Obj,Goal,Res)* which, when called, produces

a list *Res* of all the objects *Obj* that satisfy goal *Goal*. It is a built-in predicate in many Prolog systems.

Consider a knowledge base containing the test grades of students $s1, s2, s3$:

```
grade(s1,98).
grade(s2,45).
grade(s3,99).
```

The following rule defines the number of students who received an A on their test:

```
num_of(a,N) :-
    findall(X, (grade(X,Y), Y>=90), L),
    length(L,N).
```

Note that (G1,G2) denotes a conjunction of goals G1 and G2. Since L is a list and, as such, may contain duplicate entries, accidental duplication of records in the knowledge base could cause an incorrect answer. To avoid this we can replace *findall* by

```
set_of_all(X,G,L) :-
    findall(X,G,L1),
    rm_dupl(L1,L).
```

12.2.2 Parts Inventory Program

In this section we will write a more advanced Prolog program which solves the *parts inventory problem*. Any business that makes something from parts that are in turn made from other parts is faced with the need to know how many of what part to keep in stock. The program below deals with a small number of bicycle parts, but it can be easily expanded to large, realistic examples while preserving the program structure. We simply need to add more facts to our knowledge base and tailor them to the business at hand.

Here we are given a knowledge base which consist of a collection of basic bicycle parts together with the list of compound parts followed by their immediate components. We need to find the type and quantity of each basic part needed to assemble more-complex ones. More precisely, we assume that our knowledge base contains a complete list of basic parts, e.g.

```
basic_part(rim).
basic_part(nut).
basic_part(spoke).
```

```
basic_part(frame).
basic_part(brakes).
basic_part(tire).
```

By a lists of parts we mean a list of the form $[[P_1, K_1] \ldots [P_n, K_n]]$ where Ps are names of parts, if $i \neq j$ then $P_i \neq P_j$, and Ks are positive integers. $[P, K]$ is read as "K items of a part P." If all Ps of L are basic parts, we say that L is a list of basic parts. We also assume that the knowledge base contains a complete collection of compound parts together with information about the names and quantities of their immediate components, e.g.

```
components(bike,[[wheel,2],[frame,1],[steering,1]]).
components(wheel ,[[spoke,4],[rim,1],[tire,1] ,[nut,5]]).
components(steering, [[brakes,2], [nut,10]]).
```

The first parameter of this relation is a compound part P, while the second is a list L of its immediate subparts; i.e., $[P_i, K]$ is in L iff P has exactly K immediate subparts of type P_i.

We are interested in defining a relation $parts\_required(N, P, L)$ which satisfies the following two conditions: For any positive integer N and a part P

1. there is a list L such that $parts\_required(N, P, L)$ is true;
2. if $parts\_required(N, P, L)$ is true then L is a list of basic parts required to assemble N parts of type P.

We start our design by defining $parts\_required(N, P, L)$ with the mode $parts\_required(+, +, -)$. The definition is given by two rules

```
parts_required(N,P,[[P,N]]) :-
    basic_part(P).
parts_required(N,P,L) :-
    components(P,C),
    parts_for_list(C,L1),
    times(N,L1,L).
```

where the relation *parts_for_list(C,L)* satisfies the following conditions. For any list C of parts

1. there is L such that *parts_for_list(C,L)* holds;
2. if *parts_for_list(C,L)* holds then L is a list of basic parts required to assemble parts from C, and relation $times(N, L_1, L_2)$ constructs list L_2 by replacing every element $[P, M]$ of L_1 by $[P, N * M]$.

To define *parts_for_list(C,L)* with mode *parts_for_list(+,-)* we need the following rules:

```
parts_for_list([ ],[ ]).
parts_for_list([[P,K]|T],L) :-
    parts_required(K,P,Lp),
    parts_for_list(T,Lt),
    combine(Lp,Lt,L).
```

Here $combine(L_1, L_2, L)$ is a new relation which appends two lists L_1 and L_2 of parts, transforms the result into a list of parts and stores it in L.

To define relation *combine* we will first introduce an auxiliary relation $insert\_one(X, L_1, L_2)$ which inserts a list X of the form $[P, N]$ into a list L_1 of parts. The result L_2 should be a list of parts. For example

```
insert_one( [wheel,2],
            [[frame,1],[wheel,3]],
            [[frame,1],[wheel,5]]).
insert_one( [wheel,2],
            [[frame,1]],
            [[wheel,2],[frame,1]]).
```

Here is the relation:

```
%mode insert_one(+,+,-)

insert_one( X, [], [X]).
insert_one( [P,N1], [[P,N2] | T],  [[P,N] | T]) :-
    N is N1 + N2.
insert_one( [P1, N1], [ [P2,N2] | T1], [ [P2,N2] | T ]) :-
    diff(P1,P2),
    insert_one( [P1,N1], T1, T).
```

(You can see that arithmetic operators are defined in Prolog, as well as assignment (`is`)).

Now we can define relation $combine(L_1, L_2, L)$ such that

1. For every lists L_1 and L_2 of parts there is a list L such that $combine(L_1, L_2, L)$ holds.

2. If $combine(L_1, L_2, L)$ holds then $[P, N]$ belongs to L iff the list of elements of L_1 and L_2 of the form $[P, K1], \ldots, [P, Km]$ is not empty and $N = K1 + \cdots + Km$,; e.g.,

   $combine([[wheel, 2], [spoke, 5]], [[spoke, 1], [wheel, 3]], [[spoke, 6], [wheel, 5]])$

Note that the actual order of elements in the constructed list may differ from the one above.

```
%mode combine(+,+,-)

combine([],L,L).
combine([H | T], L1, L) :-
    insert_one(H,L1,L2),
    combine(T,L2,L).
```

To complete the assignment we define the relation $times(N, L_1, L_2)$ which satisfies the following conditions:

1. For any positive number N and a list L_1 of parts there is a list L_2 such that $times(N, L_1, L_2)$ holds.
2. If $times(N, L_1, L_2)$ holds then $[P, K]$ is in L_2 iff $[P, K/N]$ is in L_1.

```
%mode times(+,+,-)

times(_,[],[]).
times(N,[[P,K]|T],[[P,M]|TN ]) :-
    M is K*N,
    times(N,T,TN).
```

Finally, we define *diff(X,Y)*:

```
eq(X,X).
diff(X,Y) :-
    \+ eq(X,Y).
```

To run this program in Prolog, collect all the typewritten code into a file, say `parts.pl`. Launch Prolog from the directory that your code is in, consult the program and ask it how many basic parts are required to assemble 10 bicycles or, say, 100 wheels. Here is a sample run:

```
?- [parts].
% parts compiled 0.00 sec, 7,504 bytes
true.

?- parts_required(10, bike, L).
L = [[spoke, 80], [rim, 20], [tire, 20],
```

```
     [frame, 10], [brakes, 20], [nut, 200]] .

?- parts_required(100, wheel, L).
L = [[spoke, 400], [rim, 100], [tire, 100], [nut, 500]] .

?- halt.
```

12.2.3 (*) Finding Derivatives of Polynomials

Here is another example of using Prolog to solve complex problems with elegance. In what follows we will write a program computing derivatives of polynomials — a typical and important example of *symbolic computation.* By polynomials we mean functions constructed from a variable x, integers, function symbols $+$,$-$,$*$, exponent `^`, and parentheses. Note that X^N is defined for a non-negative integer N only. Among other things the program is aimed to illustrate the use of basic Prolog data structures — terms and lists. Those who use Sictus Prolog will need to include a library module for list operations. For SWI Prolog this is not needed.

Derivative will be computed using relation $derivative(F, DF)$ which, given a polynomial F computes the canonical form DF of its derivative. For instance, both queries

```
derivative(3*x^2+4*x+1,DF).
derivative(4*x+3*x^2+1,DF).
```

will be answered by

```
DF = 6*x+4
```

Queries

```
derivative(3*x^2-4*x+1,DF).
derivative((2*x+3)^2,DF).
```

will be answered by

```
DF = 6*x-4
```

and

```
DF = 8*x+12
```

respectively.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% COMPUTING DERIVATIVES of POLYNOMIALS
% For the purpose of this program polynomials are functions
% constructed from a variable x, integers, function symbols
% +, -, * and ^, and parentheses. Exponentiation X^N is
% defined for non-negative integer N.
% The derivative is only computed once. Multiple calls can lead
% to an infinite loop.

:-use_module(library(lists)). % not needed for SWI Prolog

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% derivative(F,DF) computes a derivative DF of a polynomial F.
% The result is given in canonical form.
% type F, DF - polynomials.
% mode derivative(+,-)

derivative(F,DF) :-
    der(F,G),
    simplify(G,DF).
```

The derivative is computed in two steps. First *der* applies the rules of derivation to F to produce a polynomial G which is then reduced to its canonical form DF by *simplify*.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% der(F,DF) computes a derivative DF of a polynomial F.
% type F, DF - polynomials.
% mode der(+,-).

der(N,0) :-
    number(N).
der(x,1).
der(-x,-1).
der(F^0,0).
der(F^N, N*F^M*DF) :-
    N>=1,
    M is N-1,
    der(F,DF).
der(E1*E2, E1*DE2 + E2*DE1) :-
```

```
    der(E1,DE1),
    der(E2,DE2).
der(E1+E2, DE1+DE2) :-
    der(E1,DE1),
    der(E2,DE2).
der(E1-E2,DE1-DE2) :-
    der(E1,DE1),
    der(E2,DE2).
der(-(E),DE) :-
    der((-1)*E,DE).
```

One can now test the definition of *der* above. Use your favorite Prolog system, consult the program, and ask the query `der(x^2,D)`. Prolog will answer with `D = 2*x^1*1`. Clearly, the answer is correct but to have a decent output this should be simplified and reduced to canonical form `2*x`.

To simplify the resulting derivative we represent polynomials as lists of the form `[...[Ci,Ni]...]` where `[Ci,Ni]` corresponds to the term `Ci*x^Ni`. Polynomials written in this form will be called **l-polynomials**. The derivative DF of F computed by `der(F,DF)` will be translated into its equivalent l-polynomial, then written in canonical form and, finally transformed back into term form. A list `L = [[C1,N1],...,[Ck,Nk]])` will be called the **list-representation** of polynomial `T = C1*x^N1 +...+ Ck*x^Nk`. This change of representation will greatly simplify the reduction to canonical form.

Relation *simplify(P,SP)*

1. Transforms polynomial P into equivalent l-polynomial SP;
2. combines like term;
3. removes terms whose coefficients are 0;
4. sorts terms in increasing order of exponents;
5. converts the resulting l-polynomial back to canonical form.

For instance, query

```
simplify(3*x^2+x^2-2*x^2+0*x^3+x^4,SP).
```

will be answered by

```
SP = x^4+2*x^2
```

Below is the complete definition.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simplify(P,SP) iff SP is a canonical form of a polynomial P
% type: P, SP - polynomials
% mode simplify(+,?)

simplify(P,SP) :-
    transform(P,TP),
    add_similar(TP,S),
    remove_zero(S,S1),
    poly_sort(S1,S2),
    back_to_terms(S2,SP).
```

Next we will describe each step in detail. Relation *transform(P,L)* translates polynomial P into an equivalent l-polynomial L. For instance query

```
transform(3*x^2+4*x+1,L).
```

will be answered by

```
L = [[3,2],[4,1],[1,0]]
```

Here is the definition:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transform(T,[[C1,N1],...,[Ck,Nk]]) implies that
% T = C1*x^N1 +...+ Ck*x^Nk.
% type: T - polynomial, C - integer, N - nonnegative integer.
% mode: transform(+,?).

transform(C,[[C,0]]) :-
    integer(C).
transform(x,[[1,1]]).
transform(-x,[[-1,1]]).
transform(A*B,L) :-
    transform(A,L1),
    transform(B,L2),
    multiply(L1,L2,L).
transform(A^N,L) :-
     transform(A,L1),
     get_degree(L1,N,L).
```

```
transform(A+B,RT) :-
     transform(A,RA),
     transform(B,RB),
     append(RA,RB,RT).
transform(A-B,RT) :-
     transform(A,RA),
     transform(-B,RB),
     append(RA,RB,RT).
transform(-(A),R) :-
     transform((-1)*A,R).
```

The definition uses two new relations, $multiply(L_1, L_2, L)$ and $get\_degree(L_1, N, L)$. The former simply multiplies two polynomials written in list form. For instance, a query

```
multiply([[1,2],[3,4]],[[2,3],[3,2]],Y).
```

will be answered by

```
Y = [[2,5],[3,4],[6,7],[9,6]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% multiply(L1,L2,L) L is the product of L1 and L2
% type: L1,L2,L - l-polynomials
% mode: multiply(+,+,-)

multiply_one(_,[],[]).
multiply_one([C1,N1],[[C2,N2]|R],[[C,N]|T]) :-
    C is C1*C2,
    N is N1+N2,
    multiply_one([C1,N1],R,T).

multiply([],_,[]).
multiply([H1|T1],T2,T) :-
    multiply_one(H1,T2,R1),
    multiply(T1,T2,R2),
    append(R1,R2,T).
```

The second relation,$get\_degree(L_1, N, L)$, used in $transform$, raises polynomial L_1 to the Nth degree. L is the non-simplified result of this operation. Both polynomials are written in list form. For instance, query

```
get_degree([[3,2],[1,1]],2,L)
```

is answered by

```
L = [[9,4],[3,3],[3,3],[1,2]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get_degree(L1,N,L) L is L1^N
% type: L1,L - l-polynomials, N - nonnegative integer
% mode: get_degree(+,+,-)

get_degree(L1,0,[[1,0]]).
get_degree(L1,1,L1).
get_degree(L1,N,L) :-
    N > 1,
    M is N-1,
    get_degree(L1,M,L2),
    multiply(L1,L2,L).
```

Returning to the definition of *simplify*, we define the second relation, $add\_similar(L_1, L_2)$, which simplifies l-polynomial L_1 by combining terms with like degrees. For instance, query

```
add_similar([[9,4],[3,3],[3,3],[1,2]],L)
```

will be answered by

```
L = [[9,4],[6,3],[1,2]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% add_similar(L1,L2) implies that L2 is the result of
% adding up similar terms of L2.
% type: L1 and L2 are l-polynomials.
% mode: add_similar(+,?)

add_similar([],[]).
add_similar([[C1,N]|T],S) :-
    append(A,[[C2,N]|B],T),
    C is C1+C2,
    append(A,[[C,N]|B],R),
    add_similar(R,S).
```

```
add_similar([[C,N]|T],[[C,N]|ST]) :-
    not_in(N,T),
    add_similar(T,ST).
```

The relation $not\_in(N, L)$ holds if l-polynomial L does not contain a term with degree N. For instance, a query

```
not_in(2,[[1,2],[3,4]]).
```

will be answered by *no* while a query

```
not_in(1,[[1,2],[3,4]]).
```

will return *yes*.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% not_in(N,L) holds if an l-polynomial L does not contain
% a term with degree N
% type: N - non-negative integer, L l-polynomial
% mode: not_in(+,+)

not_in(_,[]).
not_in(_,[[]]).
not_in(N,[[_,M]|T]) :-
     diff(N,M),
     not_in(N,T).
```

The third relation of *simplify* is $remove\_zero(L_1, L_2)$. It holds if L_2 is obtained from L_1 by removing terms whose coefficients are 0. For instance, query

```
remove_zero([[3,4],[0,3],[0,2],[3,0]],X).
```

will be answered by

```
X = [[3,4],[3,0]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% remove_zero(L1,L2) iff list L2 is obtained from list L1
% by removing terms of the form [0,N].
% type: L1,L2  l-polynomilas
% mode: remove_zero(+,?)
```

```
remove_zero([],[]).
remove_zero([[0,_]|T],T1) :-
    remove_zero(T,T1).
remove_zero([[C,N]|T],[[C,N]|T1]) :-
    diff(C,0),
    remove_zero(T,T1).
```

The next step in our description of *simplify* is to sort terms in decreasing order of exponents. Sorting is needed for computing canonical form of the derivative, and it is easier to do it while the derivative is still in list form. Here is an example. Query

```
poly_sort([[3,4],[5,1],[2,5]],X).
```

returns

```
X = [[2, 5], [3, 4], [5, 1]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% poly_sort(L1,L2) sorts l-polynomial L1 in decreasing order
% of exponents and stores the result in L2.
% type: L1,L2  l-polynomilas
% mode: poly_sort(+,-)

poly_sort([X|Xs],Ys) :-
     poly_sort(Xs,Zs),
     insert(X,Zs,Ys).
poly_sort([],[]).

insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]) :-
     greater(Y,X),
     insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :-
     greatereq(X,Y).

greater([_,N1],[_,N2]) :-
     N1 > N2.

greatereq([_,N1],[_,N2]) :-
     N1 >= N2.
```

To complete the definition of *simplify* we define relation $back\_to\_terms(L, T)$ which translates l-polynomial L into its canonical equivalent, T. For instance, a query

```
back_to_terms([[3,2],[2,1],[1,0]],T).
```

is answered by

```
T = 3*x^2 + 2*x + 1
```

The definition of *back_to_terms* uses recursion on the length of l-polynomial L. The base case is an l-polynomial of length 1. Note, that instead of a simple rule

```
back_to_terms([[C,N]],C*x^N)
```

we have seven different rules. This happens because we are not merely translating back to terms but also do some simplification along the way. For instance, instead of translating $[[1, 1]]$ as `1*x^1`, we translate it simply as `x`. (We do need to take care of the case where $C = 0$ because we have already removed these terms.) The last two rules of the definition take care of the inductive step. Relation *append* discussed earlier in this chapter is used to split the list L into list L_1 and a singleton $[[C, N]]$. If $C > 0$ then the translation of L is the sum of the translations of L_1 and $[[C, N]]$; otherwise, it is the difference of the translations of L_1 and $[[AC, N]]$ where AC is the absolute value of C.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% back_to_terms(L,T) computes a term-representation T of L.
% L must be an l-polynomial sorted in decreasing order of
% exponents and not containing terms of the form [0,N].
% type: L l-polynomial, T polynomial.
% mode: back_to_terms(+,?)

back_to_terms([[1,1]],x).
back_to_terms([[-1,1],-x]).
back_to_terms([[C,0]],C) :-
     integer(C).
back_to_terms([[1,N]],x^N):-
      diff(N,0),
      diff(N,1).
back_to_terms([[-1,N]],-x^N):-
```

```
      diff(N,0),
      diff(N,1).
back_to_terms([[C,1]],C*x):-
      diff(C,1),
      diff(C,-1).
back_to_terms([[C,N]],C*x^N):-
     diff(N,1),
     diff(C,1),
     diff(C,-1).
back_to_terms(L,F+T) :-
     append(L1,[[C,N]],L),
     diff(L1,[]),
     C > 0,
     back_to_terms(L1,F),
     back_to_terms([[C,N]],T).
back_to_terms(L,F-T) :-
     append(L1,[[C,N]],L),
     diff(L1,[]),
     C < 0,
     abs(C,AC),
     back_to_terms(L1,F),
     back_to_terms([[AC,N]],T).
```

Finally we will need

```
eq(X,X).
diff(X,Y) :-
    \+ eq(X,Y).
abs(X,X) :-
    integer(X),
    X >= 0.
abs(X,Y) :-
    integer(X),
    X < 0,
    Y is (-1)*X.
```

This completes our program. We hope that the examples of Prolog programs given in this chapter convince the reader that it is a valuable and interesting language in its own right. We encourage those interested in becoming proficient Prolog programmers to continue their study with several of the excellent textbooks currently available on the subject.

Summary

In this chapter we introduced important reasoning algorithms based on unification and resolution, and demonstrated their use in interpreters for Prolog. The more general versions of resolution are used in automatic theorem proving and other important areas of logic-based reasoning.

The fact that resolution can be used to find elegant solutions to diverse computational problems is a consequence of a deep relationship between constructive mathematical proofs and algorithms. Recall that the proof is called **constructive** if it demonstrates existence of a mathematical object by exhibiting or providing a method for exhibiting such an object. As a result an algorithm for building an object is often simply a part of a constructive proof of the object's existence (or can be automatically extracted from such a proof). That is exactly what is done by the Prolog interpreter which seeks a constructive logical proof of the existence of an object X satisfying query $q(X)$. The proof uses the resolution and the rules of a program. The answer returned by the interpreter can be viewed as a conclusion reached by a logical reasoner about the truth or falsity of a statement given that reasoner's knowledge about the world. Similar relationships between constructive proofs and computation are used in a number of other systems (see, for instance, [79], [78], [30].

References and Further Reading

Resolution for the full first-order logic was introduced in a seminal article [100] by John Alan Robinson. The SLD resolution first appeared in [60]. SLDNF resolution first appeared in [26]. A somewhat more accurate version of the definition can be found in [7]. There is a number of very good books on Prolog. A good practical introduction can be found in [27]. As a source for more advanced material see, for instance, [105] and [31]. [91] gives a good introduction to Artificial Intelligence based on Prolog.

Exercises

1. If possible, unify the following pairs of atoms; otherwise, explain why they will not unify.

   (a) $p(X)$ and $r(Y)$.

   (b) $p(X, Y)$ and $p(a, Z)$.

(c) $p(X, X)$ and $p(a, b)$.

(d) $r(f(X), Y, g(Y))$ and $r(f(X), Z, g(X))$.

(e) $ancestor(X, Y)$ and $ancestor(bill, father(bill))$.

(f) $ancestor(X, father(X))$ and $ancestor(david, george)$.

(g) $q(X)$ and $q(a)$.

(h) $p(X, a, Y)$ and $p(Z, Z, b)$.

(i) $p(f(X))$ and $p(g(a))$.

(j) $p(X, X)$ and $p(f(Y), g(Z))$.

(k) $p(f(X, g(X), g(g(Y))))$ and $p(f(g(X), V, g(Y)))$.

2. Trace the Prolog interpreter for query $p(Y)$ to the following program:

$$p(f(X)) \leftarrow q(X), r(X).$$
$$q(a).$$
$$q(b).$$
$$r(b).$$

3. Write definitions for the following list operations in Prolog. Do not use built-in predicates that are not mentioned in this chapter.

(a) $reverse(L, R)$ where R is the list containing all elements of L in reverse order.

(b) $last(X, L)$ where X is the last element of L.

(c) $second(X, L)$ where X is the second element of L.

(d) $remove(X, L, NoX)$ where NoX is the same as list L with all occurrences of element X removed.

4. We say that a list $[A_1, \ldots, A_n]$ where all As are different represents a set $\{A_1, \ldots, A_n\}$. When we talk about set operations we simply mean to treat the lists as representations of sets. Write definitions for the following predicates in Prolog. X, Y and Z are lists. Do not use built-in predicates that are not mentioned in this chapter.

(a) $union(X, Y, Z)$ where Z is a list representing the union of sets represented by X and Y.

(b) $subset(X, Y)$ where X is a subset of Y.

(c) $intersection(X, Y, Z)$ where Z is the intersection of X and Y.

Appendix A

ASP Solver Quick-Start

What follows is a very brief, operational introduction to two currently-existing ASP solvers, `clingo` and DLV. As the field is developing rapidly, we recommend that users of this information learn about current versions of solvers. To find DLV, go to `http://www.dlvsystem.com` and click on the Products tab. To find `clingo`, go to `http://potassco.sourceforge.net/`. For quick access to the manuals, just Google `DLV manual` or `clingo manual`.

To find all answer sets of a given program, type

```
clingo 0 program_name
```

or

```
dlv -n=0 program_name
```

The `0` tells the programs to return *all* answer sets. If you omit the parameter when calling `clingo`, the program will return only one answer set; DLV will return all answer sets. Changing the number will return the corresponding number of answer sets.

Within a program
`-` is used for $\neg$,
`:-` replaces $\leftarrow$,
and `|` replaces *or* .
If running `clingo` with disjunction, you must call it with option `--shift`[1].

For a solver to be able to compute values of variables, it must be able to resolve them clearly with constants. In other words, if you use some variable

[1]This is not true disjunction, but it will do for our current purposes. There is another solver made by the same group that handles true disjunction called `claspD`. Refer to their website for more information.

X in a rule, the solver must know all possible values that it can substitute for that X. `Clingo` allows programmers to use a `#domain` statement that associates variables with a predicate. For example,

```
person(sam).
person(alice).
#domain person(X).
#domain person(Y).
#domain person(Z).
```

will allow us to use X, Y, and Z in rules and the program will know to replace them with `sam` or `alice`. DLV does not allow this shortcut. Therefore, we must make sure to put `person(X), person(Y),` and/or `person(Z)` in each rule where the types of these variables may need clarification.

Often we will want to limit what a solver will output when it prints answer sets because complete sets can be large and we may only be interested in a few predicates. If using `clingo`, it is useful to learn the `#hide` and `#show` *predicate* commands. The first hides all output predicates; the second shows the specified predicate. For example, if you had a program with predicate `mother(X,Y)`, and you included the following lines in your program:

```
#hide.
#show mother(X,Y).
```

`clingo` would output only atoms with predicate `mother` (if they are in the answer sets in the first place). Adding
`#show -mother(X,Y).`
will also show all occurrences of `-mother` in the answer sets.

If using DLV, a similar (but not identical) effect can be produced with command-line options:

```
dlv -filter=mother dlvfamily.lp
dlv -pfilter=mother dlvfamily.lp
```

The first will return both negative and positive predicates; the second will only return positive predicates.

There is also a useful program called `mkatoms`[2] that formats the output of the solvers with one predicate per line. Simply pipe the output of the solver to it.

Here is a small example:

[2] `http://marcy.cjb.net/mkatoms/`

```
%% program basic_test.lp
p(a).
-q(a).
r(a) | -r(a).
```

Running this program with `clingo`

```
clingo 0 --shift basic_test.lp | mkatoms
```

or with DLV

```
dlv basic_test.lp | mkatoms
```

will give two answer sets as follows:

```
p(a)
-q(a)
-r(a)
::endmodel
p(a)
-q(a)
r(a)
::endmodel
```

This means that the answer to query $?p(a)$ should be *true* because it is true in both answer sets, the answer to query $?q(a)$ should be $false$, because it is false in both answer sets, and the answer to query $?r(a)$ should be *unknown* because it is true in one but not the other.

Appendix B

Approximating STUDENT

In this book we use the name STUDENT for a system that can answer ASP queries. Since the field is developing very quickly, and so are a variety of useful applications, we decided to keep one name, and update only the instructions for the actual tool of choice. (Think of it as a global variable.)

Currently there are several ways to get answers to queries. For instance, you can use ASPIDE — an integrated development environment for an answer set finding system (ASP solver) called DLV. It is designed for program development, computing answer sets, and running queries. You can also use QSystem, which is an interface which allows users to query an ASP program with the help of a Prolog interpreter and either another ASP solver `clingo` or DLV (your choice). And, of course, you can just use your favorite ASP solver to compute all answer sets of a program and apply the definition of query given in Chapter 2 to these answer sets, manually. Below is a quick introduction to the first two systems. Appendix A addresses the third way.

B.1 ASPIDE

ASPIDE is an interface that can be used with DLV, not a solver in itself. Therefore, you must first download DLV which can be found at
`http://www.dlvsystem.com`.
ASPIDE can be found at
`https://www.mat.unical.it/ricca/aspide/index.html`.
The newest (beta) version can be found under the Download tab. Installation instructions are under the Documentation tab. Note that, when the ASPIDE Settings box shows up during the installation process, it should be sufficient to just set the DLV path and ignore the other settings, unless, of

course, you have a DLVDB, DLT or a profiler that you want to use. All paths can also be set later by choosing Preferences from the File menu.

To start working on a new project, select "Create Project" and give it a name. The project name will appear on the left-hand side of your screen in the "Workspace Explorer" panel. To create a new file in an existing project, select "Create new DLV file." ASPIDE will pop up a window and ask you to select the project in which your want to create the file. Once this is done, you will see an editor and will be able to type in your program. If you are planning to work on an old file in an existing project, click on the arrow to the left of the project's name causing the names of the project's files to appear below. To display a file in the editor, double-click the file name.

Once you have created and saved your file, you are ready to query your program or compute its answer sets. ASPIDE views all files in your project as comprising a single program. For instructions on querying or running a program contained in one or more selected file(s), see the section on configuring a run/query.

Running Queries

To run a query, click on the toolbar button that has a question mark. This will bring up a new window. Choose the checkbox labeled "Epistemic Mode." (The default query mode does not match the definition of query given in this book.) Your choice will be remembered for the duration of the ASPIDE session, but you will need to make this selection each time you restart the application.

Once this is done, type your query in the query input box. Make sure to end it with a question mark. To see the results, click the "Execute" button which appears next to the input box. The possible answers to a ground query are *true*, *false* or *unknown*. If the query contains variables the system will display the values of variables which make the query true and those which make the query false (which is which is clearly labeled). ASPIDE does not display values of variables for which the answer to the query is *unknown*. (You may find it useful to select "Single Run" to display only the values for which a query is true.)

The query input box also allows you to enter facts and rules followed by a query. For example, you can input

```
p.
q :- p.
q?
```

This is equivalent to asking query q to the original program expanded by the first two rules. This is useful if you want to add some information before you ask a query, but do not wish to make changes to your actual program.

Computing Answer Sets

To compute the answer sets of your program, click on the far right toolbar button. The default display will show you the output in table mode. On the left is a list of the predicates in your answer set, on the right — the parameters of the highlighted predicate (if they exist). To view an entire answer set, select "Answer Set 1" shown above the predicates (or whichever answer set you wish to see).

Configuring a Run/Query

As we mentioned, by default ASPIDE treats all the files in a project as one program. Since you will be running many small programs in this class, you will, most likely, wish to avoid creating too many projects and have the ability to just run one file of the project at a time. To do this, select the file you wish to run/query in the Workspace Explorer, right-click[1] and select "Run ▶ Run Directly" or "Run ▶ Query Directly" from the pop-up menu. Once this is done, "Run" and "Query" will remember your selection and only run the chosen file.

To create more-complex run configurations, click on the "Show Run Configuration" button found on the toolbar between the "Query" button and the "Run" button. This action brings up a window with a variety of options. Pressing the green plus icon will allow you to add files to your configuration; highlighting a file and pressing the red minus icon will remove the chosen file from the configuration. The selected files will from then on be treated as one program by the "Run" and "Query" commands. Note that deselecting all files will result in the project being treated as a unit once again.

Importing Projects and Files

If you already have a file(s) you wish to run or query, you can choose to import it into ASPIDE. If you wish to import files into an existing project, choose "Import DLV File" from the list in the gray panel on the main screen. A pop-up allows you to select the project you want and browse for your file.

[1] For a one-button mouse, use Ctrl-Click.

ASPIDE makes a copy of the file in the Workspace directory you created at installation.

You may also choose to create a new project from an entire directory of files. In that case, choose "Import Project" and select the folder you wish to import. The name of the folder becomes the name of a new project. ASPIDE will expect you to work with the files through its interface, so it is best, at least for now, not to modify the Workspace directory in other ways.

There is a lot more you can do with ASPIDE, and more than one way to do things, but we hope this introduction will allow you to run and query the examples and exercises given in this book.

B.2 QSystem

QSystem can be found at
`http://code.google.com/p/student-qsystem/`.
Installation and usage instructions can be found in the Readme under the Wiki tab. It calls `clingo` or DLV to find the answer sets of the given program, formats them into facts Prolog can understand, adds special query-answering rules to the program, and feeds this new program into Prolog. Therefore, if everything is correct, you will find yourself in the Prolog interpreter, able to query your program. Queries to QSystem must be entered as a parameter to the query predicate. For example, if you wish to ask query $child(sam, john)$, you will need to type

```
query(child(sam,john),X).
```

at the Prolog prompt. The answer to the query will be assigned to variable X.

If you are unfamiliar with the Prolog interpreter, it is helpful to know that typing `halt.` will get you out of it. Also, there may be more than one answer to your query, so Prolog waits for you to type either a semicolon (to get another answer) or RETURN (to go back to the prompt).

Currently, if we wish QSystem to answer queries correctly, we must add some statements to our programs. For every predicate in the program, a new predicate must be made that defines the sorts that are allowed in its parameters. For example, if we have program

```
c(a).
c(b).
p(X,Y) :- c(X), c(Y).
```

we must add statements

```
my_atom(c(X)) :- c(X).
my_atom(p(X,Y)) :- c(X), c(Y).
```

to it.

As another example, let's look at `family.lp` from Chapter 1. If our original program is

```
person(john).
person(sam).
person(alice).
person(bill).
#domain person(X).
#domain person(Y).

father(john,sam).  %% John is the father of Sam.
father(john,bill).
mother(alice,bill).
mother(alice,sam).
gender(john,male).
gender(bill,male).
gender(sam,male).
gender(alice,female).

%% X is a parent of Y if X is a father or mother of Y.
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

%% X is a child of Y if Y is the parent of X.
child(X,Y) :- parent(Y,X).
```

then we must add statements

```
gender(male).
gender(female).
my_atom(person(X)) :-
       person(X).
my_atom(father(X,Y)) :-
       person(X),
       person(Y).
my_atom(mother(X,Y)) :-
```

```
       person(X),
       person(Y).
my_atom(gender(X,Y)) :-
       person(X),
       gender(Y).
my_atom(parent(X,Y)) :-
       person(X),
       person(Y).
my_atom(child(X,Y)) :-
       person(X),
       person(Y).
```

to make sure that `qsystem` answers queries correctly. This is, of course, inconvenient, but we hope it is not difficult for our small examples.

Bibliography

[1] DLV web page.

[2] Smodels web page.

[3] S. Abiteboul, R. Hall, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[4] J. J. Alferes and L. M. Pereira. *Reasoning with Logic Programming*. Springer, Berlin, 1996.

[5] Krzysztof Apt, Howard Blair, and Adrian Walker. *Towards a theory of declarative knowledge*, pages 89–148. Foundations of deductive databases and logic programming. Morgan Kaufmann, 1988.

[6] Krzysztof R. Apt and Roland Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19:9–71, 1994.

[7] Krzysztof R. Apt and Kees Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 28:177–190, 1994.

[8] Marcello Balduccini. CR-MODELS: An inference engine for CR-Prolog. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the 9th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'07)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 18–30. Springer, 2007.

[9] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(4–5):425–461, Jul 2003.

[10] Marcello Balduccini and Michael Gelfond. The AAA architecture: An overview. In *AAAI Spring Symposium 2008 on Architectures for Intelligent Theory-Based Agents (AITA08)*, 2008.

[11] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. Answer set based design of knowledge systems. *Annals of Mathematics and Artificial Intelligence*, 47:183–219, 2006.

[12] Marcello Balduccini and Sara Girotto. Formalizing Psychological Knowledge in Answer Set Programming. In *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR2010)*, May 2010.

[13] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, Jan 2003.

[14] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19(20):73–148, 1994.

[15] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In *Workshop on Logic-Based Artificial Intelligence*. Kluwer Academic Publishers, 2000.

[16] Chitta Baral, Michael Gelfond, and J. Nelson Rushton. Probabilistic reasoning with answer sets. *TPLP*, 9(1):57–144, 2009.

[17] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription in stratied logic programming. In *Proceedings of the Logic in Computer Science Conference*, pages 89–97, 1987.

[18] N. Bidoit and C. Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 79(1):86–112, 1991.

[19] Georg Boenn, Martin Brain, Marina De Vos, and John Fitch. Automatic music composition using answer set programming. *TPLP*, 11(2-3):397–427, 2011.

[20] G. Boole. *An Investigation of the Laws of Thought: On which are Founded the Mathematical Theories of Logic and Probabilities*. George Boole's collected logical works. Walton and Maberly, 1854.

[21] Ronald Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[22] Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109:297–356, 1998.

[23] Gerhard Brewka, Ilkka Niemela, and Miroslaw Truszczynski. Preferences and nonmonotonic reasoning. *AI Magazine*, 29(4):69–78, 2008.

[24] Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.

[25] Weidong Chen, Terrance Swift, and David S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–201, 1995.

[26] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[27] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (4. ed.)*. Springer, 1994.

[28] Jacques Cohen. A view of the origins and development of Prolog. *Communications of the ACM*, 31(1):26–36, January 1988.

[29] Alain Colmerauer and Philippe Roussel. The Birth of Prolog. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of Programming Languages II*, pages 331–367. ACM, New York, NY, USA, 1996.

[30] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing mathematics with the nuprl proof development system, 1986.

[31] Michael A. Covington, Donald Nute, and Andre Vellino. *Prolog Programming in Depth*. Prentice-Hall, 1997.

[32] McDermott D., Ghallab M., Howe A. Knoblock C., Ram A, Veloso M., Weld D., and Wilkins D. Pddl – the planning domain definition language. Tech. Rep. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[33] J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20:308–334, 2004.

[34] James P. Delgrande, Torsten Schaub, and Hans Tompits. A Framework for Compiling Preferences in Logic Programs. *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(2):129–187, 2003.

[35] Yannis Dimopoulos, Jana Koehler, and Bernhard Nebel. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 169–181, 1997.

[36] Jürgen Dix. Classifying semantics of logic programs (extended abstract). In *LPNMR*, pages 166–180, 1991.

[37] Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors. *Logic Programs with Consistency-Restoring Rules*, AAAI 2003 Spring Symposium Series, Mar 2003.

[38] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[39] François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1(1):51–60, 1994.

[40] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *IJCAI*, pages 372–379, 2007.

[41] Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

[42] R. Fikes and N. Nillson. Strips: A new approach to the application of theorem proving to problem solving. Technical Note 43R, AI Center, SRI international, 1971.

[43] R. Fitzpatrick. *Euclid's Elements*. University of Texas at Austin, Institute for Fusion Studies Department of Physics, 2007.

[44] G. Frege. *Begriffsschrift, a formal language, modeled upon that of arithmetic, for pure thought, In From Frege to Gödel: A Source Book in Mathematical Logic*. Harward University Press, 2002 (1879).

[45] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Guide to gringo, clasp, clingo, and iclingo.

[46] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*, volume 6 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool, 2012.

[47] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[48] Michael Gelfond. On stratified autoepistemic theories. In *Proceedings of Sixth National Conference on Artificial Intelligence*, pages 207–212, 1987.

[49] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.

[50] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[51] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17(2/3&4):301–321, 1993.

[52] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3, 1998.

[53] Michael Gelfond and Tran Cao Son. Reasoning with Prioritized Defaults. In *Third International Workshop, LPKR'97*, volume 1471 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 164–224, Oct 1997.

[54] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning : Theory and Practice*. Morgan Kaufmann, 2004.

[55] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[56] D. Hilbert. *Foundations of Geometry*. Open Court, 1980 (1899).

[57] R Hilborn and L Mange. *The Ecological Detective*. Priceton University Press, 1997.

[58] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[59] Robert Kowalski. Using meta-logic to reconcile reactive with rational agents. In *Meta-logics and Logic Programming*, pages 227–242. MIT Press, 1995.

[60] Robert Kowalski and Donald Kuehnm. Linear resolution with selection function. *Artificial Intelligence*, 2(3,4):227–260, 1971.

[61] Robert A. Kowalski. *Logic for problem solving*. North-Holland, 1979.

[62] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, January 1988.

[63] G. Leibniz. *Leibniz Selection*. Charles Scribner's Sons, 1951.

[64] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and computation*, 135:69–112, 1997.

[65] Y. Lierler and M. Marateo. CMODELS web page.

[66] V. Lifschitz. Computing circumscription. In *Proceedings of the International Joint Conference on Artificial Intelligence*, IJCAI'85, pages 121–127. Morgan Kaufmann Publishers Inc., 1985.

[67] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[68] Vladimir Lifschitz, Leora Morgenstern, and David Plaisted. Knowledge representation and classical logic. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 3–88. Elsevier, 2008.

[69] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, Berlin, 2nd edition, 1987.

[70] Marco Manna, Ermelinda Oro, Massimo Ruffolo, Mario Alviano, and Nicola Leone. The HiLeX system for semantic information extraction. *T. Large-Scale Data- and Knowledge-Centered Systems*, 5:91–125, 2012.

[71] Victor W. Marek and Miroslaw Truszczynski. *Nonmonotonic logics; context dependent reasoning*. Springer Verlag, Berlin, 1993.

[72] Victor W. Marek and Miroslaw Truszczynski. *Stable models and an alternative logic programming paradigm*, pages 375–398. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin, 1999.

[73] W. Marek and V. S. Subrahmanian. The relationship between logic program semantics and nonmonotonic reasoning. In *ICLP*, pages 600–617, 1989.

[74] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1-2):27–39, 1980.

[75] John McCarthy. *Formalizing common sense, papers by John McCarthy edited by V. Lifschitz*. Ablex, 1990.

[76] John McCarthy. Elaboration Tolerance, 1998.

[77] Drew V. McDermott and Jon Doyle. Non-monotonic logic I. *Artificial Intelligence*, 13(1-2):41–72, 1980.

[78] P. Miglioli, U. Moscato, and M. Ornaghi. Pap: a logic programming system based on a constructive logic. In *Foundations of Logic and Functional Programming*, volume 306 of *Lecture Notes in Computer Science*, pages 141–156, 1988.

[79] G. Mints and E. Tyugu. The programming system priz. *J. Symb. Comput.*, 5(3):359–375, June 1988.

[80] Robert C. Moore. Semantical considerations on nonmonotonic logic. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 272–279. Morgan Kaufmann, Aug 1983.

[81] A. Nerode and R.A. Shore. *Logic for Applications*. Graduate Texts in Computer Science. Springer-Verlag GmbH, 1997.

[82] Ilkka Niemela. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3–4):241–273, 1999.

[83] Ilkka Niemela and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 420–429, 1997.

[84] Ilkka Niemela, Patrik Simons, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, Jun 2002.

[85] Nils Nillson. Shakey the robot. Technical Report 323, AI Center, SRI international, 1984.

[86] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An A-Prolog decision support system for the Space Shuttle. In Alessandro Provetti and Tran Cao Son, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, AAAI 2001 Spring Symposium Series, Mar 2001.

[87] David Pearce. A new logical characterisation of stable models and answer sets. In *In Proc. of NMELP 96, LNCS 1216*, pages 57–70. Springer, 1997.

[88] David Pearce. Equilibrium logic. *Ann. Math. Artif. Intell.*, 47(1-2):3–41, 2006.

[89] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 2000.

[90] Edwin P. D. Pednault. Adl and the state-transition model of action. *J. Log. Comput.*, 4(5):467–512, 1994.

[91] David Poole, Alan Mackworth, and Randy Goebel. *Computational Inteligence: A Logical Approach*. Oxford University Press, 1998.

[92] David Poole and Alan K. Mackworth. *Artificial Intelligence - Foundations of Computational Agents*. Cambridge University Press, 2010.

[93] Halina Przymusinska and Teodor Przymusinski. Weakly Stratified Logic Programs. *Fundamenta Informaticae*, XIII:51–65, 1990.

[94] Teodor Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[95] Teodor C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1995.

[96] Georgeff Rao. Modeling rational agents within a BDI-architecture. In *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann Publishers, 1991.

[97] Raymond Reiter. *On Closed World Data Bases*, pages 119–140. Logic and Data Bases. Plenum Press, 1978.

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

[99] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-building with answer set programming in the gioia-tauro seaport. *TPLP*, 12(3):361–381, 2012.

[100] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[101] Kenneth Ross. A procedural semantics for well founded negation in logic programs. In *Journal of Logic Programming*, pages 22–33, 1989.

[102] Chiaki Sakama and Katsumi Inoue. Prioritized Logic Programming and its Application to Commonsense Reasoning. *Artificial Intelligence*, 123:185–222, 2000.

[103] Patrik Simons. *Computing Stable Models*, Oct 1996.

[104] Timo Soininen and Ilkka Niemela. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, May 1999.

[105] Leon Sterling and Ehud Shapiro. *The Art of Prolog (2nd ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[106] V. S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In *Proceedings of ICLP-95*, pages 233–247. 1995.

[107] A. Tarski. *Introduction to Logic and the Methodology of Deductive Sciences*. Dover, 1995 (1941).

[108] D. S. Touretzky. *The Mathematics of Inheritance Systems*. Research Notes in Artificial Intelligence. Morgan Kaufmann, 1986.

[109] Bonnie Traylor and Michael Gelfond. Representing null values in logic programming. In Anil Nerode and Yuri Matiyasevich, editors, *LFCS*, volume 813 of *Lecture Notes in Computer Science*. Springer, 1994.

[110] Phan Huy Tu, Tran Cao Son, Michael Gelfond, and A. Ricardo Morales. Approximation of action theories and its application to conformant planning. *Artificial Intelligence*, 175(1):79–119, 2011.

[111] Hudson Turner. Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *Journal of Logic Programming*, 31(1-3):245–298, Jun 1997.

[112] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.

[113] A. Whitehead and B. Russell. *Principia Mathematica, 3 vols*. Cambridge University Press, 1910, 1912, 1913.

[114] Michael Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.

[115] C. Zaniolo. Design and implementaion of a logic based language for data intensive applications. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1666–1687, 1988.

Index

$\Pi(\mathcal{SD})$ (ASP encoding of $\mathcal{AL}$ system description), 187
Π^S (reduct), 38
$\Pi_c(\mathcal{SD})$, 183
Σ (signature), 23
σ_0 (initial state), 177
σ_n (final state), 178
$\mathcal{SD}$ (system description), 183
$\mathcal{T}(\mathcal{SD})$ (transition diagram), 183

abductive support, **117**, 117–120
aces in succession example, 280–282
action languages, **181**
agent, **12**
agent action, **243**
agent architecture, **12**
agent loop, **13**
$\mathcal{AL}$, 181–203
  statements of, 182
algorithm
  for answering queries, 164–165
  for computing answer sets, 147–166
  for computing answer sets of disjunctive programs, 164
  for computing models of propositonal formulas, 147–152
  resolution, **301**
  SLD resolution, 306–312
  SLDNF resolution, 312–314
  translation from $\mathcal{AL}$ to ASP, 187–189
  unification, **301**, 302–306
algorithmic language, **14**
ancestor example, 81–84
answer set, **23**
  formal definition, Part I, **34**
  formal definition, Part II, **38**
  intuition for, 28–29
  of CR-Prolog, **117**
answer set planning, 214
Answer Set Prolog (ASP), **15**, 23–47
  semantics, 28–41
  syntax, 23–28
answer set solver, **41**, 127–129
arithmetic term, 24, 27
arity, **23**
ASP program, **26**
  properties of, 43
ASP solver, *see* answer set solver
atom, **25**
  ground, **26**
attributes (in P-log), **271**
autoepistemic logic, 57, 59–60
Awareness Axiom, **252**, 265
axiomatic method, 17–19

backtracking, 148, 310, 312
Bayesian squirrel example, 291–294
biased dice example, 276–277
blocks world domain, **169**
  concurrent actions in, 198–200
  first try in ASP, 168–177
  modeling with $\mathcal{AL}$, 195–198

planning, 212–217
body (of a rule), **27**
briefcase domain, 190–195
recorded history in, 246–247

Cancellation Axiom, **101**
special case, 105
causal law, **173**, 178, 182, 188
causal probability, **276**, 277, 278
causal probability statement, **276**
choice rule, 128, **133**, 139, 143, 210
circuit diagnostic example, 244–245, 248–250, 254–258
circumscription, 57–59
Clark's completion, **63**, 63–66
Clark, Keith L., 63
Clasp, **128**
ClaspD, **128**, 199
classical planning, **209**
Clingo, **129**, 337
Closed World Assumption (CWA), 33, 44, 79, 99
Colmerauer, Alain, 18
commonsense knowledge, 75, 80, 87, 88, 140, 141
commonsense reasoning, 32, 80, 99, 266–268
complementary, **26**
concurrent planning, 233–234
Cons(), 149
*Cons*1(), 154
*Cons*2(), 160, 161
consequence
ASP, **32**
computation for ASP, 154–163
computation for FOL, 149–152
FOL, 55
well-founded, **67**
consistent
set of domain literals, 182
set of e-literals, 153
set of ground literals, **34**
constraint, **27**, 46
example, 31, 38
removing, 46
Contingency Axiom, **116**, 118
cowardly students example, 103–104
CR-Prolog, 115–120, 234–236, 259–260, 294–297
cr-rule, **116**
CWA, *see* Closed World Assumption

Datalog, 19
Davis-Putnam procedure, 147–152
declarative language, **14**
default, **99**, 99–120
Contingency Axiom, **116**
exception to, 100–120
priorities between, 107–111, 114
Reiter's, 60
default logic, 57, 60–62
default negation, **26**, 62, 63, 65
example, 31, 38
removal, 38
default rule, 101
default theory, **61**
defined fluent, **177**, 187, 188, 215
definite program, **157**, 302
dependency graph, **45**
derivatives example (Prolog), 325–334
diagnostic problem, **243**
example, 244–245
diagnostics, 243–262
dice example, 273–276
DLV, **128**, 337
domain literal, **182**
complete set of, **182**
consistent set of, **182**
domain properties, **182**
Doyle, Jon, 57

dynamic domain, 167
  diagnostics, 243–262
  modeling, 167–204
  planning, 209–238
dynamic range, **272**
  effect on possible worlds, 280–282

elaboration tolerance, **16**, 93–94, 169, 200
electrical circuit example, 84–88
e-literal, *see* extended literal
encoding
  of a system description, **187**
entailment, *see also* consequence
  ASP, **32**
  FOL, 55
  minimal (circumpscription), 57
  recorded history, **246**
epistemic disjunction, **26**
  example, 29–30, 36–37
Euclid, 17
exception to a default, 100–120
  given complete information, 105
  indirect, 115–120
  strong, **101**
  weak, **101**
executability condition, 175, 182, 189
exogenous action, **243**
expert knowledge, 75, 87
explanation, **244**
  best, **251**
  computation of, 251–259
  generation rule, 253
  minimal, 259–260
  possible, **248**
explanation generation rule, 253
extended literal, **27**, 153
extension of a default theory, **61**

fact, **27**
Fages, François, 66
family example, 14–16, 76–79
`findall` (Prolog built-in), 320
first-order logic, 53–56
fixpoint, **66**
floundering, 314, 319
fluent, **171**
  defined, **177**, 187, 188, 215
  inertial, **177**, 187
fluent dependency graph, **185**
fluent literal, **182**
FOL, *see* first-order logic
Frame Problem, **178**

goal, **209**, 210
Gringo, **128**
ground
  atom, **26**
  instantiation, **27**
  literal, **26**
  term, **24**
grounding, **27**

Hamiltonian paths example, 129–134
head (of a rule), **27**
heuristic for ASP solvers, 163
heuristics for planning, 229–233
hierarchical information, 88–94, 111–115
holds(), 171
horizon (of a planning problem), **209**
hpd(), 246, 260–261

igniting the burner example, 217–222
inconsistent program, **39**
Inertia Axiom, **174**
  encoding, 189
  Frame Problem, 178
inertial fluent, **177**, 187
inheritance hierarchy, **89**, 88–94
  with defaults, 111–115

interpretation
first-order, 54
logic programming, 153
partial (logic programming), 153
partial (propositional), 148
propositional, 149
is_a(), 91, 113–115
IsAnswerSet(), 156
is_subclass(), 90, 113–114

jungle example, 268–271

knowledge base, 14–16, 26, 75–94
with null values, 105–106
Kowalski, Robert, 18

l-polynomial, **327**
law of the exclusive middle, 26
LB(), 154
Least(), 157
Leibniz Dream, 18
Leibniz, Gottfried Wilhelm, 18
level mapping, **43**, 43–44
list (in Prolog), 317–321
literal, **26**
domain, **182**
extended, **27**
ground, **26**
negative, **26**
removing, 46
locally stratified, **43**, 46
logic program, *see* ASP program
logic-based approach to agent design, **14**, 13–19
logic-based approach to AI, **18**
lower bound, 154
Lparse, 128

McCarthy, John, 18, 57, 178
McDermott, Drew, 57
member(), 91–92, 112–114
mgu, *see* most general unifier
minimal explanation, 259–260
minimal plan, 234–236
`minimize` statement, 236, 260
missionaries and cannibals example, 222–229
`mkatoms`, 338
mode (of a Prolog predicate), 318
model
FOL, 55
mathematical model of an intelligent agent, 12
minimal (FOL), 57
of a recorded history, **246**
probabilistic, 266
propositional, 149
supported, 64
well-founded, 66
monotonic, **32**
monotonicity, **32**
Monty Hall example, 282–287
Moore, Robert, 57
most general unifier (mgu), **302**
mystery puzzle, 140–143

Narwhal example, *see* submarine example
natural language (translation into ASP), 41–43
negation as failure, *see* default negation
negation as finite failure, 65, **313**
negative information, 89, 103, 106
negative literal, **26**
removing, 46
nlp, *see* normal logic program
non-determinism in $\mathcal{AL}$, 200–201
non-monotonic, **32**, 62, 101
non-monotonic logics, 32, 57–62
normal logic program, **62**, 67, 301

tight, **66**
null value, 105–106

obs(), 246, 260–261
observation, **246**
occurs(), 171
orphan example, 79
with incompleteness, 111
over-specification, 200

P-log, **268**, 265–299
parts inventory example (Prolog), 321–325
plan, **209**
planning, 209–238
concurrent, 233–234
heuristics, 229–233
minimimal plan, 234–236
planning problem, **209**
possible world, **267**
pr-atom, **276**
predicate, 23
preference relation (CR-Prolog), **116**
Principle of Indifference, **268**
probabilistic measure, **267**
probabilistic model, **266**
probabilistic reasoning, 16, 265–299
probability function, **267**
probability of a proposition, **273**
Prolog, 18, 301–336
interpreter, 302–314
programming, 314–334
Przymusinski, Teodor, 69

query, **32**
ASP answer to, **32**
SLD resolution answer to, **306**
SLDNF resolution answer to, **312**
Query(), 164
query rule, **307**

Ramification Problem, **179**
random attribute, **269**
random selection rule, **269**
Rationality Principle, **29**, 31, 47
Reality Check Axioms, **252**, 260
recorded history, **245**
briefcase domain, 246–247
circuit diagnosis example, 249
semantics, **246**
syntax, **246**
recursive definition, **83**
reduct, **38**, 39
reification, **90**, 90–94, 115, 171
Reiter's default theory, **61**
Reiter, Raymond, 57
resolution, 56, **301**, 314
SLD, 18, **302**, 306–312
SLDNF, 62, 65, 69, **302**, 312–314
SLS, 69
resolvent, **308**
Ross, Kenneth A., 66
Roussel, Philippe, 18
rule (of ASP), **26**

Sat(), 149
satisfiability (of a rule), **28**
satisfiability solver, **134**, 134–135, 147–152
schema, 191
Schlipf, John S., 66
signature, **23**
$\mathcal{AL}$, 182, 187
FOL, 148
P-log, 268
sorted, 25
simple planning module, **210**, 210–211
SLD derivation, **309**
SLD resolution, 18, **302**, 306–312

SLD resolution inference rule, **308**
SLDNF resolution, 62, 65, 69, **302**, 312–314
SLS resolution, 69
Smodels, **128**
Solver(), 152
*Solver*1(), 152, 153
*Solver*2(), 152
sort, **24**
sorted signature, **24**, 25
specificity principle, **114**, 113–115
spider bite example, 287–291
stable expansion, **59**
stable model, 23, *see also* answer set
stable model semantics, 66
  connection to autoepistemic logic, 60
state
  of a transition diagram, **184**, 183–187
state constraint, 173, 182, 188
static domains, 167
static literal, **182**
statics, **171**
stratification, 43–46
stratified, **45**, 60, 69
  locally, **43**, 46
strong exception, **101**
subclass relation, 89–94, 113–114
submarine example, 88–94
substitution, **302**
Sudoku puzzle, 135–140
supported model, **64**
symptom, **243**, **248**
symptom checking, 252
system configuration, **247**
system description, **182**, 182–201

Tarski, Alfred, 18
temporal projection, **202**
term, **24**
  ground, **24**
tight normal logic program, **66**
Touretzky, David S., 114
transition, 171, **190**, 177–203
transition diagram, 177–203
  non-deterministic, 200–201
Tweety example, 113–115

UB(), 160
uncaring John example, 100–103
unification, **301**, 302–306
unifier, **302**
Unique-Name Assumption (UNA), **78**
unit propagation, 150
upper bound, 160

Van Gelder, Allen, 66
variant, **309**

wandering robot example, 294–297
weak acyclicity, **186**
weak exception, **101**
well-founded
  consequence, 67
  model, 66, **67**
  semantics, 66–69
  system description, **185**

XSB, 69