Knowledge Representation and Reasoning

Logic meets Probability Theory

Lecture Notes 2012–2013
BSc Programme on AI, Nijmegen

Peter Lucas, Martijn van Otterlo and Arjen Hommersom
iCIS and Donders Institute, Radboud University Nijmegen
Email: {peterl,arjenh}@cs.ru.nl, m.vanotterlo@donders.ru.nl

6th November, 2012

Preface to these notes

This set of lecture notes gives a brief summary of the basic ideas and principles underlying the lectures given in the course *Knowledge Representation and Reasoning*. The basic idea underlying the course is that in AI you need to be able to *precisely represent* knowledge in order to explore that knowledge to solve problems. This is because, in contrast to humans, computers can only manipulate knowledge that has a precise syntax and semantics, and even if the requirements are somewhat loosened, there must be still some underlying language that is precise.

Although knowledge representation can be done using different formal languages, the two languages that have become dominant in AI in the course of time are *predicate logic* and *probability theory*. In contrast to the usual way logic and probability theory are used in mathematics and computing science, AI researchers have a tendency to be *creative* with logic and probability theory, and play a bit with these languages in order to better understand the problem—how to represent particular types of knowledge and to reason with it—they are tackling. To be able to use logic and probability theory in this fashion, an AI researcher needs a really good *understand* of these languages; it is not enough only being able to reproduce simple facts about logic and probability theory. In the end, the languages are used as vehicles to represent *knowledge*, and so the semantics, i.e., what one is trying to say, is of utmost importance. Semantics is the *central theme* that is visible in all the lectures offered in the course.

The structure of the lectures is as follows:

- Introduction (Chapter 1): relationship between cognitive aspects of knowledge and formal and computer-based representation. Why are formal languages crucial in this context and what are relevant properties?
- Logic programming en Prolog (Chapter 2): knowledge as computer programs.
- Description logics and frames (Chapter 3): relationship between knowledge representation and current world-wide efforts of making human knowledge available through the Semantic Web.
- Model-based reasoning (Chapter 4): representation and use of models of structure and behaviour to solve problems, in particular diagnostic problems.
- Reasoning and decision-making under uncertainty (Chapter 5): AI methods for representing and reasoning with the uncertainty met in many real-world problems. This includes preferences and making decisions. (Most topics here are a recap from the previous two AI courses).
- Probabilistic logic (Chapter 6): here we are back to merging logical and probabilistic reasoning (as in the early days of AI), but now in a single powerful and mathematically sound framework. This chapter reflects recent advances in knowledge representation and reasoning in AI.
- Reasoning in dynamic worlds (Chapter 7): here we take a look at how one can specify logical models for changing information, such as for action planning. We take a look at different ways to represent dynamics, and how to create high-level decision policies in logic. Probability can sometimes be added in an easy way, for example using AILog.

- Applications of (probabilistic) logic in AI (Chapter 8): logic was used much in the early days in AI, but was sometimes neglected somewhat because (among other things) it was hard to add probabilities or learning in many settings. The last decade we are seeing many applications using probabilistic logic for robotics, language processing and computer vision. In this lecture we take a look at image interpretation, or computer vision.

Note that the lecture notes complement the slides used in the lectures. The slides often contain other examples and sometimes also more detail. Thus, you need to study the slides together with the lecture notes, and not only the lecture notes!

Peter Lucas, Martijn van Otterlo and Arjen Hommersom
Nijmegen, 6th November, 2012

Contents

Preface to these notes **i**

1 Lecture 1 – Introduction **1**
1.1 Aims of the course . . . 1
1.2 Knowledge representation requirements . . . 2
1.3 Logic as a knowledge representation language . . . 3
1.3.1 Horn-clause logic . . . 3
1.3.2 Objects, attributes and values . . . 5
1.4 Reasoning with uncertainty . . . 6
1.5 Conclusions . . . 7

2 Lecture 2-4 – Logic Programming **9**
2.1 Introduction . . . 9
2.2 Logic programming . . . 10
2.3 SLD resolution: a special form of resolution . . . 11
2.4 Programming in Prolog . . . 17
2.4.1 The declarative semantics . . . 18
2.4.2 The procedural semantics and the interpreter . . . 21
2.5 Overview of the Prolog language . . . 28
2.5.1 Reading in programs . . . 28
2.5.2 Input and output . . . 29
2.5.3 Arithmetical predicates . . . 29
2.5.4 Examining instantiations . . . 31
2.5.5 Controlling backtracking . . . 32
2.5.6 Manipulation of the database . . . 35
2.5.7 Manipulation of terms . . . 36
2.6 Suggested reading and available resources . . . 38

3 Lecture 5 – Description Logics and Frames **39**
3.1 Introduction . . . 39
3.2 Description logics . . . 39
3.2.1 Knowledge servers . . . 39
3.2.2 Basics of description logics . . . 40
3.2.3 Meaning of description logics . . . 42
3.2.4 Reasoning . . . 43
3.3 Frames . . . 44

3.3.1 Definition . . . 44
3.3.2 Semantics . . . 45
3.3.3 Relationship with description logic . . . 47
3.3.4 Reasoning — inheritance . . . 48

4 Lectures 6-8 – Model-based Reasoning **55**
4.1 Introduction . . . 55
4.2 Concistency-based diagnosis . . . 56
4.3 Abductive diagnosis . . . 59
4.3.1 Logical abduction . . . 59
4.3.2 Set-covering theory of diagnosis . . . 65
4.4 Non-monotonic reasoning . . . 70
4.5 The AILog system . . . 72

5 Lecture 9 – Reasoning and Decision Making under Uncertainty **75**
5.1 Introduction . . . 75
5.2 Rule-based uncertainty knowledge . . . 75
5.3 Probabilistic graphical models . . . 76
5.4 Towards Decision Making . . . 80
5.5 Preferences and utilities . . . 81
5.6 Decision problems . . . 82
5.7 Optimal policies . . . 84

6 Lecture 10 – Probabilistic logic **87**
6.1 Probabilistic logic based on logical abduction . . . 87
6.2 Probabilistic reasoning in AILog . . . 91

7 Lecture 11 – Logic for Dynamic Worlds **93**
7.1 States and Operators . . . 93
7.2 STRIPS . . . 97
7.2.1 Situation calculus . . . 99

8 Lecture 12 – AI Applications of Probabilistic Logic **100**
8.0.2 Vision as constraint satisfaction . . . 100
8.0.3 Vision using probabilistic explanations . . . 100

A Logic and Resolution **104**
A.1 Propositional logic . . . 105
A.2 First-order predicate logic . . . 110
A.3 Clausal form of logic . . . 115
A.4 Reasoning in logic: inference rules . . . 119
A.5 Resolution and propositional logic . . . 121
A.6 Resolution and first-order predicate logic . . . 124
A.6.1 Substitution and unification . . . 124
A.6.2 Resolution . . . 127
A.7 Resolution strategies . . . 130
A.8 Applying logic for building intelligent systems . . . 131
A.8.1 Reasoning with equality and ordering predicates . . . 132

Exercises 135

Chapter 1

Lecture 1 – Introduction

Knowledge representation and reasoning is about establishing a relationship between human knowledge and its representation, by means of formal languages, within the computer.

1.1 Aims of the course

Although much human knowledge can be conveyed by natural language and by informal diagrams, artificial intelligence (AI) researchers and practitioners wish not only to represent knowledge, but also to automatically *reason* with the knowledge. Despite many attempts, in fact especially by AI researchers, no automatic method is currently available to reason with natural language and informal diagrams. Few people nowadays believe it will ever be possible to develop such methods. Instead, AI researchers have focused on developing *formal* methods to represent and reason with knowledge. Formal languages have a precise, i.e., mathematical, syntax and semantics.

Since the inception of AI as a separate research discipline, researchers have proposed many different languages. In time, logical languages have become the most dominant, mainly because they do have a precise semantics and associated reasoning methods. Many of these languages extend or modify standard logic. Of course, the precise nature of logical languages makes it sometimes difficult to represent objects in the real world, also called *natural kinds*, as these are often incompletely understood. However, even then logical language allow approximating what is known. There is also much AI research that focuses on the representation of human-made artifacts, such as devices and equipment. Although representing such artifacts is typically easier, there are many facets of machines which are also incompletely understood. In that sense, the difference between natural kinds and human-made artifacts is smaller than most people think. Figure 1.1 summarises the relationship between knowledge about real objects, knowledge representation and reasoning.

Often a distinction is made between the *knowledge level* and *symbol level* of knowledge. The knowledge level is about what the knowledge is about, i.e., its content. The symbol level concerns the representation of knowledge using a formal language. Typically, some of the content of the knowledge may be lost in translating the knowledge from the knowledge to the symbol level. This need not always affect the quality of the conclusions that can be drawn from the symbol-level representation. Although such loss of content is unavoidable in general, one should be aware of it.

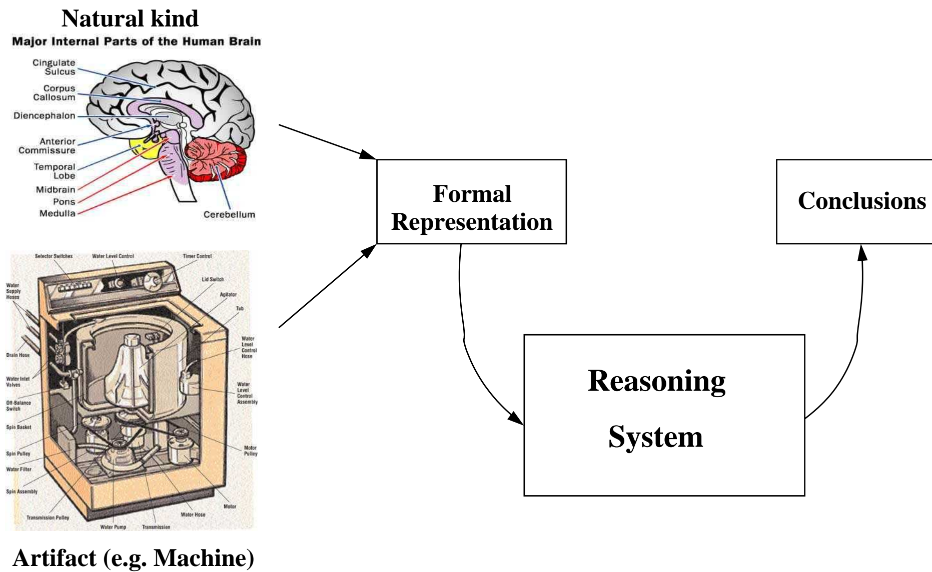

Figure 1.1: Role of formal representation.

Prolog is a logical programming language, and has characteristics that renders it very close to knowledge representation and reasoning systems. Because of this closeness, it is relatively easy to implement knowledge systems in Prolog. This explains why Prolog has been taken as the primary implementation language in the course.

The aims of the course are:

- To understand the relationship between knowledge level and symbol level, which is mainly reflected by understanding how to translate particular intuitive concepts into logic and probability theory (directly or via intermediate languages);
- Be familiar with the most important logical and probabilistic methods to represent and reason with knowledge;
- Being able to develop simple reasoning systems using Prolog.

1.2 Knowledge representation requirements

For a knowledge-representation language to be *practical*, there are particular requirements that must be fulfilled:

- It must have a precise semantics (i.e., in terms of mathematical objects such as sets, numbers, etc.);
- There must be information available about the relationship between *expressive power*, i.e., what one can express in the language, and computational complexity in terms of required running time and space of the associated reasoning algorithms.

For languages such as logic and probability theory, both having a precise semantics, much is known about the relationship between expressive power and computational complexity. Knowledge about this relationship is important, because unrestricted logical and probabilistic

languages have very unfavourable properties in terms of decidability (whether the system is able to produce guaranteed output in finite time), time complexity (whether the system is able to efficiently compute a conclusion, where 'efficient' is normally understood as in at least polynomial time). Many knowledge representation languages are NP hard, i.e., require in the worst case exponential time for computing answers to queries. It appears that many AI problems are of such generality that they share these unfavourable characteristics.

1.3 Logic as a knowledge representation language

As an example, consider standard first-order logic as a language that we use to specify a knowledge base KB. It is known that first-order predicate logic (check your book on logic, Appendix A, or [8]) is undecidable. However, when it is known that KB is unsatisfiable i.e., $\text{KB} \vDash \bot$ holds, then $\text{KB} \vdash \bot$ is decidable. As a consequence, first-order logic is also known as being 'semi-decidable'. Propositional logic (assuming that we have a finite number of formulae) is of course decidable: it is always possible to get an answer in a finite amount of time on the question of whether or not a knowledge base is satisfiable. The appendix to the lecture notes includes a summary of logic in AI you need to be familiar with. Consult Appendix A before reading on.

1.3.1 Horn-clause logic

A *Horn clause* or *rule* is a logical implication of the following form

$$\forall x_1 \cdots \forall x_m ((A_1 \wedge \cdots \wedge A_n) \rightarrow B) \tag{1.1}$$

where A_i, B are positive literals, also called atoms, of the form $P(t_1, \ldots, t_q)$, i.e. without a negation sign, representing a relationship P between terms t_k, which may involve one or more universally quantified variables x_j, constants and terms involving function symbols. As all variables in rules are assumed to universally quantified, the universal quantifiers are often omitted if this does not give rise to confusion. If $n = 0$, then the clause consists only of a conclusion, which may be taken as a *fact*. If, on the other hand, the conclusion B is empty, indicated by $\bot$, the rule is also called a *query*. If the conditions of a query are satisfied, this will give rise to a contradiction or inconsistency, denoted by $\Box$ or $\bot$, as the conclusion is empty. So, an empty clause actually means inconsistency.

A popular method to reason with clauses, and Horn clauses in particular, is *resolution*. Let KB be a set of rules not containing queries, and let $Q \equiv (A_1 \wedge \cdots \wedge A_n) \rightarrow \bot$ be a query, then

$$\text{KB} \cup \{Q\} \vdash \bot$$

where $\vdash$ means the application of resolution, implies that the conditions

$$\forall x_1 \cdots \forall x_m (A_1 \wedge \cdots \wedge A_n)$$

are satisfied for some values of $x_1, \ldots, x_m$. Since resolution is a sound inference rule, meaning that it respects the logical meaning of clauses, it also holds that $\text{KB} \cup \{Q\} \vDash \bot$, or equivalently

$$\text{KB} \vDash \exists x_1 \cdots \exists x_m (A_1 \wedge \cdots \wedge A_n)$$

if KB only consists of Horn clauses. This last interpretation explains why deriving inconsistency is normally not really the goal of using resolution; rather, the purpose is to derive certain facts. Since resolution is only complete for deriving inconsistency, called *refutation completeness*, it is only safe to 'derive' knowledge in this indirect manner, i.e., by deriving inconsistency. There exist other reasoning methods which do not have this limitation. However, resolution is a simple method that is understood in considerable depth. As a consequence, state-of-the-art resolution-based reasoners are very efficient.

Resolution can also be used with clauses in general, which are logical expressions of the form

$$(A_1 \wedge \cdots \wedge A_n) \rightarrow (B_1 \vee \cdots \vee B_m)$$

usually represented as:

$$\neg A_1 \vee \cdots \vee \neg A_n \vee B_1 \vee \cdots \vee B_m$$

Rules of the form (1.1) are particularly popular as the reasoning with propositional Horn clauses is known to be possible in polynomial, even linear time, i.e. efficiently, whereas reasoning with propositions or clauses in general (where the right-hand side consists of disjunctions of literals) is known to be NP complete, i.e., may require time exponential in the size of the clauses. The explanation for this is that for Horn clauses there is always at most *one* conclusion B and not, as in non-Horn clauses, a disjunction $(B_1 \vee \cdots \vee B_m)$ that must be checked. In the latter case, there are 2^m possible ways to assign a truth value to the disjunction. Note that allowing negative literals at the left-hand site of a rule is equivalent to having disjunctions at the right-hand side. Using a logical language that is more expressive than Horn-clause logic is sometimes unavoidable, and special techniques have been introduced to deal with their additional power.

Let KB be a knowledge base consisting of a set (conjunction) of rules, and let F be a *set of facts* observed for a particular problem $\mathcal{P}$, then there are generally three ways in which a problem can be solved, yielding different types of solutions. Let $\mathcal{P}$ be a problem, then there are different classes of solutions to this problem:

- **Deductive solution:** S is a *deductive solution* of a problem $\mathcal{P}$ with associated set of observed findings F iff

  $$\mathrm{KB} \cup F \vDash S \tag{1.2}$$

  and $\mathrm{KB} \cup F \nvDash \bot$, where S is a set of solution formulae.

- **Abductive/inductive solution:** S is an *abductive solution* of a problem $\mathcal{P}$ with associated set of observed findings F iff the following *covering condition*

  $$\mathrm{KB} \cup S \cup K \vDash F \tag{1.3}$$

  is satisfied, where K stands for *contextual knowledge*. In addition, it must hold that $\mathrm{KB} \cup S \cup C \nvDash \bot$ (consistency condition), where C is a set of logical constraints on solutions. For the abductive case, it is assumed that the knowledge base KB contains a logical representation of *causal knowledge* and S consists of facts; for the inductive case, KB consists of background facts and S, called an *inductive solution*, consists of rules.

- **Consistency-based solution**: S is a *consistency-based solution* of a problem $\mathcal{P}$ with associated set of observed findings F iff

$$\mathrm{KB} \cup S \cup F \nvDash \bot \tag{1.4}$$

Note that a deductive solution is a consistent conclusion that follows from a knowledge base KB and a set of facts, whereas an abductive solution acts as a hypothesis that *explains* observed facts in terms of causal knowledge, i.e. cause-effect relationships. An inductive solution also explains observed facts, but in terms of any other type of knowledge. A consistency-based solution is the weakest kind of solution, as it is neither required to be concluded nor is it required to explain observed findings. You will encounter examples of these different ways of solving problems in the next chapters, in particular in Chapter 4 on model-based reasoning.

1.3.2 Objects, attributes and values

Even though facts or observed findings can be represented in many different ways, in many knowledge systems facts are represented in an object-oriented fashion. This means that facts are described as properties, or *attributes*, of objects in the real world. Attributes of objects can be either multivalued, meaning that an object may have more than one of those properties at the same time, or singlevalued, meaning that values of attributes are mutually exclusive.

In logic, multivalued attributes are represented by predicate symbols, e.g.:

$$\mathrm{Parent}(\mathrm{John}, \mathrm{Ann}) \wedge \mathrm{Parent}(\mathrm{John}, \mathrm{Derek})$$

indicates that the 'object' John, represented as a constant, has two parents (the attribute 'Parent'): Ann and Derek, both represented by constants. Furthermore, singlevalued attributes are represented as function symbols, e.g.

$$gender(\mathrm{John}) = male$$

Here, '*gender*' is taken as a singlevalued attribute, 'John' is again a constant object, and '*male*' is the value, also represented as a constant.

It is, of course, also possible to state general properties of objects. For example, the following bi-implication:

$$\forall x \forall y \forall z ((\mathrm{Parent}(x, y) \wedge \mathrm{Parent}(y, z)) \leftrightarrow \mathrm{Grandparent}(x, z))$$

defines the attribute 'Grandparent' in terms of the 'Parent' attribute.

Another typical example of reasoning about properties of objects is *inheritance*. Here one wishes to associate properties of objects with the classes the objects belong to, mainly because this yields a compact representation offering in addition insight into the general structure of a problem domain. Consider, for example, the following knowledge base KB:

$$\forall x(\mathrm{Vehicle}(x) \to \mathrm{HasWheels}(x))$$
$$\forall x(\mathrm{Car}(x) \to \mathrm{Vehicle}(x)$$
$$\forall x(\mathrm{Car}(x) \to \textit{number-of-wheels}(x) = 4)$$

Clearly, it holds that

$$\mathrm{KB} \cup \{\mathrm{Car}(\mathrm{Bugatti})\} \vDash \textit{number-of-wheels}(\mathrm{Bugatti}) = 4$$

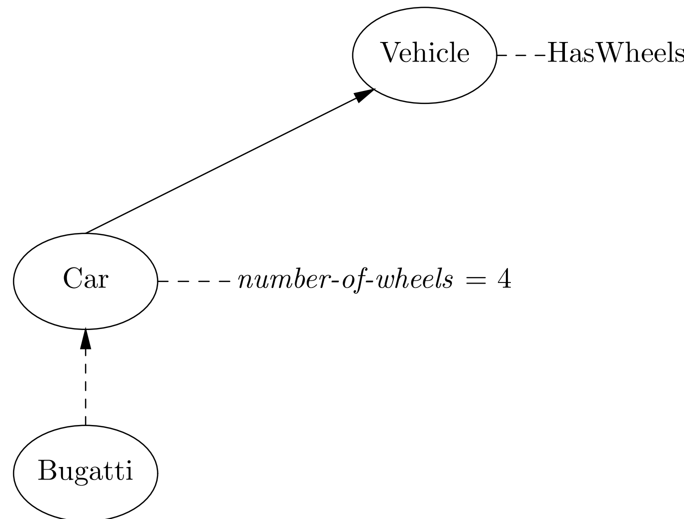

Figure 1.2: An object taxonomy.

as the third rule expresses that as a typical property of cars. However, the knowledge base also incorporates more general properties of cars, such as:

$$\text{KB} \cup \{\text{Car(Bugatti)}\} \vDash \text{Vehicle(Bugatti)}$$

Now, given the fact that a car is a vehicle, we can now also conclude

$$\text{KB} \cup \{\text{Car(Bugatti)}\} \vDash \text{HasWheels(Bugatti)}$$

The example knowledge base discussed above can also be represented as a graph, called an object *taxonomy*, and is shown in Figure 1.2. Here ellipses indicate either classes of objects (Car and Vehicle) or specific objects (Bugatti). Solid arcs in the graph indicate that a class of objects is a subclass of another class of objects; a dashed arc indicates that the parent object is an element – often the term 'instance' is used instead – of the associated class of objects. The term 'inheritance' that is associated with this type of logical reasoning derives from the fact that the reasoning goes from the children to the parents in order to derive properties. More about inheritance will be said in Chapter 3.

Describing the objects in a domain, usually but not always in a way resembling a taxonomy, usually with the intention to obtain a formal description of the terminology in a domain, is known as an *ontology*.

In conclusion, finding the right *trade-off* between expressive power and computational complexity is one of the most important issues for (practical) knowledge representation systems in AI [1].

1.4 Reasoning with uncertainty

In the early days of knowledge systems, *rule-based systems*, which can be looked at as system that represent knowledge as rules of the form $A \rightarrow B$, were popular. Early reasoning with uncertainty was, therefore, studied in the context of such rule-based systems with rules of the form $A \rightarrow B_x$, with x some measure of uncertainty. The meaning of this rule is that when A

is absolutely true then B is also true, but with certainty x. Of course, the actual meaning of x depends on the theory being used.

At the end of the 1980s, rule-based systems were slowly replaced by the then new Bayesian networks. Bayesian networks are structured joint (multivariate) probability distributions. They allow for computing of any probability of interest. For example, if one has the joint distribution $P(X,Y)$ one can compute $P(X)$ simply by marginalising out Y:

$$P(X) = \sum_{Y} P(X,Y)$$

In addition, it is straightforward to determine the effect of observations X (facts known with certainty) on the uncertainty of any set of random variables Y by computing the *conditional probability distribution* $P(Y \mid X)$.

It appears to be fruitful to use probability theory as a basic framework to understand both early, rule-based approached to uncertainty reasoning (e.g. the certainty-factor calculus), more recent work on probabilistic graphical models (e.g. Bayesian networks), and the latest work on probabilistic logics (e.g. Markov logic).

1.5 Conclusions

The field of knowledge representation and reasoning has seen a development from practical systems that solved actual problems at the end of the 1970s and the beginning of the 1980s, such as the MYCIN system that was developed to assist clinicians in the diagnosis and treatment of infectious diseases, to the development of the underlying theory in the 1980s and 1990s. In particular many different ways to use logic as a language for the development of AI systems has attracted much attention in the 1980s and 1990s, and we will look at examples of such research during the course. Finally the development of reasoning of uncertainty from rule-based reasoning to network models and, recently, to probabilistic logic is an interesting example of progress in AI research. This last development, which explains the subtitle of the course, will be considered as well. The book by Van Harmelen et al. [2] offers a detailed account of all of these developments.

Chapter 2

Lecture 2-4 – Logic Programming

Logic programming and the associated programming language Prolog are convenient vehicles for this course on knowledge representation and reasoning, because of the dominant role played by logic and logical reasoning in the area.

2.1 Introduction

Prolog is a simple, yet powerful programming language, based on the principles of first-order predicate logic. The name of the language is an acronym for the French 'PROgrammation en LOGique'. About 1970, Prolog was designed by A. Colmerauer and P. Roussel at the University of Marseille, influenced by the ideas of R.A. Kowalski concerning programming in the Horn clause subset of first-order predicate logic. The name of Prolog has since then been connected with a new programming style, known as *logic programming*.

Until the end of the seventies, the use of Prolog was limited to the academic world. Only after the development of an efficient Prolog interpreter and compiler by D.H.D. Warren and F.C.N. Pereira at the University of Edinburgh, the language entered the world outside the research institutes. The interest in the language has increased steadily. However, Prolog is still mainly used by researchers, even though it allows for the development of serious and extensive programs in a fraction of the time needed to develop a C or Java program with similar functionality. The only explanation is that people like waisting their precious time. Nevertheless, there are a large number of fields in which Prolog has been applied successfully. The main applications of the language can be found in the area of Artificial Intelligence; but Prolog is being used in other areas in which symbol manipulation is of prime importance as well. Some application areas are:

- Natural-language processing;
- Compiler construction;
- The development of knowledge systems;
- Work in the area of computer algebra;
- The development of (parallel) computer architectures;
- Database systems.

Prolog is particularly strong in solving problems characterised by requiring complex symbolic computations. As conventional imperative programs for solving this type of problems tend to be large and impenetrable, equivalent Prolog programs are often much shorter and easier to grasp. The language in principle enables a programmer to give a formal specification of a program; the result is then almost directly suitable for execution on the computer. Moreover, Prolog supports stepwise refinement in developing programs because of its modular nature. These characteristics render Prolog a suitable language for the development of prototype systems.

There are several dialects of Prolog in use, such as for example, C-Prolog, SWI-Prolog, Sicstus-Prolog, LPA-Prolog. C-Prolog, also called Edinburgh Prolog, was taken as a basis for the ISO standard. C-Prolog itself is now no longer in use.

The language definition of C-Prolog is derived from an interpreter developed by D.H.D. Warren, D.L. Bowen, L. Byrd, F.C.N. Pereira, and L.M. Pereira, written in the C programming language for the UNIX operating system. Most dialects only have minor syntactical and semantical differences with the standard language. However, there are a small number of dialects which change the character of the language in a significant way, for example by the necessity of adding data-type information to a program. A typical example is offered by the version of the Prolog language supported by Visual Prolog. In recent versions of Prolog, several features have been added to the ISO standard. Modern Prolog versions provide a module concept and extensive interfaces to the operating system, as well as tools for the development of graphical user interfaces. As these have not been standardised, we will not pay attention to them here.

2.2 Logic programming

In more conventional, imperative languages such as C++ and Java a program is a specification of a sequence of instructions to be executed one after the other by a target machine, to solve the problem concerned. The description of the problem is incorporated implicitly in this specification, and usually it is not possible to clearly distinguish between the description of the problem, and the method used for its solution. In logic programming, the description of the problem and the method for solving it are explicitly separated from each other. This separation has been expressed by R.A. Kowalski in the following equation:

$$\boxed{\text{algorithm} = \text{logic} + \text{control}}$$

The term 'logic' in this equation indicates the descriptive component of the algorithm, that is, the description of the problem; the term 'control' indicates the component that tries to find a solution, taking the description of the problem as a point of departure. So, the logic component defines *what* the algorithm is supposed to do; the control component indicates *how* it should be done.

A specific problem is described in terms of relevant objects and relations between objects, which are then represented in the clausal form of logic, a restricted form of first-order predicate logic. The logic component for a specific problem is generally called a *logic program*. The control component employs logical deduction or reasoning for deriving new facts from the logic program, thus solving the given problem; one speaks of the *deduction* or *reasoning method*. The deduction method is assumed to be quite general, in the sense that it is capable of dealing with any logic program respecting the clausal form syntax.

The splitting of an algorithm into a logic component and a control component has a number of advantages:

- The two components may be developed separately from each other. For example, when describing the problem we do not have to be familiar with how the control component operates on the resulting description; knowledge of the declarative reading of the problem specification suffices.
- A logic component may be developed using a method of stepwise refinement; we have only to watch over the correctness of the specification.
- Changes to the control component affect (under certain conditions) only the efficiency of the algorithm; they do not influence the solutions produced.

An environment for logic programming offers the programmer a reasoning method, so that only the logic program has to be developed for the problem at hand. The reasoning method of logic programming is known as SLD resolution.

2.3 SLD resolution: a special form of resolution

For details about predicate logic, clausal form, resolution and unification, you are referred to Appendix A. You are expected to understand resolution before continuing reading.

SLD resolution is the reasoning method used to reason with logic programs. It operates on *Horn clauses*, which are logical implications taking the following form:

$$B \leftarrow A_1, \ldots, A_n$$

where B, $A_1, \ldots, A_n$, $n \geq 0$, are atomic formulas. The commas ',' here have the meaning of a conjunction $\wedge$, whereas $\leftarrow$ is the reverse logical implication symbol. Thus, outside the area of logical programming, a Horn clause is denoted as follows:

$$\forall x_1 \cdots \forall x_m ((A_1 \wedge \cdots \wedge A_n) \rightarrow B)$$

SLD resolution is a form of *linear resolution*: in every resolution step the last generated resolvent is taken as a one of the two parent clauses. The other parent clause is either a clause from the original set of clauses or a resolvent that has been generated before. With SLD resolution, each resolution step, with the exception of the first one, is carried out on the last generated resolvent and a clause from the original set of clauses. The former clauses are called *goal clauses*; the latter clauses are called *input clauses* or *program clauses*. SLD resolution also includes a *selection rule* which determines at every step which literal from the goal clause is selected for resolution.

An SLD derivation is defined as follows:

Definition 2.1 *Let $\{C_i\}$ be a set of Horn clauses with*

$$C_i = B \leftarrow B_1, \ldots, B_p$$

where $p \geq 0$, and let G_0 be a goal clause of the form

$$G_0 = \; \leftarrow A_1, \ldots, A_q$$

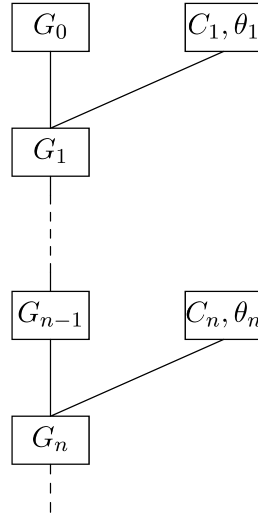

Figure 2.1: Derivation tree of SLD resolution.

where $q \geq 0$. *An* SLD derivation *is a finite or infinite sequence* $G_0, G_1, \ldots$ *of goal clauses, a sequence* $C_1, C_2, \ldots$ *of variants of input clauses and a sequence* $\theta_1, \theta_2, \ldots$ *of most general unifiers, such that each* G_{i+1} *is derived from* $G_i = \leftarrow A_1, \ldots, A_k$ *and* C_{i+1} *using* θ_{i+1} *if the following conditions hold:*

(1) A_j *is the atom in the goal clause* G_i *chosen by the selection rule to be resolved upon, and*

(2) C_{i+1} *is an input clause of the form*

$$C_{i+1} = B \leftarrow B_1, \ldots, B_p$$

(in which variables have been renamed, if necessary), such that $A_j\theta_{i+1} = B\theta_{i+1}$, *where* θ_{i+1} *is a most general unifier of* A_j *and* B.

(3) $G_i + 1$ *is the clause*

$$G_{i+1} = \leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_p, A_{j+1}, \ldots, A_k)\theta_{i+1}$$

If for some $n \geq 0$, $G_n = \square$, *then the derivation is called an* SLD refutation *and the number* n *is called the* length of the refutation.

Note that a new goal clause G_{i+1} is the resolvent of the last computed resolvent G_i and (a variant of) an input clause C_{i+1}. Figure 2.1 shows the general form of a derivation tree by SLD resolution. In this figure the sequence of successive goal clauses (resolvents) $G_0, G_1, \ldots$ has been indicated.

EXAMPLE 2.1

Consider the following set of Horn clauses:

$$\{R(g(x)) \leftarrow T(x, y, f(x)), T(a, b, f(a)), P(v, w) \leftarrow R(v)\}$$

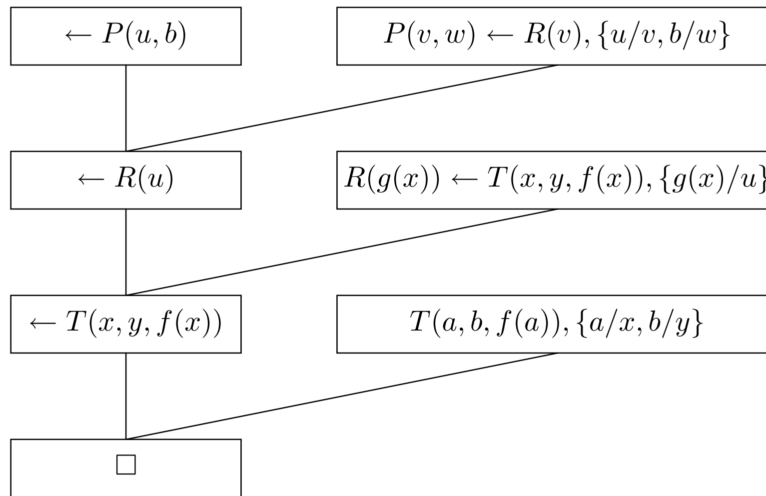

Figure 2.2: An SLD refutation.

Furthermore, let the following goal clause be given:

$$\leftarrow P(u,b)$$

The clause set obtained by adding the goal clause to the original set of clauses is unsatisfiable. This can be proven using SLD resolution. Figure 2.2 depicts this proof by SLD refutation as a derivation tree.

SLD resolution is both sound and complete for Horn clauses. It furthermore is similar to the set-of-support strategy in the sense that it is also a resolution strategy controlled by a set of goals. So, SLD resolution is a form of top-down inference as well. In general it is advantageous to restrict applying the resolution principle to clauses satisfying the Horn clause format: various resolution algorithms for propositional Horn clause logic are known to have a worst-case time complexity almost linear in the number of literals. When applying some resolution strategy suitable for the clausal form of logic in general, we always have to face the danger of a combinatorial explosion. Moreover, for systems based on SLD resolution many efficient implementation techniques have been developed by now, one of which will be discussed in the next section. But there definitely are problems for which a resolution strategy applying some form of bottom-up inference turns out to be more efficient than SLD resolution.

Before introducing the notion of a search space for SLD resolution, we give another example.

EXAMPLE 2.2

Consider the following set of Horn clauses:

$$\begin{aligned} C_1 &= P(x) \leftarrow P(f(x)) \\ C_2 &= P(f(f(a))) \leftarrow \end{aligned}$$

If these clauses are 'tried' in the order in which they are specified, then for the goal clause $\leftarrow P(a)$ no refutation is found in a finite number of steps, although the resulting

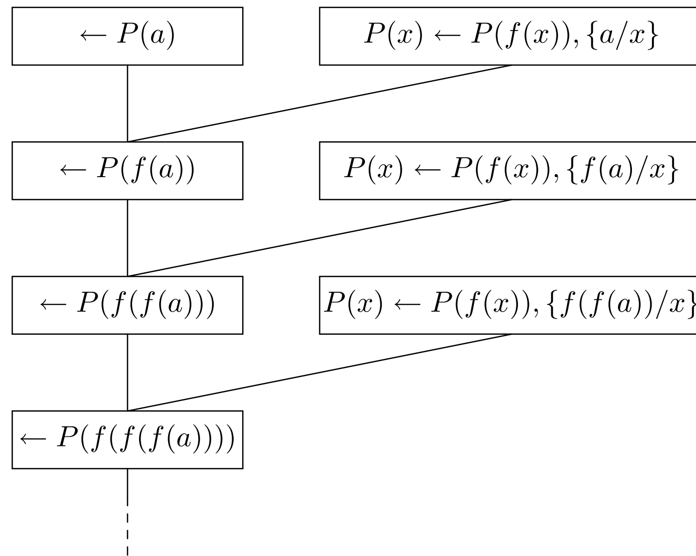

Figure 2.3: Infinite derivation tree by SLD resolution.

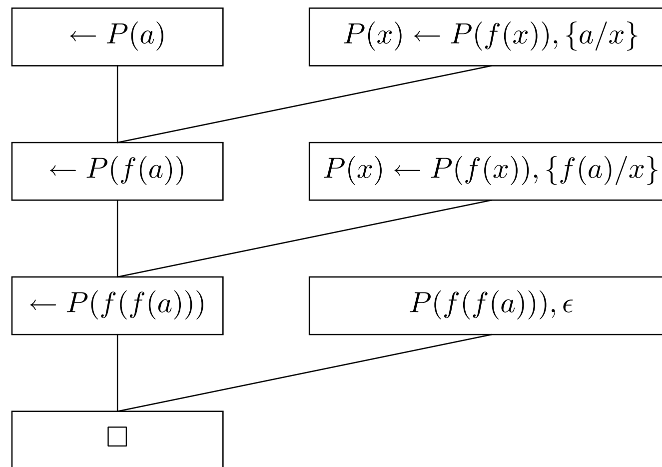

Figure 2.4: Refutation by SLD resolution.

set of clauses obviously is unsatisfiable. The corresponding derivation tree is shown in Figure 2.3. However, if the clauses C_1 and C_2 are processed in the reverse order C_2, C_1, then a refutation will be found in finite time: the resulting refutation tree is shown in Figure 2.4.

Now let the search space for SLD resolution for a given goal on a set of clauses be a graph in which every possible SLD derivation is shown. Such a search space is often called an *SLD tree*. The branches of the tree terminating in the empty clause $\square$ are called *success branches*. Branches corresponding to infinite derivations are called *infinite branches*, and the branches representing derivations which have not been successful and cannot be pursued any further are called *failure branches*. The *level* of a vertex in an SLD tree is obtained by assigning the number 0 to the root of the tree; the level of each other vertex of the tree is obtained by incrementing the level of its parent vertex by 1.

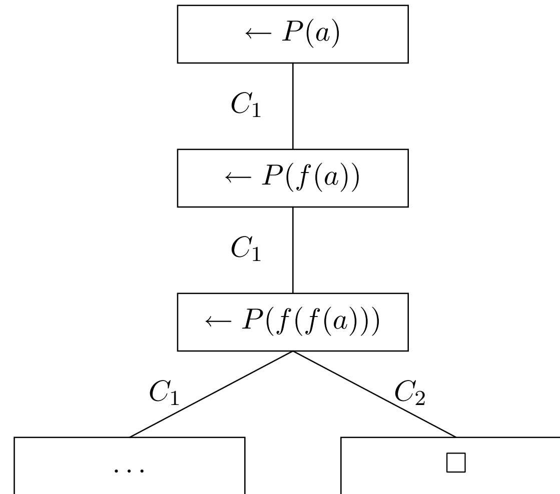

Figure 2.5: An SLD tree.

EXAMPLE 2.3

Figure 2.5 shows the SLD tree corresponding to SLD resolution on the set of clauses from the previous example. The right branch of the tree is a success branch and corresponds to the refutation depicted in Figure 2.4; the left branch is an example of a failure branch.

It can easily be seen that a specific, fixed order in choosing parent clauses for resolution such as in the previous example, corresponds to a depth-first search in the search space. Note that such a depth-first search defines an incomplete resolution procedure, whereas a breadth-first search strategy defines a complete one. Although SLD resolution is both sound and complete for Horn clauses, in practical realisations for reasons of efficiency, variants of the algorithm are used that are neither sound nor complete. First of all, in many implementations the 'expensive' occur check has been left out from the unification algorithm, thus destroying the soundness; the lack of the occur check might lead to circular variable bindings and yield 'resolvents' that are no logical consequences of the set of clauses. Furthermore, often the original clauses are 'tried' in some specific order, such as for example the order in which the clauses have been specified; the next input clause is only examined after the previous one has been fully explored. As a consequence, the algorithm might not be able to find a proof of a given theorem: due to an inappropriate choice of the order in which the clauses are processed, an infinite derivation tree can be created. This way, completeness of SLD resolution will be lost.

We have mentioned before that SLD resolution is of major interest because of its relation with the programming language Prolog. In Prolog, the control strategy employed is roughly an implementation of SLD resolution; the variant used however, is neither sound nor complete. In most (standard) Prolog systems, the selection rule picks the leftmost atom from a goal for resolution. A depth-first strategy for searching the SLD tree is used: most Prolog systems 'try' the clauses in the order in which they have been specified. Furthermore, in many Prolog systems, for efficiency reasons, the occur check has been left out from the implementation.

The Horn clause subset of logic is not as expressive as the full clausal form of logic is. As is shown in the following example, this might lead to problems when translating the logical

formulas into the Horn clause subset.

EXAMPLE 2.4

In Section A.2 we defined the following predicates with their associated intended meaning:

Car	=	'is a car'
Fast	=	'is a fast car'
Vehicle	=	'is a vehicle'
FourWheels	=	'has four wheels'
Exception	=	'is an exception'

The formula $\forall x(Car(x) \rightarrow Vehicle(x))$ represents the knowledge that every car is a vehicle. This formula is logically equivalent to $\forall x(\neg Car(x) \vee Vehicle(x))$ and results in the following Horn clause:

$$Vehicle(x) \leftarrow Car(x)$$

The knowledge that a Bugatti is a fast car, is represented as a Horn clause representing a single fact:

$$Fast(bugatti) \leftarrow$$

The implication

$$\forall x((Car(x) \wedge \neg Exception(x)) \rightarrow FourWheels(x))$$

stating that almost every car, except for instance a Bugatti, has four wheels. This formula is equivalent to

$$\forall x(\neg(Car(x) \wedge \neg Exception(x)) \vee FourWheels(x))$$

and to the formula

$$\forall x(\neg Car(x) \vee Exception(x) \vee FourWheels(x))$$

in disjunctive normal form. Unfortunately, it is not possible to translate this formula directly into logic programming representation, since the clause contains two positive literals instead of at most one, and, thus, is not a Horn clause. However, it is possible to represent the knowledge expressed by the clause in logic programming, by means of the rather special programming trick offered by the meaning of the standard predicate '*not*', which will be discussed below. The logic programming clause we arrive at is the following:

$$Fourwheels(x) \leftarrow Car(x), not(Exception(x))$$

This this is essentially a Horn clause with an extra predicate: '*not*'. Note that in the analogous example in Section A.2 it was necessary to specify that an alfa-romeo is not an exception to the general rule that cars have four wheels. In fact, for a correct behaviour

of a proof procedure it was necessary to specify for each artery explicitly whether or not it is an exception to the rule. In most applications however, it is unreasonable to expect users to explicitly express all negative information relevant to the employed proof procedure. This problem can be handled by considering a ground literal $not(P)$ proven if an attempt to prove P using SLD resolution has not succeeded. So, in the particular case of the example, it is assumed that the goal clause

$$\leftarrow not(Exception(alfa\text{-}romeo))$$

is proved.

The inference rule that a negative literal is assumed proven when the attempt to prove the complementary literal has failed is called *negation as failure*. Negation as failure is similar to the so-called *closed-world assumption* which is quite common in database applications. In Prolog, an even stronger assumption, known as negation as *finite* failure, is made by taking $not(P)$ proven, or satisfied, only if proving P using SLD resolution has failed in a finite number of steps. Also Prolog includes this predicate `not` is the implementation of this negation as finite failure and therefore should not be taken as the ordinary negation: it is an extra-logical feature of Prolog.

2.4 Programming in Prolog

The programming language Prolog can be considered to be a first step towards the practical realisation of logic programming; as we will see in below, however, the separation between logic and control has not been completely realised in this language. Figure 2.6 shows the relation between Prolog and the idea of logic programming discussed above. In addition, predicates in Prolog always start with a lower-case letter, e.g., 'car(bugatti)' rather than 'Car(bugatti)'. Variables, which as in logical programming are implicitly universally quantified, always start with an upper-case letter or underscore, e.g., 'X' rather than 'x'. Combined with a predicate we get 'car(X)' instead of the '$Car(x)$' we used earlier. A Prolog system consists of two components: a *Prolog database* and a *Prolog interpreter*.

A Prolog program, essentially a logic program consisting of *Horn clauses* (which however may contain some directives for controlling the inference method), is entered into the Prolog database by the programmer. As mentioned above, the Prolog interpreter offers a reasoning method based on SLD resolution.

Solving a problem in Prolog starts with discerning the objects that are relevant to the particular problem, and the relationships that exist between them.

EXAMPLE 2.5

In a problem concerning sets, we for instance take constants as separate objects and the set as a whole as another object; a relevant relation between constants and sets is the membership relation.

When we have identified all relevant objects and relations, it must be specified which *facts* and *rules* hold for the objects and their interrelationships.

EXAMPLE 2.6

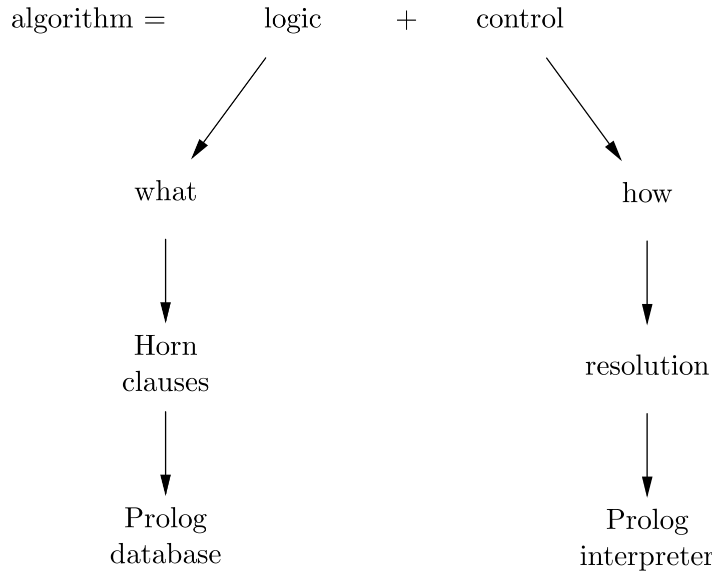

Figure 2.6: The relationship between Prolog and logic programming.

Suppose that we are given a problem concerning sets. We may for example have the fact that a certain constant a is a member of a specific set S. The statement 'the set X is a subset of the set Y, if each member of X is a member of Y' is a rule that generally holds in set theory.

When all facts and rules have been identified, then a specific problem may be looked upon as a query concerning the objects and their interrelationships. To summarise, specifying a logic program amounts to:

- Specifying the *facts* concerning the objects and relations between objects relevant to the problem at hand;
- Specifying the *rules* concerning the objects and their interrelationships;
- Posing *queries* concerning the objects and relations.

2.4.1 The declarative semantics

As mentioned above, knowledge (facts, rules, and queries) is represented in Prolog using the formalism of *Horn clause logic*. A *Horn clause* takes the following form:

$$B \leftarrow A_1, \ldots, A_n$$

where B, $A_1, \ldots, A_n$, $n \geq 0$, are atomic formulas. Instead of the (reverse) implication symbol, in Prolog usually the symbol `:-` is used, and clauses are terminated by a dot. An *atomic formula* is an expression of the following form:

$$P(t_1, \ldots, t_m)$$

Formal	Name	In Prolog	Name
$A \leftarrow$	unit clause	A.	fact
$\leftarrow B_1, \ldots, B_n$	goal clause	?- $B_1, \ldots, B_n$.	query
$A \leftarrow B_1, \ldots, B_n$	clause	A:-$B_1, \ldots, B_n$.	rule

Table 2.1: Horn clauses and Prolog.

where P is a *predicate* having m arguments, $m \geq 0$, and $t_1, \ldots, t_m$ are terms. A *term* is either a constant, a variable, or a function of terms. In Prolog two types of constants are distinguished: numeric constants, called *numbers*, and symbolic constants, called *atoms*. (Note that the word atom is used here in a meaning differing from that of atomic formula, thus deviating from the standard terminology of predicate logic.) Because of the syntactic similarity of predicates and functions, both are called *functors* in Prolog. The terms of a functor are called its *arguments*. The arguments of a functor are enclosed in parentheses, and separated by commas.

Seen in the light of the discussion from the previous section, the predicate P in the atomic formula $P(t_1, \ldots, t_m)$ is interpreted as the name of the relationship that holds between the objects $t_1, \ldots, t_m$ which occur as the arguments of P. So, in a Horn clause B `:-` $A_1, \ldots, A_n$, the atomic formulas B, $A_1, \ldots, A_n$, denote relations between objects. A Horn clause now is interpreted as stating:

'B (is true) if A_1 and A_2 and ... and A_n (are true)'

$A_1, \ldots, A_n$ are called the *conditions* of the clause, and B its *conclusion*. The commas between the conditions are interpreted as the logical $\wedge$, and the `:-` symbol as the (reverse) logical implication $\leftarrow$.

If $n = 0$, that is, if conditions A_i are lacking in the clause, then there are no conditions for the conclusion to be satisfied, and the clause is said to be a *fact*. In case the clause is a fact, the `:-` sign is replaced by a dot.

Both terminology and notation in Prolog differ slightly from those employed in logic programming. Table 2.1 summarises the differences and similarities. The use of the various syntactic forms of Horn clauses in Prolog will now be introduced by means of examples.

EXAMPLE 2.7

The Prolog clause

```
/*1*/     member(X,[X|_]).
```

is an example of a fact concerning the relation with the name `member`. This relation concerns the objects `X` and `[X|_]` (their meaning will be discussed shortly). The clause is preceded by a comment; in Prolog, comments have to be specified between the delimiters `/*` and `*/`.

If a clause contains one or more conditions as well as a conclusion, it is called a *rule*.

EXAMPLE 2.8

Consider the Prolog clause

```
/*2*/     member(X,[_|Y]) :- member(X,Y).
```

which is a rule concerning the relation with the name `member`. The conclusion `member(X,[_|Y])` is only subjected to one condition: `member(X,Y)`.

If the conclusion is missing from a clause, then the clause is considered to be a query to the logic program. In case a clause is a query, the sign `:-` is usually replaced by the sign `?-`.

EXAMPLE 2.9

The Prolog clause

```
/*3*/     ?- member(a,[a,b,c]).
```

is a typical example of a query.

A symbolic constant is denoted in Prolog by a name starting with a lower-case letter. Names starting with an upper-case letter, or an underscore sign, `_`, indicate *variables* in Prolog. A relation between objects is denoted by means of a functor having a name starting with a lower-case letter (or a special character, such as &, not having a predefined meaning in Prolog), followed by a number of arguments, that is the objects between which the relation holds. Recall that arguments are terms, that is, they may be either constants, variables, or functions of terms.

EXAMPLE 2.10

Consider the three clauses from the preceding examples once more. `member` is a functor having two arguments. The names `a`, `b`, and `c` in clause 3 denote symbolic constants; `X` and `Y` are variables.

In Prolog, a collection of elements enclosed in square brackets denotes a *list*. It is possible to explicitly decompose a list into its first element, the *head* of the list, and the remaining elements, the *tail* of the list. In the notation `[X|Y]`, the part in front of the bar is the head of the list; `X` is a single element. The part following the bar denotes its tail; `Y` itself is a list.

EXAMPLE 2.11

Consider the list `[a,b,c]`. Now, `[a|[b,c]]` is another notation for the same list; in this notation, the head and the tail of the list are distinguished explicitly. Note that the tail again is a list.

Each clause represents a separate piece of knowledge. So, in theory, the meaning of a set of clauses can be specified in terms of the meanings of each of the separate clauses. The meaning of a clause is called the *declarative semantics* of the clause. Knowledge of the declarative semantics of first-order predicate logic helps in understanding Prolog. Broadly speaking, Prolog adheres to the semantics of first-order logic. However, there are some differences, such as the use of negation as finite failure which will be discussed below.

EXAMPLE 2.12

Consider the clauses 1, 2 and 3 from the preceding examples once more. Clause 1 expresses that the relation with the name `member` holds between a term and a list of terms, if the head of the list equals the given term. Clause 1 is not a statement concerning specific terms, but it is a general statement; this can be seen from the use of the variable `X` which may be substituted with any term. Clause 2 represents the other possibility that the constant occurs in the tail of the list. The last clause specifies the query whether or not the constant `a` belongs to the list of constants `a`, `b`, and `c`.

2.4.2 The procedural semantics and the interpreter

In the preceding section we have viewed the formalism of Horn clause logic merely as a formal language for representing knowledge. However, the Horn clause formalism can also be looked upon as a programming language. This view of Horn clause logic is called its *procedural semantics*. Essentially, the procedural interpretation is obtained through SLD resolution with the addition of some special programming-language like terminology and some restrictions due to the difficulty of providing a full implementation of SLD resolution.

In the procedural semantics, a set of clauses is viewed as a program. Each clause in the program is seen as a *procedure (entry)*. In the clause

$$B\texttt{:-}A_1, \ldots, A_n.$$

we look upon the conclusion B as the *procedure heading*, composed of a procedure name, and a number of formal parameters; $A_1, \ldots, A_n$ is then taken as the *body* of the procedure, consisting of a sequence of *procedure calls*. In a program all clauses having the same predicate in their conclusion, are viewed as various entries to the same procedure. A clause without any conclusion, that is, a query, acts as the *main program*. Here no strict distinction is made between both types of semantics; it will depend on the subject dealt with, whether the terminology of the declarative semantics is used, or the terminology of procedural semantics is preferred. In the remainder of this section we shall discuss the Prolog interpreter.

When a Prolog program has been entered into the Prolog database, the main program is executed by the Prolog interpreter. The way the given Prolog clauses are manipulated, will be demonstrated by means of some examples.

EXAMPLE 2.13

The three clauses introduced in Section 2.4.1 together constitute a complete Prolog program:

```
/* 1*/     member(X,[X|_]).
/* 2*/     member(X,[_|Y]) :-
              member(X,Y).

/* 3*/     ?- member(a,[a,b,c]).
```

Clauses 1 and 2 are entries to the same `member` procedure. The body of clause 2 consists of just one procedure call. Clause 3 fulfils the role of the main program.

Let us suppose that the Prolog database initially contains the first two clauses, and that clause 3 is entered by the user as a query to the Prolog system. The Prolog interpreter tries to derive an answer to the query using the information stored in the database. To this end, the interpreter employs two fundamental techniques: matching and backtracking.

Matching of clauses

To answer a query, the Prolog interpreter starts with the first condition in the query clause, taking it as a procedure call. The Prolog database is subsequently searched for a suitable entry to the called procedure; the search starts with the first clause in the database, and continues until a clause has been found which has a conclusion that can be matched with the procedure call. A *match* between a conclusion and a procedure call is obtained, if there exists a substitution for the variables occurring both in the conclusion and in the procedure call, such that the two become (syntactically) equal after the substitution has been applied to them. Such a match exists

- If the conclusion and the procedure call contain the same predicate, and
- If the terms in corresponding argument positions after substitution of the variables are equal; one then also speaks of a match for argument positions.

Applying a substitution to a variable is called *instantiating* the variable to a term. The most general substitution making the selected conclusion and the procedure call syntactically equal, is called the *most general unifier* (*mgu*) of the two. The algorithmic and theoretical basis of matching is given by *unification* (See Appendix A and [14] for details).

If we have obtained a match for a procedure call, the conditions of the matching clause will be executed. In case the matching clause has no conditions, the next condition from the calling clause is executed. The process of matching (and instantiation) can be examined by means of the special infix predicate `=`, which tries to match the terms at its left-hand and right-hand side and subsequently investigates whether the terms have become syntactically equal.

EXAMPLE 2.14

Consider the following example of the use of the matching predicate `=`. The first line representing a query has been entered by the user; the next line is the system's output.

```
?- f(X) = f(a).
X = a
```

As can be seen, the variable `X` is instantiated to `a`, which leads to a match of the left-hand and right-hand side of `=`.

On first thoughts, instantiation seems similar to the assignment statement in conventional programming languages. However, these two notions differ considerably. An instantiation is a binding of a variable to a value which cannot be changed, that is, it is not possible to overwrite the value of an instantiated variable by some other value (we will see however, that

under certain conditions it is possible to create a new instantiation). So, it is not possible to express by instantiation a statement like

$X := X + 1$

which is a typical assignment statement in a language like Pascal. In fact, the 'ordinary' assignment which is usually viewed as a change of the state of a variable, cannot be expressed in standard logic.

A variable in Prolog has for its lexical scope the clause in which it occurs. Outside that clause, the variable and the instantiations to the variable have no influence. Prolog does not have global variables. We shall see later that Prolog actually does provide some special predicates which have a global effect on the database; the meanings of such predicates, however, cannot be accounted for in first-order logic. Variables having a name only consisting of a single underscore character, have a special meaning in Prolog. These variables, called *don't-care variables*, match with any possible term. However, such a match does not lead to an instantiation to the variable, that is, past the argument position of the match a don't care variable looses its 'binding'. A don't care variable is usually employed at argument positions which are not referred to later in some other position in the clause.

EXAMPLE 2.15

In our `member` example, the interpreter tries to obtain a match for the following query:

```
/*3*/     ?- member(a,[a,b,c]).
```

The first clause in the database specifying the predicate `member` in its conclusion, is clause 1:

```
/*1*/     member(X,[X|_]).
```

The query contains at its first argument position the constant `a`. In clause 1 the variable `X` occurs at the same argument position. If the constant `a` is substituted for the variable `X`, then we have obtained a match for the first argument positions. So, `X` will be instantiated to the constant `a`. As a consequence, the variable `X` at the second argument position of the conclusion of clause 1 has the value `a` as well, since this `X` is the same variable as at the first argument position of the same clause. We now have to investigate the respective second argument positions, that is, we have to compare the lists `[a,b,c]` and `[a|_]`. Note that the list `[a,b,c]` can be written as `[a|[b,c]]`; it is easily seen that we succeed in finding a match for the second argument positions, since the don't care variable will match with the list `[b,c]`. So, we have obtained a match with respect to the predicate name as well as to all argument positions. Since clause 1 does not contain any conditions, the interpreter answers the original query by printing `yes`:

```
/*3*/     ?- member(a,[a,b,c]).
yes
```

EXAMPLE 2.16

Consider again the clauses 1 and 2 from the preceding example. Suppose that, instead of the previous query, the following query is entered:

```
/*3*/     ?- member(a,[b,a,c]).
```

Then again, the interpreter first tries to find a match with clause 1:

```
/*1*/     member(X,[X|_]).
```

Again we have that the variable `X` will be instantiated to the constant `a`. In the second argument position of clause 1, the variable `X` also has the value `a`. We therefore have to compare the lists `[b,a,c]` and `[a|_]`: this time, we are not able to find a match for the second argument positions. Since the only possible instantiation of `X` is to `a`, we will never find a match for the query with clause 1. The interpreter now turns its attention to the following entry of the `member` procedure, being clause 2:

```
/*2*/     member(X,[_|Y]) :-
             member(X,Y).
```

When comparing the first argument positions of the query and the conclusion of clause 2 respectively, we infer that the variable `X` will again be instantiated to the constant `a`. For the second argument positions we have to compare the lists `[b,a,c]` and `[_|Y]`. We obtain a match for the second argument positions by instantiating the variable `Y` to the list `[a,c]`. We have now obtained a complete match for the query with the conclusion of clause 2. Note that all occurrences of the variables `X` and `Y` within the scope of clause 2 will have been instantiated to `a` and `[a,c]`, respectively. So, after instantiation we have

```
member(a,[_|[a,c]]) :-
   member(a,[a,c]).
```

Since, clause 2 contains a condition, its conclusion may be drawn only if the specified condition is fulfilled. The interpreter treats this condition as a new query:

```
?- member(a,[a,c]).
```

This query matches with clause 1 in the same way as has been described in the previous example; the interpreter returns success. Subsequently, the conclusion of clause 2 is drawn, and the interpreter prints the answer `yes` to the original query.

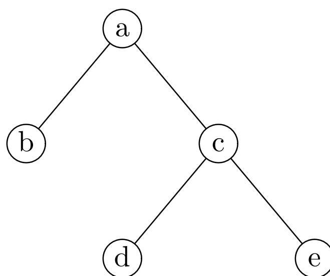

Figure 2.7: A binary tree.

Backtracking

When after the creation of a number of instantiations and matches the system does not succeed in obtaining the next match, it systematically tries alternatives for the instantiations and matches arrived at so far. This process of finding alternatives by undoing previous work, is called *backtracking*. The following example demonstrates the process of backtracking.

EXAMPLE 2.17

Consider the following Prolog program:

```
/*1*/     branch(a,b).
/*2*/     branch(a,c).
/*3*/     branch(c,d).
/*4*/     branch(c,e).
/*5*/     path(X,X).
/*6*/     path(X,Y) :-
             branch(X,Z),
             path(Z,Y).
```

The clauses 1–4 inclusive represent a specific binary tree by means of the predicate `branch`; the tree is depicted in Figure 2.7. The symbolic constants `a`, `b`, `c`, `d` and `e` denote the vertices of the tree. The predicate `branch` in `branch(a,b)` has the following intended meaning: 'there exists a branch from vertex `a` to vertex `b`'.

The clauses 5 and 6 for `path` specify under which conditions there exists a path between two vertices. The notion of a path has been defined recursively: the definition of a path makes use of the notion of a path again.

A *recursive definition* of a relation generally consists of two parts: one or more *termination criteria*, usually defining the basic states for which the relation holds, and the actual recursion describing how to proceed from a state in which the relation holds to a new, simpler state concerning the relation.

The termination criterion of the recursive definition of the `path` relation is expressed above in clause 5; the actual recursion is defined in clause 6. Note that the definition of the `member` relation in the preceding examples is also a recursive definition.

Now, suppose that after the above given program is entered into the Prolog database, we enter the following query:

```
/*7*/      ?- path(a,d).
```

The interpreter first tries to obtain a match with clause 5, the first clause in the database specifying the predicate `path` in its conclusion:

```
/*5*/      path(X,X).
```

For a match for the respective first argument positions, the variable `X` will be instantiated to the constant `a`. Matching the second argument positions fails, since `a`, the instantiation of `X`, and the constant `d` are different from each other. The interpreter therefore tries the next clause for `path`, which is clause 6:

```
/*6*/      path(X,Y) :- branch(X,Z),path(Z,Y).
```

It will now find a match for the query: the variable `X` occurring in the first argument position of the conclusion of clause 6 is instantiated to the constant `a` from the first argument position of the query, and the variable `Y` is instantiated to the constant `d`. These instantiations again pertain to the entire matching clause; in fact, clause 6 may now be looked upon as having the following instantiated form:

```
path(a,d) :- branch(a,Z),path(Z,d).
```

Before we may draw the conclusion of clause 6, we have to fulfil the two conditions `branch(a,Z)` and `path(Z,d)`. The interpreter deals with these new queries from left to right. For the query

```
?- branch(a,Z).
```

the interpreter finds a match with clause 1

```
/*1*/      branch(a,b).
```

by instantiating the variable `Z` to `b`. Again, this instantiation affects all occurrences of the variable `Z` in the entire clause containing the query; so, we have:

```
path(a,d) :- branch(a,b),path(b,d).
```

The next procedure call to be handled by the interpreter therefore is

```
?- path(b,d)
```

No match is found for this query with clause 5. The query however matches with the conclusion of clause 6:

```
/*6*/      path(X,Y) :- branch(X,Z),path(Z,Y).
```

The interpreter instantiates the variable `X` to `b`, and the variable `Y` to `d`, yielding the following instance of clause 6:

```
path(b,d) :- branch(b,Z),path(Z,d).
```

Note that these instantiations for the variables `X` and `Y` are allowed; the earlier instantiations for variables `X` and `Y` concerned *different* variables since they occurred in a different clause and therefore within a different scope. Again, before the query `path(b,d)` may be answered in the affirmative, we have to check the two conditions of the instance of clause 6 obtained. Unfortunately, the first condition

```
?- branch(b,Z).
```

does not match with any clause in the Prolog program (as can be seen in Figure 2.7, there is no outgoing branch from the vertex `b`).

The Prolog interpreter now cancels the last match and its corresponding instantiations, and tries to find a new match for the originating query. The match of the query `path(b,d)` with the conclusion of clause 6 was the last match found, so the corresponding instantiations to `X` and `Y` in clause 6 are cancelled. The interpreter now has to try to find a new match for the query `path(b,d)`. However, since clause 6 is the last clause in the program having the predicate `path` in its conclusion, there is no alternative match possible. The interpreter therefore goes yet another step further back.

The match of `branch(a,Z)` with clause 1 will now be undone by cancelling the instantiation of the variable `Z` to `b`. For the query

```
?- branch(a,Z).
```

the interpreter is able to find an alternative match, namely with clause 2:

```
/*2*/     branch(a,c).
```

It instantiates the variable `Z` to `c`. Recall that the query `branch(a,Z)` came from the match of the query `path(a,d)` with clause 6:

```
path(a,d) :- branch(a,Z),path(Z,d).
```

The undoing of the instantiation to `Z`, and the subsequent creation of a new instantiation again influences the entire calling clause:

```
path(a,d) :- branch(a,c),path(c,d).
```

Instead of the condition `path(b,d)` we therefore have to consider the condition `path(c,d)`. By means of successive matches with the clauses 6, 3 and 5, the interpreter derives the answer `yes` to the query `path(c,d)`. Both conditions to the match with the original query `path(a,d)` are now fulfilled. The interpreter therefore answers the original query in the affirmative.

This example illustrates the modus operandi of the Prolog interpreter, and, among other things, it was demonstrated that the Prolog interpreter examines clauses in the order in which they have been specified in the database. According to the principles of logic programming, a logic program is viewed as a set of clauses; so, their respective order is of no consequence to the derived results. As can be seen from the previous example, however, the order in which clauses have been specified in the Prolog database may be important. This is a substantial difference between a logic program and a Prolog program: whereas logic programs are purely declarative in nature, Prolog programs tend to be much more procedural. As a consequence, the programmer must bear in mind properties of the Prolog interpreter when developing a Prolog program. For example, when imposing some order on the clauses in the database, it is usually necessary that the clauses acting as a termination criterion for a recursive definition, or having some other special function, are specified before the clauses expressing the general rule.

2.5 Overview of the Prolog language

Until now, all predicates discussed in the examples have been defined on purpose. However, every Prolog system offers a number of predefined predicates, which the programmer may utilise in programs as desired. Such predicates are usually called *standard predicates* or *built-in predicates* to distinguish them from the predicates defined by the programmer.

In this section, we shall discuss several standard predicates and their use. Only frequently applied predicates will be dealt with here. A complete overview is usually included in the documentation concerning the particular Prolog system. This discussion is based on SWI-Prolog.

2.5.1 Reading in programs

By means of the predicate `consult` programs can be read from file and inserted into the Prolog database. The predicate `consult` takes one argument which has to be instantiated to the name of a file before execution.

EXAMPLE 2.18

The query

```
?- consult(file).
```

instructs the interpreter to read a Prolog program from the file with the name `file`.

It is also possible to insert into the database several programs from different files. This may be achieved by entering the following clause:

?- consult(file$_1$),...,consult(file$_n$).

Prolog offers an abbreviation for such a clause; the required file names may be specified in a list:

?- [file$_1$,...,file$_n$].

2.5.2 Input and output

Printing text on the screen can be done by means of the predicate `write` which takes a single argument. Before execution of the procedure call `write(X)`, the variable `X` must be instantiated to the term to be printed.

EXAMPLE 2.19

The clause

```
?- write(output).
```

prints the term `output` on the screen. Execution of the call

```
?- write('This is output.').
```

results in

```
This is output.
```

When the clause

```
?- create(Output),write(Output).
```

is executed, the value to which `Output` is instantiated by a call to some user-defined predicate `create` will be printed on the screen. If the variable `Output` is instantiated to a term containing uninstantiated variables, then (the internal representation of) the variables will be shown as part of the output.

The predicate `nl` just prints a new line, causing output to start at the beginning of the next line. `nl` takes no arguments.

We also have some means for input. The predicate `read` reads terms entered from the keyboard. The predicate `read` takes only one argument. Before executing the call `read(X)`, the variable `X` has to be uninstantiated; after execution of the `read` predicate, `X` will be instantiated to the term that has been entered. A term entered from the keyboard has to end with a dot, followed by a carriage return.

2.5.3 Arithmetical predicates

Prolog provides a number of arithmetical predicates. These predicates take as arguments arithmetical expressions; arithmetical expressions are constructed as in usual mathematical practice, that is, by means of infix operators, such as `+`, `-`, `*` and `/`, for addition, subtraction, multiplication, and division, respectively. Generally, before executing an arithmetical predicate all variables in the expressions in its left-hand and right-hand side have to be instantiated to terms only containing numbers and operators; the arguments will be evaluated before the test specified by means of the predicate is performed. For example, in a condition `X < Y` both `X` and `Y` have to be instantiated to terms which upon evaluation yield numeric constants, before the comparison is carried out. The following arithmetical relational predicates are the ones most frequently used:

```
X > Y.
X < Y.
X >= Y.
X =< Y.
X =:= Y.
X =\= Y.
```

The last two predicates express equality and inequality, respectively. Note that the earlier mentioned matching predicate `=` is not an arithmetical predicate; it is a more general predicate the use of which is not restricted to arithmetical expressions. Furthermore, the predicate `=` does not force evaluation of its arguments.

Besides the six arithmetical relational predicates shown above, we also have in Prolog an infix predicate with the name `is`. Before executing

```
?- X is Y.
```

only the right-hand side `Y` has to be instantiated to an arithmetical expression. Note that the `is` predicate differs from `=:=` as well as from the matching predicate `=`; in case of `=:=` both `X` and `Y` have to be instantiated to arithmetical expressions, and in case of the matching predicate neither `X` nor `Y` has to be instantiated. If in the query shown above `X` is an uninstantiated variable, it will after execution of the query be instantiated to the value of `Y`. The values of both left-hand and right-hand side are subsequently examined upon equality; it is obvious that this test will always succeed. If, on the other hand, the variable `X` is instantiated to a number (or the left-hand side itself is a number), then the condition succeeds if the result of evaluating the right-hand side of `is` equals the left-hand side, and it fails otherwise. All other uses of the predicate `is` lead to a syntax error.

EXAMPLE 2.20

Consider the following queries and answers which illustrate the differences and similarities between the predicates `=`, `=:=`, and `is`:

```
?- 3 = 2+1.
no

?- 3 is 2+1.
yes

?- 3 =:= 2+1.
yes

?- 3+1 = 3+1.
yes

?- 3+1 =:= 3+1.
yes

?- 3+1 is 3+1.
no
```

```
?- 1+3 = 3+1.
no

?- 1+3 =:= 3+1.
yes
```

The following examples illustrate the behaviour of these predicates in case the left-hand side is an uninstantiated variable. Prolog returns by showing the computed instantiation:

```
?- X is 2+1.
X = 3

?- X = 2+1.
X = 2+1
```

We have left out the example `?- X =:= 2+1`, since it is not permitted to have an uninstantiated variable as an argument to `=:=`.

The predicates `=:=` and `is` may only be applied to arithmetical arguments. The predicate `=` however, also applies to non-arithmetical arguments, as has been shown in Section 2.4.2.

EXAMPLE 2.21

Execution of the query

```
?- X = [a,b].
```

leads to the instantiation of the variable `X` to the list `[a,b]`. In case the predicate `=:=` or the predicate `is` would have been used, the Prolog interpreter would have signalled an error.

2.5.4 Examining instantiations

A number of predicates is provided which can be used to examine a variable and its possible instantiation. The predicate `var` taking one argument, investigates whether or not its argument has been instantiated. The condition `var(X)` is fulfilled if `X` at the time of execution is uninstantiated; otherwise, the condition fails. The predicate `nonvar` has a complementary meaning.

By means of the predicate `atom`, also taking one argument, it can be checked whether the argument is instantiated to a symbolic constant. The predicate `atomic`, which also takes a single argument, investigates whether its argument is instantiated to a symbolic or numeric constant. The one-argument predicate `integer` tests if its argument is instantiated to an integer.

EXAMPLE 2.22

Consider the following queries specifying the predicates mentioned above, and answers of the Prolog interpreter:

```
?- atomic([a]).
no

?- atomic(3).
yes

?- atom(3).
no

?- atom(a).
yes

?- integer(a).
no
```

2.5.5 Controlling backtracking

Prolog offers the programmer a number of predicates for explicitly controlling the backtracking behaviour of the interpreter. Note that here Prolog deviates from the logic programming idea.

The predicate `call` takes one argument, which before execution has to be instantiated to a procedure call; `call` takes care of its argument being handled like a procedure call by the Prolog interpreter in the usual way. Note that the use of the `call` predicate allows for 'filling in' the program during run-time.

The predicate `true` takes no arguments; the condition `true` always succeeds. The predicate `fail` also has no arguments; the condition `fail` never succeeds. The general application of the predicate `fail` is to enforce backtracking, as shown in the following example.

EXAMPLE 2.23

Consider the following clause:

```
a(X) :- b(X),fail.
```

When the query `a(X)` is entered, the Prolog interpreter first tries to find a match for `b(X)`. Let us suppose that such a match is found, and that the variable `X` is instantiated to some term. Then, in the next step `fail`, as a consequence of its failure, enforces the interpreter to look for an alternative instantiation to `X`. If it succeeds in finding another instantiation for `X`, then again `fail` will be executed. This entire process is repeated until no further instantiations can be found. This way all possible instantiations for `X` will be found. Note that if no side-effects are employed to record the instantiations of `X` in some way, the successive instantiations leave no trace. It will be evident that in the end the query `a(X)` will be answered by `no`.

The predicate `not` takes a procedure call as its argument. The condition `not(P)` succeeds if the procedure call to which `P` is instantiated fails, and vice versa. Contrary to what one would expect in case of the ordinary logical negation, Prolog does not look for facts `not(P)` in the database (these are not even allowed in Prolog). Instead, negation is handled by confirming failed procedure calls. This form of negation is known as *negation as (finite) failure*; for a more detailed discussion of this notion the reader is referred to [13].

The *cut*, denoted by `!`, is a predicate without any arguments. It is used as a condition which can be confirmed only once by the Prolog interpreter: on backtracking it is not possible to confirm a cut for the second time. Moreover, the cut has a significant side effect on the remainder of the backtracking process: it enforces the interpreter to reject the clause containing the cut, and also to ignore all other alternatives for the procedure call which led to the execution of the particular clause.

EXAMPLE 2.24

Consider the following clauses:

```
/* 1 */     a :- b,c,d.
/* 2 */     c :- p,q,!,r,s.
/* 3 */     c.
```

Suppose that upon executing the call `a`, the successive procedure calls `b`, `p`, `q`, the cut and `r` have succeeded (the cut by definition always succeeds on first encounter). Furthermore, assume that no match can be found for the procedure call `s`. Then as usual, the interpreter tries to find an alternative match for the procedure call `r`. For each alternative match for `r`, it again tries to find a match for condition `s`. If no alternatives for `r` can be found, or similarly if all alternative matches have been tried, the interpreter normally would try to find an alternative match for `q`. However, since we have specified a cut between the procedure calls `q` and `r`, the interpreter will not look for alternative matches for the procedure calls preceding `r` in the specific clause. In addition, the interpreter will not try any alternatives for the procedure call `c`; so, clause 3 is ignored. Its first action after encountering the cut during backtracking is to look for alternative matches for the condition preceding the call `c`, that is, for `b`.

There are several circumstances in which specification of the cut is useful for efficiency or even necessary for correctness. In the first place, the cut may be used to indicate that the selected clause is the only one that can be applied to solve the (sub)problem at hand, that is, it may be used to indicate 'mutually exclusive' clauses.

EXAMPLE 2.25

Suppose that the condition `b` in the following clause has been confirmed:

```
a :- b,c.
```

and that we know that this clause is the only one in the collection of clauses having `a` as a conclusion, which is applicable in the situation in which `b` has been confirmed.

When the condition `c` cannot be confirmed, there is no reason to try any other clause concerning `a`: we already know that `a` will never succeed. This unnecessary searching can be prevented by specifying the cut following the critical condition:

```
a :- b,!,c.
```

Furthermore, the cut is used to indicate that a particular procedure call may never lead to success if some condition has been fulfilled, that is, it is used to identify exceptional cases to a general rule. In this case, the cut is used in combination with the earlier mentioned predicate `fail`.

EXAMPLE 2.26

Suppose that the conclusion `a` definitely may not be drawn if the condition `b` succeeds. In the clause

```
a :- b,!,fail.
```

we have used the cut in conjunction with `fail` to prevent the interpreter to look for alternative matches for `b`, or to try any other clause concerning `a`.

We have already remarked that the Prolog programmer has to be familiar with the workings of the Prolog interpreter. Since the cut has a strong influence on the backtracking process, it should be applied with great care. The following example illustrates to what errors a careless use of the cut may lead.

EXAMPLE 2.27

Consider the following three clauses, specifying the number of parents of a person; everybody has two of them, except Adam and Eve, who have none:

```
/* 1 */      number_of_parents(adam,0) :- !.
/* 2 */      number_of_parents(eve,0) :-  !.
/* 3 */      number_of_parents(X,2).
```

Now, the query

```
?- number_of_parents(eve,2).
```

is answered by the interpreter in the affirmative. Although this is somewhat unexpected, after due consideration the reader will be able to figure out why `yes` instead of `no` has been derived.

For convenience, we summarise the side-effects of the cut:

- If in a clause a cut has been specified, then we have normal backtracking over the conditions preceding the cut.

- As soon as the cut has been 'used', the interpreter has committed itself to the choice for that particular clause, and for everything done after calling that clause; the interpreter will not reconsider these choices.
- We have normal backtracking over the conditions following the cut.
- When on backtracking a cut is met, the interpreter 'remembers' its commitments, and traces back to the originating query containing the call which led to a match with the clause concerned.

We have seen that all procedure calls in a Prolog clause will be executed successively, until backtracking emerges. The procedure calls, that is, the conditions are connected by commas, which have the declarative semantics of the logical $\wedge$. However, it is also allowed to specify a logical $\vee$ in a clause. This is done by a semicolon, `;`, indicating a choice between conditions. All conditions connected by `;` are evaluated from left to right until one is found that succeeds. The remaining conditions will then be ignored. The semicolon has higher precedence than the comma.

2.5.6 Manipulation of the database

Any Prolog system offers the programmer means for modifying the content of the database during run-time. It is possible to add clauses to the database by means of the predicates `asserta` and `assertz`. Both predicates take one argument. If this argument has been instantiated to a term before the procedure call is executed, `asserta` adds its argument as a clause to the database before all (possibly) present clauses that specify the same functor in their conclusions. On the other hand, `assertz` adds its argument as a clause to the database just after all other clauses concerning the functor.

EXAMPLE 2.28

Consider the Prolog database containing the following clauses:

```
fact(a).
fact(b).
yet_another_fact(c).
and_another_fact(d).
```

We enter the following query to the system:

```
?- asserta(yet_another_fact(e)).
```

After execution of the query the database will have been modified as follows:

```
fact(a).
fact(b).
yet_another_fact(e).
yet_another_fact(c).
and_another_fact(d).
```

Execution of the procedure call

```
?- assertz(fact(f)).
```

modifies the contents of the database as follows:

```
fact(a).
fact(b).
fact(f).
yet_another_fact(e).
yet_another_fact(c).
and_another_fact(d).
```

By means of the one-placed predicate `retract`, the first clause having both conclusion and conditions matching with the argument, is removed from the database.

2.5.7 Manipulation of terms

Terms are used in Prolog much in the same way as records are in Pascal, and structures in C. In these languages, various operations are available to a programmer for the selection and modification of parts of these data structures. Prolog provides similar facilities for manipulating terms. The predicates `arg`, `functor` and `=..` (pronounced as 'univ') define such operations.

The predicate `arg` can be applied for selecting a specific argument of a functor. It takes three arguments:

```
arg(I,T,A).
```

Before execution, the variable `I` has to be instantiated to an integer, and the variable `T` must be instantiated to a term. The interpreter will instantiate the variable `A` to the value of the `I`-th argument of the term `T`.

EXAMPLE 2.29

The procedure call:

```
arg(2,employee(john,mccarthy),A)
```

leads to instantiation of the variable `A` to the value `mccarthy`.

The predicate `functor` can be used for selecting the left-most functor in a given term. The predicate `functor` takes three arguments:

```
functor(T,F,N).
```

If the variable `T` is instantiated to a term, then the variable `F` will be instantiated to the functor of the term, and the variable `N` to the number of arguments of the functor.

EXAMPLE 2.30

The procedure call

```
functor(employee(john,mccarthy),F,N).
```

leads to instantiation of the variable `F` to the constant `employee`. The variable `N` will be instantiated to the integer 2.

The predicate `functor` may also be applied in a 'reverse mode': it can be employed for constructing a term with a given functor `F` and a prespecified number of arguments `N`. All arguments of the constructed term will be variables.

The predicate `=..` also has a dual function. It may be applied for selecting information from a term, or for constructing a new term. If in the procedure call

```
X =.. L.
```

`X` has been instantiated to a term, then after execution the variable `L` will be instantiated to a list, the first element of which is the functor of `X`; the remaining elements are the successive arguments of the functor.

EXAMPLE 2.31

Consider the following procedure call:

```
employee(john,mccarthy,[salary=10000]) =.. L.
```

This call leads to instantiation of the variable `L` to the list

```
[employee,john,mccarthy,[salary=10000]]
```

The predicate `=..` may also be used to organize information into a term. This is achieved by instantiating the variable `L` to a list. Upon execution of the call `X =.. L`, the variable `X` will be instantiated to a term having a functor which is the first element from the list; the remaining elements of the list will be taken as the arguments of the functor.

EXAMPLE 2.32

The procedure call

```
X =.. [employee,john,mccarthy,[salary=10000]].
```

leads to instantiation of the variable `X` to the term

```
employee(john,mccarthy,[salary=10000]).
```

Note that, contrary to the case of the predicate `functor`, in case of the predicate `=..` prespecified arguments may be inserted into the new term.

To conclude this section, we consider the predicate `clause`, which can be used for inspecting the contents of the database. The predicate `clause` takes two arguments:

```
clause(Head,Body).
```

The first argument, `Head`, must be sufficiently instantiated for the interpreter to be able to find a match with the conclusion of a clause; the second argument, `Body`, will then be instantiated to the conditions of the selected clause. If the selected clause is a fact, `Body` will be instantiated to `true`.

2.6 Suggested reading and available resources

Readers interested in the theoretical foundation of Prolog and logic programming should consult Lloyd's *Foundations of Logic Programming* [13]. Prolog is one of the few programming language with a simple formal semantics. This is mainly due to the declarative nature of the language. Students of computing *science* should know at least something of this semantics. A good starting point for the study of this semantics is knowledge of logical deduction in predicate logic [14].

An excellent introductory book to programming in Prolog, with an emphasis on Artificial Intelligence applications, is [10].

The Prolog community has its own Usenet newsgroup: `comp.lang.prolog`. There are quite a number of Prolog programs in the public domain which researchers can use in their own research. SWI-Prolog is a good complete Prolog interpreter and compiler, which is freely available for Linux, MacOS, and Windows at:

```
http://www.swi-prolog.org
```

Chapter 3

Lecture 5 – Description Logics and Frames

An important and popular application of knowledge representation and reasoning is the description of the objects or things that exist in the real world. Much of this research drives new developments around the World Wide Web, in particular the development of the Semantic Web.

3.1 Introduction

Research on description logics drives most of the work on large-scale knowledge representation, such as for the World Wide Web. The main current application area for this research is biomedicine, as there is now so much known about biomolecules and their interaction that there is a strong need by biomedical scientists of having computer systems available for representing and reasoning with this knowledge. Description logics are related to so-called frame languages, a collection of earlier formalisms for the representation and reasoning with objects in the world also coming from AI. Both formalisms are covered in the lectures. A knowledge base containing a representation of a problem domain by means of a description logic or frame language is usually called an *ontology*. We start with description logics and subsequently discuss frames and the relationship between frames and description logics.

3.2 Description logics

During the last couple of years, in particular ongoing work on the *Semantic Web* has had an enormous impetus on the development these languages, as description logics and frames act as the basis for it. OWL (Web Ontology Language) DL is the specific description logic that has been developed by the WWW Consortium. The language is XML based, but for the rest very similar to the description logic ALC (Attribute concept Language with Complement) discussed during the course.

3.2.1 Knowledge servers

There are currently a number of implementations available of reasoning engines for description logics, and these are often remotely accessible on the Internet. Usually, the implemen-

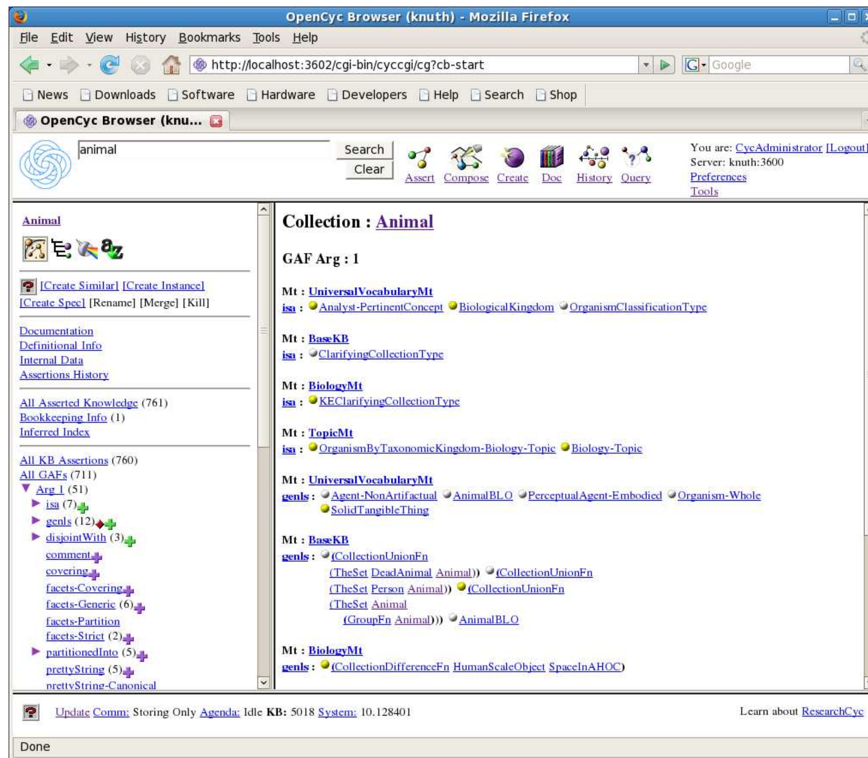

Figure 3.1: Web interface used by OpenCyc.

tations come with extensive knowledge bases built using the specific description logic as a language. The reasoning engine, knowledge bases, and user interfaces offered in this way are together called *knowledge servers*. A well-known example of a knowledge server is OpenCyc (www.opencyc.com); Figure 3.1 shows the web interface used by OpenCyc.

3.2.2 Basics of description logics

There are descriptions logics with varying expressive power. For example, a description logic that offers concepts, roles (attributes), negation (complement) is called ALC (Attribute concept Language with Complement. One of the main design considerations of description logics is to make sure that the resulting language is *decidable*, i.e., one always get an answer to a query (although it may take a long while). Recall that first-order predicate logic is undecidable *in general*, and is thus less suitable as a language for use by non-experts.

The basic ingredients of expressions in ALC are:

- concepts, the entities in the domain of concern;
- roles, relationships between concepts;
- Boolean operators, which allow combining concepts.

Consider the following phrase:

> “A man is married to a doctor, and all of whose children are either doctors or professors”

In ACL, this phrase looks as follows:

$$\text{Human} \sqcap \neg\text{Female} \sqcap \exists\text{married.Doctor}) \sqcap (\forall\text{hasChild.}(\text{Doctor} \sqcup \text{Professor}))$$

In the following we explain how such an expression should be read. Syntactically, 'married' and 'hasChild' are examples of roles, whereas all other symbols, such as 'Human' and 'Doctor' are concepts.

There are two primary ways in which knowledge is being described using ALC:

1. by *combining concepts* using Boolean operators, such as $\sqcap$ (conjunction), and $\sqcup$ (disjunction);
2. by *defining relationships* between concepts (whether primitive or obtained by combining primitive concepts) using the *subsumption* relation $\sqsubseteq$ (also called *general concept inclusion* – GCI).

Thus, a concept description is constructed from

- *primitive concepts* C, e.g., Human, $\top$ (most general), $\bot$ (empty);
- *primitive roles* r, e.g., hasChild;
- *conjunctions* $\sqcap$, e.g., SmartHuman $\sqcap$ Student;
- *disjunctions* $\sqcup$, e.g., Truck $\sqcup$ Car;
- a *complement* $\neg$, e.g., $\neg$Human;
- a *value restriction* $\forall r.C$, e.g., $\forall$hasChild.Doctor;
- an *existential restriction* $\exists r.C$, e.g., $\exists$happyChild.Parent.

All understood in terms of (groups of) individuals and properties of individuals.

EXAMPLE 3.1

For example, by

$$\text{Student} \sqcup \text{Professor}$$

we have combined two concepts, but we have not established how they are related to each other. By writing:

$$\text{Student} \sqsubseteq \text{Person}$$

we have established a relationship between the concepts 'Student' and 'Person', where the first is less or equally general than the latter. By combining two subsumption relations, it is possible to *define* a new concept:

$$\text{Father} \sqsubseteq \neg\text{Female} \sqcap \exists\text{hasChild.Human}$$

and

$$\neg\text{Female} \sqcap \exists\text{hasChild.Human} \sqsubseteq \text{Father}$$

is abbreviated to

$$\text{Father} \equiv \neg\text{Female} \sqcap \exists\text{hasChild.Human}$$

Note that an expression such as ∃hasChild.Human is also a concept: the role hasChild establishes a relationship between an instance of the concept Human (the child) and all concepts that participate in the role, which are then intersected with the concept 'Man', represented as ¬Female, yielding a definition of 'Father'.

General descriptions of a domain form, what is called, the TBox (Terminology Box). In a sense, the TBox restricts the terminology we are allowed to use when describing a domain. The actual domain is described by means of *assertions*, which together form the ABox (Assertion Box). For example,

$$\text{Allen} : \text{Person}$$

asserts that the Allen is a person, i.e., an instance of the concept 'Person'. Similarly, instances of *roles* can be used to establish relationships between instances (see slides). Thus, a *knowledge base* KB is defined as the pair $\text{KB} = (\text{TBox}, \text{ABox})$.

3.2.3 Meaning of description logics

The notations used in description logic look very similar to *set theory.* It, therefore, does not come as a surprise that the semantics of description logic is indeed defined in terms of set theory.

The general idea is to start with the entire collection of objects with respect to which one wishes to interpret the description-logic formulae, denoted by Δ. Each concept is then interpreted by associating with it a subset from Δ. For example, the concept $P \sqsubseteq \text{Person}$ may at first sight be unclear, but if we now define

$$\Delta = \{\text{Pieter}, \text{Peter}, \text{John}\},$$

with $P^{\mathcal{I}} = \{\text{Pieter}, \text{Peter}\} \subseteq \Delta$, then it is clear that, according to the interpretation $\mathcal{I}$, the concept P stands for all persons, whose name start with the letter 'P'.

As roles r stand for binary relations between concepts, it also should not come as a surprise that the meaning of a role is defined as $r^{\mathcal{I}} \subseteq \Delta \times \Delta$.

To summarise, let $\mathcal{I} = (\Delta, \cdot)$ be an *interpretation*, then

- $\top^{\mathcal{I}} = \Delta$, and $\bot^{\mathcal{I}} = \varnothing$;
- each concept $C^{\mathcal{I}} \subseteq \Delta$;
- each role $r^{\mathcal{I}} \subseteq \Delta \times \Delta$;
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$;
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$;
- $(\exists r.C)^{\mathcal{I}} = \{c \in \Delta \mid \exists d \in C^{\mathcal{I}} \text{ with } (c, d) \in r^{\mathcal{I}}\}$;
- $(\forall r.C)^{\mathcal{I}} = \{c \in \Delta \mid \forall d \in \Delta, \text{if } (c, d) \in r^{\mathcal{I}} \text{ then } d \in C^{\mathcal{I}}\}$,

where $C^{\mathcal{I}}$ and $r^{\mathcal{I}}$ are interpretations of C and r as sets. See the slides for examples. Note that the result of the existential restriction formula $\exists r.C$ is all elements of Δ for which there exists a concept d that takes part in the role r. A similar interpretation is obtained for value restriction formulae $\forall r.C$.

EXAMPLE 3.2

The formula

$$\exists \text{happyChild.Parent}$$

the 'happyChild' expresses a binary relationship between all children (or any other more general concept, as the actual concept is not indicated in the expression) and at least one parent (this is the concept to which the existential quantifier $\exists$ applies). Thus, this expression yields all children (concepts) that have at least one parent, and are happy children according to the role 'happyChild'.

As the meaning of predicate logic is usually also defined in terms of set theory (see Appendix A, Definitions A.10–A.12 and Example A.15), it is also possible to use predicate logic to better understand the meaning of description logics. Such a translation could also be used to save time and to use a standard reasoning engine for predicate logic for the translated formulae in description logic, rather than implementing a special-purpose reasoning engine for description logics. For example, a predicate P in predicate logic plays the role of a relation in set theory. So, when the predicate is unary, i.e., takes only one argument, then this relation corresponds to a unary relation, that is, a set. Hence, where the concept 'Person' in description logic is interpreted as a set (of persons), in predicate logic, the representation $\text{Person}(x)$ is used, where the free variable x stands for an actual person. This explains the translation rule:

$$\tau_x(C) = C(x)$$

where τ_x is the translation from description logic to predicate logic used in the slides.

Furthermore, the Boolean operators, such as $\sqcap$ (conjunction), have a natural interpretation in predicate logic. For example, $\sqcap$ is interpreted as $\wedge$, thus:

$$\tau_x(C \sqcap D) = (\tau_x(C) \wedge \tau_x(D))$$

and as a consequence of the rule for concepts, we get:

$$\tau_x(C \sqcap D) = (C(x) \wedge D(x))$$

See the slides for the interpretation of the other operators and of subsumption $\sqsubseteq$.

3.2.4 Reasoning

As mentioned above, one way to reason with a description logic knowledge base is to translate the formulae into predicate logic, and next to use one of the many reasoning engines available for predicate logic. Note that if one wishes to use resolution as the basic inference rules, the description logic formulae need to be translated into clausal form and one uses refutation in order to derive a formula. Thus, rather than

$$\tau(\text{KB}) \vDash \varphi$$

one tries to derive an inconsistency by adding the negation of φ to KB, i.e.,

$$\tau(\text{KB}) \wedge \neg\varphi \models \bot$$

where τ(KB) is the translation of the description logic knowledge base KB into clausal form.

Usually, special purpose inference methods are used, as only then is it possible to take into account the restricted nature of most description logics. This has a positive effect on computational complexity and decidability.

An example of such an algorithm is the *completion forest* algorithm. The basic idea underlying the algorithm is to reason about individual cases, i.e., instances. A similar way of reasoning is often carried out in the context of predicate logic. Suppose, for example, that we have the following formulae in predicate logic:

$$\forall x(P(x) \rightarrow Q(x))$$
$$(P(a) \wedge \neg Q(a))$$

One simple way to prove inconsistency for these formulae is to first assume that $P(a)$ is true (which follows directly), and then to derive $Q(a)$ from the logical implication and $P(a)$. Next assuming that $\neg Q(a)$ is true, the inconsistency follows from $(Q(a) \wedge \neg Q(a))$. Thus, reasoning from individual instances is often effective in proving inconsistency. Similarly, the completion forest algorithm starts with the instances from the ABox, and relates these to other instances using instances of roles in the ABox. It then using reasoning from the Boolean operators, such as $\sqcap$, to derive properties for individual instances of concepts, which, as in the example above for predicate logic, may give rise to a clash (inconsistency), if the concepts C and $\neg C$ are members of the same node label. This way of reasoning is another example of reasoning by refutation, as in resolution.

3.3 Frames

Similar to description logics, the frame formalism has been introduced in AI in order to represent and reason about objects in the real world. The frame formalism is more restrictive than most description logics, and is, in addition, characterised by a specific reasoning method, called *inheritance*.

3.3.1 Definition

A *frame* is a statement having the following form:

⟨frame⟩ ::= ⟨class⟩ | ⟨instance⟩

⟨class⟩ ::= **class** ⟨class-name⟩ **is**
    superclass ⟨super-specification⟩;
    ⟨class-attributes⟩
  end

⟨instance⟩ ::= **instance** ⟨instance-name⟩ **is**
    instance-of ⟨super-specification⟩;
    ⟨instance-attributes⟩

end

⟨super-specification⟩	::=	⟨class-name⟩ | **nil**
⟨class-attributes⟩	::=	⟨declaration⟩ {; ⟨declaration⟩}* | ⟨empty⟩
⟨instance-attributes⟩	::=	⟨attribute-value-pair⟩ {; ⟨attribute-value-pair⟩}* | ⟨empty⟩
⟨declaration⟩	::=	⟨attribute-value-pair⟩
⟨attribute-value-pair⟩	::=	⟨attribute-name⟩ = ⟨value⟩
⟨value⟩	::=	⟨elementary-constant⟩ | ⟨instance-name⟩
⟨empty⟩	::=	

A ⟨super-specification⟩ equal to the special symbol **nil** is used to indicate that the frame concerned is the root of the tree-shaped taxonomy. As a type, a ⟨set⟩ consists of elementary constants and instance names, separated by comma's and enclosed in curly brackets. An elementary constant is either a real or integer constant, or a string of non-blank characters, that is, an instance of one of the predefined (or standard) classes **real**, **int**, and **string**. The ⟨instance-name⟩ value of an attribute refers to a uniquely defined instance in the taxonomy.

EXAMPLE 3.3

Consider the following example: the aorta is an artery having a diameter of 2.5 cm. Using our frame formalism, this information may be represented as follows

instance *aorta* **is**
  instance-of *artery*;
  diameter = 2.5
end

The information that an artery is a blood vessel having a muscular wall is represented in the following class frame:

class *artery* **is**
  superclass *blood-vessel*;
  wall = *muscular*
end

3.3.2 Semantics

The semantics of first-order predicate logic can be exploited to define a semantics for the frame formalism. Under the assumption that each attribute only occurs once in the taxonomy, we may ascribe a meaning based on first-order predicate logic to the set of frames of this taxonomy using the following general translation scheme:

class C **is**
  superclass S; $\forall x(C(x) \rightarrow S(x))$
  $a_1 = b_1$; $\Rightarrow$ $\forall x(C(x) \rightarrow a_1(x) = b_1)$
  $\vdots$ $\vdots$
  $a_n = b_n$ $\forall x(C(x) \rightarrow a_n(x) = b_n)$
end

instance I **is**
  instance-of C; $C(I)$
  $a_1 = b_1$; $\Rightarrow$ $a_1(I) = b_1$
  $\vdots$ $\vdots$
  $a_n = b_n$ $a_n(I) = b_n$
end

Note that there is a (slight) difference between the semantics of frames and the predicate logic translation. When two classes are inconsistent according inheritance with exceptions using the definition of a conclusion set $C(\Omega_T)$, the predicate logic representation may be consistent.

EXAMPLE 3.4 ___

Suppose that we have the following taxonomy T:

class C **is**
  superclass nil;
  $a = 1$
end

class D **is**
  superclass C;
  $a = 2$
end

The corresponding formulae in predicate logic are as follows:

$$\forall x(C(x) \rightarrow a(x) = 1)$$
$$\forall x(D(x) \rightarrow a(x) = 2)$$
$$\forall x(D(x) \rightarrow C(x))$$

Using *inheritance chains* (see slides and below), we would be able to compute the following conclusion set $C(\Omega_T) = \{C[a = 1], D[a = 1], D[a = 2]\}$ from the taxonomy T, which would be inconsistent, as 1 and 2 are different values for the same attribute a of class D. However, the formulae in predicate logic above are *consistent*. (Using resolution, we would be able to derive $\neg C(x) \vee \neg D(x)$ by cancelling the literals $a(x) = 1$ and $a(x) = 2$, and subsequently from $\neg C(x) \vee \neg D(x)$ and $\neg D(x) \vee C(x)$ (obtained by translating $\forall x(D(x) \rightarrow C(x))$ into clausal form), we derive $\neg D(x)$. However, as soon as we add an instance of class D we also get inconsistency. For example, let us add to T:

instance I **is**

instance-of D
end

Which corresponds to the formula $D(I)$ in predicate logic, we get an inconsistence with $\neg D(x)$ just derived.

Thus, the semantics of frames is as if there is always at least one instance available for each class, even if this is not the case in reality. This semantics is from a practical point of view better than that of predicate logic, as one will to meet unexpected inconsistencies by adding instances in frame theory, which in predicate logic may occur. This example also shows that it is useful to think afresh about semantics of formal languages rather than simply accepting that others have already made a choice, as in predicate logic.

3.3.3 Relationship with description logic

As frames and description logic are related in the way they are used, one may wonder whether it is possible to establish a relationship between the two formalisms. One way to answer this question is by translating both to predicate logic and to compare the resulting formulae. However, one would also expect it to be possible to translate frames directly into description logic, as description logics are in general more expressive than frame languages. However, it appears that this is only possible after making a slight addition to the syntax of the description logics discussed so far.

Consider the following translation table for classes, which will give elements in the TBox of a knowledge base:

	TBox:
class C **is**	$C \sqsubseteq S$
superclass S;	
$a_1 = b_1$;	$C \sqsubseteq \exists a_1.\{b_1\}$
$\vdots$	$\vdots$
$a_n = b_n$	$C \sqsubseteq \exists a_n.\{b_n\}$
end	

To make the translation possible it is necessary to introduce concepts that only include a *single* element; these are indicates by the notation $\{b_k\}$ with an existential quantifier $\exists$, i.e., $\exists a_k.b_k$, which does nothing in this case, as the concept contains exactly one element which thus always exists.

The translation for instances yields elements for the ABox of a knowledge base. The resulting table is the following:

	ABox:
instance i **is**	$i : C$
instance-of C;	
$d_1 = e_1$;	$(i, e_1) : d_1$
$\vdots$	$\vdots$
$d_m = e_m$	$(i, e_m) : d_m$
end	

Here $(i, e_k) : d_k$ indicate instances (i, e_k) of roles d_k.

EXAMPLE 3.5

The instance:

instance *aorta* **is**
instance-of *artery*;
diameter $= 2.5$
end

would yield the following ABox:

$$\{aorta : artery,\ \ (aorta, 2) : diameter\}$$

3.3.4 Reasoning — inheritance

Basically, inheritance is nothing but reasoning from classes to superclasses, and collecting all attributes with associated value for the class where the reasoning started. In principle, this form of reasoning is straightforward. However, problems arise when there are inconsistencies between values that may be inherited by attributes of a class. For example, an attribute a may inherit values 1 and 2 (assuming that the attribute is *single valued, i.e., takes only a single elementary value*). Then, either an inconsistency must be signalled or a choice must be made between the two values. The algorithm of inheritance with *exceptions* is an example of an algorithm that is able to make such choices based on the relative position of a class in the frame taxonomy.

In *single inheritance with exceptions* the taxonomy has the shape of a tree and the problem is, therefore, easily solved. Start with the class for which one wishes to infer all relevant attribute-value pairs, and collect all information travelling on the path from the class to the root of the tree-shaped taxonomy. If one meets another value for an attribute for which already a value has been obtained, it is simply ignored (see the slides for the algorithm of single inheritance with exceptions).

The algorithm for *multiple inheritance with exceptions* is more difficult, as in this case the taxonomy does not have the shape of a tree. To establish the relative position of a class in the frame taxonomy with respect to all the other classes, the idea to use a representation of the paths from the class to the other classes, including the root, are used. These representations are called *inheritance chains*. Basically, an inheritance chain is a potential way for inheritance of attribute values. Of course, the mathematics of inheritance chains should also work for single inheritance withe exceptions, even though, strictly speaking, it is not needed.

From here to the end of chapter 3, reading is optional, as this material was not fully covered in the lectures.

Formally, the definition is as follows. Let $T = (N, \Theta, \ll, \leq)$ be a taxonomy, where $N = (I, K, A, C)$ is the set of instances I, classes K, attributes A, and constants C; $\ll$ is the instance-of function and $\leq$ the subclass relation.

Definition 3.1 *Let F be the set of frames such that $F = I \cup K$, where I is the set of instance frames in F, and K the set of class frames in F. Let A be a fixed set of attribute names and*

let C be a fixed set of constants. Then, a triple $(x, a, c) \in F \times A \times C$, denoted by $x[a = c]$, is called an attribute-value specification. *An* attribute-value relation Θ *is a ternary relation on F, A and C, that is, $\Theta \subseteq F \times A \times C$.*

We give an example of the frame formalism we have just defined and its relation with the frame formalism introduced above.

EXAMPLE 3.6

Consider the information specified in the following three classes:

class *blood-vessel* **is**
  superclass nil;
  contains = *blood-fluid*
end

class *artery* **is**
  superclass *blood-vessel*;
  blood = *oxygen-rich*;
  wall = *muscular*
end

class *vein* **is**
  superclass *blood-vessel*;
  wall = *fibrous*
end

instance *aorta* **is**
  instance-of *artery*;
  diameter = 2.5
end

In the specified taxonomy, we have that $I = \{aorta\}$ is the set of instance frames and that $K = \{artery, vein, blood\text{-}vessel\}$ is the set of classes. Furthermore, we have that $A = \{contains, blood, wall, diameter\}$, and $C = \{blood\text{-}fluid, oxygen\text{-}rich, muscular, fibrous, 2.5\}$. We have the following set of attribute-value specifications:

$$\begin{aligned}\Theta = \{&blood\text{-}vessel[contains = blood\text{-}fluid],\\ &artery[blood = oxygen\text{-}rich],\\ &artery[wall = muscular],\\ &vein[wall = fibrous],\\ &aorta[diameter = 2.5]\}\end{aligned}$$

The function $\ll$ and the relation $\leq$ are defined by

$aorta \ll artery$
$artery \leq blood\text{-}vessel$
$vein \leq blood\text{-}vessel$

Now, $T = (N, \Theta, \ll, \leq)$ is the taxonomy shown above, this time represented using our new formalism.

A taxonomy $T = (N, \Theta, \ll, \leq)$ can be represented graphically by means of a graph in which the vertices represent the frames in I and K, and the arcs represent the relation $\leq$ and the function $\ll$. A vertex is assumed to have an internal structure representing the collection of attribute-value specifications associated with the frame by the relation Θ. In the graphical representation, an attribute-value specification is depicted next to the vertex it belongs to; only the attribute and constant of an attribute-value specification are shown. We indicate the relation $\leq$ by means of a pulled arrow; and the function $\ll$ will be depicted by means of a dashed arrow. Figure 3.2 shows the taxonomy from the previous example.

An *inheritance chain* in T is an expression having one of the following forms:

$y_1 \leq \ldots \leq y_n$
$y_1 \leq \ldots \leq y_n[a = c]$

where $y_i \in K$ are class frames, and $y_n[a = c] \in \Theta$ is an attribute-value specification.

The set Ω_T of inheritance chains in T is inductively defined as the smallest set such that:

- For each $y \in K$, we have $y \in \Omega_T$.
- For each $y[a = c] \in \Theta$ where $y \in K$, we have $y[a = c] \in \Omega_T$.
- For each pair $(y_1, y_2) \in \leq$ we have $y_1 \leq y_2 \in \Omega_T$.
- For each $y_1 \leq \ldots \leq y_k \in \Omega_T$ and $y_k \leq \ldots \leq y_n \in \Omega_T$, with $y_i \in K$ for each i, we have that $y_1 \leq \ldots \leq y_n \in \Omega_T$.
- For each $y_1 \leq \ldots \leq y_n \in \Omega_T$ and $y_n[a = c] \in \Omega_T$, where $y_i \in K$, for each i, we have that $y_1 \leq \ldots \leq y_n[a = c] \in \Omega_T$.

The *conclusion* $c(\omega)$ of an inheritance chain $\omega \in \Omega_T$ is defined as follows:

- For each $\omega \equiv y_1 \leq \ldots \leq y_n[a = c]$, we have that $c(\omega) = y_1[a = c]$.
- For all other ω, we have that $c(\omega)$ is not defined.

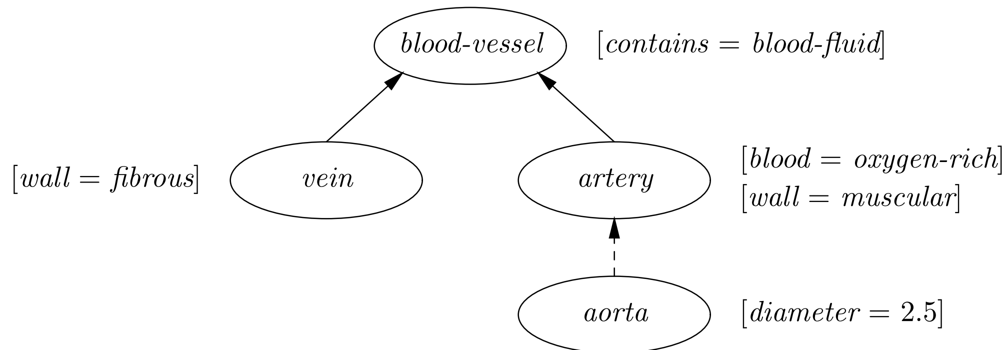

Figure 3.2: A taxonomy consisting of three classes and one instance.

The *conclusion set* $C(\Omega_T)$ of Ω_T is defined as the set of conclusions of all elements from Ω_T, that is,

$$C(\Omega_T) = \{c(\omega) \mid \omega \in \Omega_T\}$$

Of course the concept of conclusion set ignores the whole idea that we can make use of the order information represented in inheritance chains to decide about whether or not a value should be inherited for an attribute of a class. Inheritance with exceptions requires the notion of *preclusion*. This is then used to define the inheritable conclusion set.

For cancelling inheritance chains, we shall exploit the notion of an intermediary, which is introduced in the following definition.

Definition 3.2 *Let* $T = (N, \Theta, \ll, \leq)$ *be a taxonomy where* $N = (I, K, A, C)$*. Let* Ω_T *be the set of inheritance chains in* T*. A class* $y \in K$ *is called an* intermediary *to an inheritance chain* $y_1 \leq \ldots \leq y_n \in \Omega_T$*,* $y_i \in K$*, if one of the following conditions is satisfied:*

- *We have* $y = y_i$ *for some* i*.*
- *There exists a chain* $y_1 \leq \ldots \leq y_p \leq z_1 \leq \ldots \leq z_m \leq y_q \in \Omega_T$*, for some* p, q*, where* $z_j \neq y_i$*,* $z_j \in K$*, such that* $y = z_k$*, for some* k*.*

EXAMPLE 3.7

Consider the taxonomy $T = (N, \Theta, \ll, \leq)$, where $I = \varnothing$, $K = \{$*blood-vessel*, *artery*, *oxygen-poor-artery*, *pulmonary-artery*$\}$, Θ is empty, and the relation $\leq$ is defined by

pulmonary-artery $\leq$ *oxygen-poor-artery*
pulmonary-artery $\leq$ *artery*
artery $\leq$ *blood-vessel*
oxygen-poor-artery $\leq$ *artery*

The graphical representation of the taxonomy is shown in Figure 3.3. The set of inheritance chains in T contains, among other ones, the following two chains:

pulmonary-artery $\leq$ *artery* $\leq$ *blood-vessel*
pulmonary-artery $\leq$ *oxygen-poor-artery* $\leq$ *artery*

It will be evident that the class *oxygen-poor-artery* is an intermediary to both chains.

Figure 3.3 introduced in the foregoing example is useful for gaining some intuitive feeling concerning the notion of an intermediary.

We shall see that intermediaries may be applied for solving part of the problem of multiple inheritance with exceptions. We take a closer look at the figure. It seems as if the arc between the vertices *pulmonary-artery* and *artery*, an arc resulting from the transitivity property of the subclass relation, is redundant, since all attribute-value specification from the classes *artery* and *blood-vessel* can be inherited for *pulmonary-artery* via the vertex *oxygen-poor-artery*. Therefore, the removal of this arc from the taxonomy should not have any influence on the result of multiple inheritance. Whether or not this is true is, of course, dependent on our formalization of multiple inheritance. Therefore, let us investigate whether the notion of

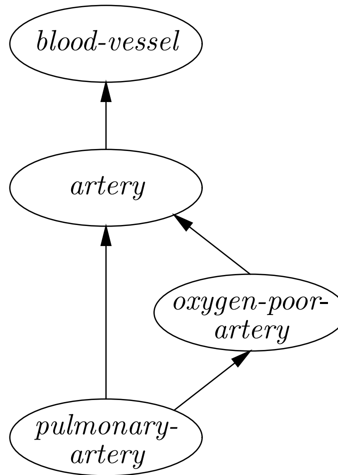

Figure 3.3: A taxonomy with an intermediary.

conclusion set defined in the foregoing renders a suitable means for dealing with exceptions. We do so by means of an example.

EXAMPLE 3.8

Consider Figure 3.3 once more. Figure 3.4 shows the taxonomy from Figure 3.3 after removal of the seemingly redundant arc. Now, suppose that the following attribute-value specifications are given:

oxygen-poor-artery[*blood* = *oxygen-poor*]
artery[*blood* = *oxygen-rich*]

Furthermore, suppose that no attribute-value specifications have been given for *pulmonary-artery*. In the taxonomy shown in Figure 3.4, the frame *pulmonary-artery* inherits only the value *oxygen-poor* for the attribute *blood*; note that this is a consequence of the way exceptions are handled in tree-shaped taxonomies. However, in Figure 3.3 the frame *pulmonary-artery* inherits both values *oxygen-poor* and *oxygen-rich* for the attribute *blood*, leading to an inconsistent conclusion set. The conclusion set of the taxonomy in Figure 3.3 therefore differs from the one obtained for the taxonomy shown in Figure 3.4, using the algorithm for single inheritance with exceptions discussed in the slides.

It turns out that a conclusion set only reveals the presence of exceptions in a taxonomy. We shall see that the notion of an intermediary is more useful in dealing with exceptions in multiple inheritance. In Figure 3.3 we have that the class *oxygen-poor-artery* lies in between the classes *pulmonary-artery* and *artery*, and is an intermediary to the inheritance chains in which the class *pulmonary-artery* and either or both the classes *artery* and *oxygen-poor-artery* occur. As we have suggested before, by means of intermediaries some of the inheritance chains may be cancelled rendering a different set of conclusions of the taxonomy. Such cancellation of inheritance chains is called *preclusion* and is defined more formally below.

Definition 3.3 *Let* $T = (N, \Theta, \ll, \leq)$ *be a taxonomy where* $N = (I, K, A, C)$*. Let* Ω_T *be the set of inheritance chains in* T*. A chain* $y_1 \leq \ldots \leq y_n[a = c_1] \in \Omega_T$ *is said to* preclude *a*

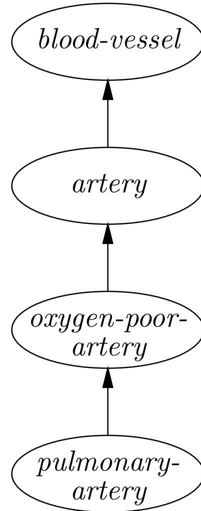

Figure 3.4: The taxonomy after removal of the redundant arc.

chain $y_1 \leq \ldots \leq y_m[a = c_2] \in \Omega_T$ *with* $m \neq n$, *and* $c_1, c_2 \in C$ *with* $c_1 \neq c_2$, *if* y_n *is an intermediary to* $y_1 \leq \ldots \leq y_m$.

EXAMPLE 3.9

Consider the set Ω_T of inheritance chains consisting of the following elements:

ω_1: *pulmonary-artery* $\leq$ *oxygen-poor-artery*
ω_2: *pulmonary-artery* $\leq$ *artery*
ω_3: *pulmonary-artery* $\leq$ *oxygen-poor-artery* $\leq$ *artery*
ω_4: *pulmonary-artery* $\leq$ *oxygen-poor-artery*[*blood* = *oxygen-poor*]
ω_5: *pulmonary-artery* $\leq$ *artery*[*blood* = *oxygen-rich*]
ω_6: *pulmonary-artery* $\leq$ *oxygen-poor-artery* $\leq$ *artery*[*blood* = *oxygen-rich*]

The reader can easily verify that the inheritance chain ω_4 precludes both chains ω_5 and ω_6 since *oxygen-poor-artery* is an intermediary to the chains ω_2 and ω_3.

The notion of preclusion is used for introducing a new type of conclusion set of a set of inheritance chains.

An inheritance chain $\omega \in \Omega_T$ is said to be *inheritable* if there exists no other inheritance chain $\omega' \in \Omega_T$ which precludes ω. The set of conclusions of all inheritable chains $\omega \in \Omega_T$ is called the *inheritable conclusion set* of Ω_T and is denoted by $H(\Omega_T)$.

Note that we have definitions of consistency and inconsistency for both the conclusion set and inheritable conclusion set (see the slides). A taxonomy T is said to be *consistent* if $H(\Omega_T)$ is consistent; otherwise T is said to be *inconsistent*.

Finally, the consequences of these definition for instances of classes can be investigated. For each instance frame $x \in I$, the set $e_H(x)$ is defined by

$$e_H(x) = \{x[a = c] \mid x[a = c] \in \Theta\} \cup \{x[a = c] \mid x \ll y, y \in K, y[a = c] \in H(\Omega_T)$$

and for all $c \neq d$, $x[a = d] \notin \Theta\}$ if $H(\Omega_T)$ is consistent; $e_H(x)$ is undefined otherwise. In words: all attribute values already defined for x are supplemented with those inherited by the class y of which x is an instance, i.e., $x \ll y$.

The *inheritable extension of* Ω_T, denoted by $E_H(\Omega_T)$, is defined by

$$E_H(\Omega_T) = \bigcup_{x \in I} e_H(x)$$

if $H(\Omega_T)$ is consistent; $E_H(\Omega_T)$ is undefined otherwise. See the slides for an example.

Note that inheritance with exceptions is an example of non-monotonic reasoning. One way to get insight into the non-monotonic characteristics of inheritance is by mapping frame taxonomies to a non-monotonic logic, such as default logic (see slides and Section 4).

Chapter 4

Lectures 6-8 – Model-based Reasoning

The term 'model-based reasoning' is used to refer to the use of models, by definitions abstractions of systems in the real world, to solve various problems. Usually, such models give insight into the workings of the system under study. Sometimes, the model includes details concerning the structure, i.e., the way it is built, as well, again using some level of abstraction Causal knowledge is often used in the development of model-based reasoning systems.

4.1 Introduction

Although there are many ways to use principles of model-based reasoning, in AI most of the research has been done in the context of so-called *model-based diagnosis*. Diagnosis is defined as establishing what is wrong with a system, e.g., an electronic device. Although the term 'diagnosis' comes from medicine, ideas around model-based diagnosis arose originally from work on fault finding in electronic circuits, where in particular Johan de Kleer has been very successful. Today, the majority of applications are still outside the medical fields; example include the aerospace industry, the automotive industry, mobile networks, robotics, etc.

There are two common types of model-based diagnosis. One of the first formal theories of diagnosis emerged from the Johan de Kleer's early research: the theory of consistency-based diagnosis as proposed by Ray Reiter (he was, in fact, Johan de Kleer's PhD thesis supervisor). *Consistency-based diagnosis* offers a logic-based framework to formally describe diagnosis of abnormal behaviour in a device or system, using a model of normal structure and functional behaviour. Basically, consistency-based diagnosis amounts to finding faulty device components that account for a discrepancy between predicted normal device behaviour and observed (abnormal) behaviour. The predicted behaviour is inferred from a formal model of normal structure and behaviour of the device.

Where consistency-based diagnosis traditionally employs a model of *normal* behaviour, *abduction* has been the principal model-based technique for describing and analysing diagnosis using a model of *abnormal* behaviour in terms of cause-effect relationships. Early work on abduction has been done by Harry Pople and David Poole.

4.2 Concistency-based diagnosis

The logical specification of knowledge concerning structure and behaviour in Reiter's theory is a triple SYS = (SD, COMPS), called a *system*, where

- SD denotes a finite set of formulae in first-order predicate logic, specifying normal structure and behaviour, called the *system description*;
- COMPS denotes a finite set of constants (nullary function symbols) in first-order logic, denoting the *components* of the system.

A *diagnostic problem* DP is defined as a pair DP = (SYS, OBS), where

- SYS is a system, and
- OBS denotes a finite set of formulae in first-order predicate logic, denoting *observations*, i.e. observed findings.

It is, in principle, possible to specify normal as well as abnormal (faulty) behaviour within a system description SD, but originally SD was designed to comprise a logical specification of normal behaviour of the modelled system only.

The essential part of a formal model of normal structure and behaviour of a system consists of logical axioms of the form

$$\forall x((\mathrm{COMP}(x) \wedge \neg \mathrm{Ab}(x)) \rightarrow o(x)_{norm})$$

where the predicate 'COMP' refers to a specific class of components, for example NAND gates, $x \in \mathrm{COMPS}$ must belong to this specific class, and $o(x)_{norm}$ denotes a finding that may be observed if the component x is normal, i.e. is nondefective. The observable findings $o(x)_{norm}$ need not be unique. Axioms of the above form are provided for each class of component in COMPS. For example, if COMPS $= \{A_1, A_2, O_1, O_2\}$, where A_1 and A_2 are AND gates, and O_1, O_2 are two OR gates, then one would get the following two formulae as part of SD:

$$\forall x((\mathrm{AND}(x) \wedge \neg \mathrm{Ab}(x)) \rightarrow (out(x) = (in1(x) \text{ and } in2(x)))))$$
$$\forall x((\mathrm{OR}(x) \wedge \neg \mathrm{Ab}(x)) \rightarrow (out(x) = (in1(x) \text{ or } in2(x)))))$$

together with the formulae (just facts in SD) $\{\mathrm{AND}(A_1), \mathrm{AND}(A_2), \mathrm{OR}(O_1), \mathrm{OR}(O_2)\}$. One can then derive, using ordinary logical reasoning with reasoning with equality (=) and Boolean operators (and, or), that:

$$\mathrm{SD} \cup \neg \mathrm{Ab}(O_1) \cup \{in1(O_1) = 1, in2(O_1) = 0\} \vdash out(O_1) = 1$$

The equality and Boolean extension to logical reasoning is needed to derive from $out(O_1) = (in1(O_1) \text{ or } in2(O_1))$ and $\{in1(O_1) = 1, in2(O_1) = 0\}$ that $out(O_1) = 1$. It is a technical issue, not unimportant for those wishing to implement model-based reasoning. Adopting the definition from De Kleer a diagnosis in the theory of consistency-based diagnosis can be defined as follows.

Definition 4.1 *(consistency-based diagnosis) Let* SYS = (SD, COMPS) *be a system and* DP = (SYS, OBS) *be a diagnostic problem with set of observations* OBS. *Let*

$$H_P = \{Ab(c) \mid c \in \mathrm{COMPS}\}$$

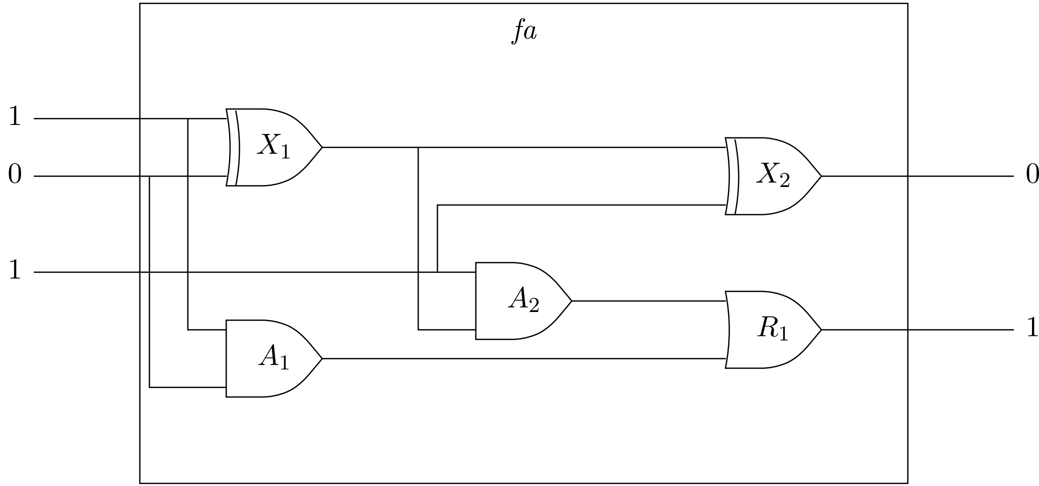

Figure 4.1: Full adder.

be the set of all positive 'Ab' literals, and

$$H_N = \{\neg Ab(c) \mid c \in \text{COMPS}\}$$

be the set of all negative 'Ab' literals. Furthermore, let $H \subseteq H_P \cup H_N$ *be a set, called a* hypothesis*, such that*

$$H = \{Ab(c) \mid c \in D\} \cup \{\neg Ab(c) \mid \text{COMPS} - D\}$$

for some $D \subseteq \text{COMPS}$*. Then, the* D *is a* (consistency-based) diagnosis *of* DP *if the following condition, called the* consistency condition*, holds:*

$$\text{SD} \cup H \cup \text{OBS} \nvDash \bot \tag{4.1}$$

i.e. $\text{SD} \cup H \cup \text{OBS}$ *is consistent.*

In some definitions H, instead of D, is called the diagnosis. For this lecture we use the simpler definition where D is taken to be the diagnosis.

EXAMPLE 4.1

Consider the logical circuit depicted in Figure 4.1, which represents a full adder, i.e. a circuit that can be used for the addition of two bits with carry-in and carry-out bits. The components X_1 and X_2 represent exclusive-OR gates, A_1 and A_2 represent AND gates, and R_1 represents an OR gate.

The system description consists of the following axioms:

$$\begin{aligned}
\forall x(\text{ANDG}(x) \wedge \neg Ab(x) &\rightarrow out(x) = and(in1(x), in2(x))) \\
\forall x(\text{XORG}(x) \wedge \neg Ab(x) &\rightarrow out(x) = xor(in1(x), in2(x))) \\
\forall x(\text{ORG}(x) \wedge \neg Ab(x) &\rightarrow out(x) = or(in1(x), in2(x)))
\end{aligned}$$

which describe the (normal) behaviour of each individual component (gate), and

$$\begin{aligned}
\mathit{out}(X_1) &= \mathit{in2}(A_2)\\
\mathit{out}(X_1) &= \mathit{in1}(X_2)\\
\mathit{out}(A_2) &= \mathit{in1}(R_1)\\
\mathit{in1}(A_2) &= \mathit{in2}(X_2)\\
\mathit{in1}(X_1) &= \mathit{in1}(A_1)\\
\mathit{in2}(X_1) &= \mathit{in2}(A_1)\\
\mathit{out}(A_1) &= \mathit{in2}(R_1)
\end{aligned}$$

which gives information about the connections between the components, i.e. information about the normal structure, including some electrical relationships. Finally, the various gates are defined:

$$\begin{array}{l}
\mathrm{ANDG}(A_1)\\
\mathrm{ANDG}(A_2)\\
\mathrm{XORG}(X_1)\\
\mathrm{XORG}(X_2)\\
\mathrm{ORG}(R_1)
\end{array}$$

Appropriate axioms for a Boolean algebra are also assumed to be available.

Now, let us assume that

$$\mathrm{OBS} = \{\mathit{in1}(X_1) = 1, \mathit{in2}(X_1) = 0, \mathit{in1}(A_2) = 1, \mathit{out}(X_2) = 0, \mathit{out}(R_1) = 0\}$$

Note that $\mathit{out}(R_1) = 1$ is predicted using the model of normal structure and behaviour in Figure 4.1, which is in contrast with the observed output $\mathit{out}(R_1) = 0$. Assuming that $H = \{\neg Ab(c) \mid c \in \mathrm{COMPS}\}$, it follows that

$$\mathrm{SD} \cup H \cup \mathrm{OBS}$$

is inconsistent. This confirms that some of the output signals observed differ from those expected under the assumption that the circuit is functioning normally. Using Formula (4.1), a possible hypothesis is, for instance,

$$H' = \{Ab(X_1), \neg Ab(X_2), \neg Ab(A_1), \neg Ab(A_2), \neg Ab(R_1)\}$$

since

$$\mathrm{SD} \cup H' \cup \mathrm{OBS}$$

is consistent. In terms of our definition, the corresponding diagnosis would be $D' = \{X_1\}$. Note that, given the diagnosis D', no output is predicted for the circuit; the assumption $Ab(X_1)$ completely blocks transforming input into output by the modelled circuit, because

$$\mathrm{SD} \cup H' \cup \mathrm{OBS} \backslash \{\mathit{out}(X_2) = 0\} \nvDash \mathit{out}(X_2) = 0$$

In a sense, this is too much, because there was no discrepancy between the predicted and

observed output of gate X_2. Nevertheless, D' is a diagnosis according to Definition 4.1.

Reiter has also given an analysis of consistency-based diagnosis in terms of default logic (see below and slides). A system description SD and a set of observations OBS are supplemented with default rules of the form

$$\frac{: \neg Ab(c)}{\neg Ab(c)}$$

for each component c, yielding a default theory. A default rule as above expresses that $\neg Ab(c)$ may be assumed for component c, if assuming $\neg Ab(c)$ does not give rise to inconsistency. Hence, in computing an extension of the resulting default theory, these default rules will only be applied under the condition that they do not violate consistency, which is precisely the effect of the consistency condition (4.1).

4.3 Abductive diagnosis

Abductive diagnosis uses cause-effect relationships to explain observed effects in terms of assumed causes. A diagnosis is, thus, interpreted as an *explanation* of those observed effects. In the scientific literature, there are two different approaches to formalise abductive diagnosis: (1) a logical approach, described in the next section, and (2) a set-theoretical approach, which is described in Section 4.3.2.

4.3.1 Logical abduction

As in consistency-based diagnosis, abductive diagnosis starts by considering of how to represent knowledge about systems, in this case in terms of cause-effect relationships. Much of the theory comes from Pietro Torasso and Luca Console from the University of Turin (where you find the Fiat factory, where model-based diagnosis has been one of the research topics).

How to model cause-effect relationships in logic is a non-trivial question (the question could even be whether it is possible to model causality in logic). The first step in answering this question is to consider the axioms that must be fulfilled (see slide about causality and implication). Most people would agree that at least transitivity, if a causes b and b causes c, then a causes c, should be fulfilled, but even in this case, transitivity may not always hold (consider, e.g., weak causality mentioned below).

In the following, it shall be assumed that axioms are of the following two forms:

$$d_1 \wedge \cdots \wedge d_n \rightarrow f \quad (4.2)$$

$$d_1 \wedge \cdots \wedge d_n \rightarrow d \quad (4.3)$$

where d, d_i, $i = 1, \ldots, n$, represent defects, disorders, etc.

Console and Torasso also provide a mechanism in their logical formalisation to weaken the causality relation. To this end, literals α are introduced into the premises of the axioms of the form (4.2) and (4.3), which can be used to block the deduction of an observable finding f or defect d if the defects d_i, $i = 1, \ldots, n$, hold true, by assuming the literal α to be false. The weakened axioms have the following form:

$$d_1 \wedge \cdots \wedge d_n \wedge \alpha_f \rightarrow f \quad (4.4)$$

$$d_1 \wedge \cdots \wedge d_n \wedge \alpha_d \rightarrow d \quad (4.5)$$

The literals α are called *incompleteness-assumption literals*, abbreviated to *assumption literals*. Axioms of the form (4.2) – (4.5) are now taken as the (abnormality) axioms.

In the following, let $\Sigma = (\Delta, \Phi, \mathcal{R})$ stand for a *causal specification* in the theory of diagnosis by Console and Torasso, where:

- Δ denotes a set of possible defect and assumption literals;
- Φ denotes a set of possible (positive and negative) observable finding literals;
- $\mathcal{R}$ ('Causal Model') stands for a set of logical (abnormality) axioms of the form (4.2) – (4.5).

Subsets of the set Δ will be called *hypotheses*.

A causal specification can then be employed for the prediction of observable findings. Let $\Sigma = (\Delta, \Phi, \mathcal{R})$ be a causal specification. Then, a hypothesis $V \subseteq \Delta$ is called a *prediction* for a set of observable findings $E \subseteq \Phi$ if

(1) $\mathcal{R} \cup V \vDash E$, and

(2) $\mathcal{R} \cup V$ is consistent.

See the slides for examples of how this idea is used.

An *abductive diagnostic problem* $\mathcal{P}$ is now defined as a pair $\mathcal{P} = (\Sigma, F)$, where $F \subseteq \Phi$ is called a *set of observed findings*.

Formally, a solution to an abductive diagnostic problem $\mathcal{P}$ can be defined as follows.

Definition 4.2 *(abductive diagnosis) Let $\mathcal{P} = (\Sigma, F)$ be an abductive diagnostic problem, where $\Sigma = (\Delta, \Phi, \mathcal{R})$ is a causal specification with $\mathcal{R}$ a set of abnormality axioms of the form (4.2) – (4.5), and $E \subseteq \Phi$ a set of observed findings. A hypothesis $D \subseteq \Delta$ is called an* abductive diagnosis *of $\mathcal{P}$ if:*

(1) $\mathcal{R} \cup D \vDash F$ (covering condition)*;*

(2) $\mathcal{R} \cup D \cup C \nvDash \bot$ (consistency condition)

where C, the set of constraints*, is defined by:*

$$C = \{\neg f \in \Phi \mid f \in \Phi, f \notin F,\ f \text{ is a positive literal}\} \tag{4.6}$$

Note that the sets F and C are disjoint, and that if $f \in F$ then $\neg f \notin C$. The set C stands for findings assumed to be false, because they have not been observed (and are therefore assumed to be absent). However, other definitions are possible.

EXAMPLE 4.2

Consider the causal specification $\Sigma = (\Delta, \Phi, \mathcal{R})$, with

$$\Delta = \{fever, influenza, sport, \alpha_1, \alpha_2\}$$

and

$$\Phi = \{chills, thirst, myalgia, \neg chills, \neg thirst, \neg myalgia\}$$

'Myalgia' means painful muscles. The following set of logical formulae R, representing medical knowledge concerning influenza and sport, both 'disorders' with frequent occurrence, is given:

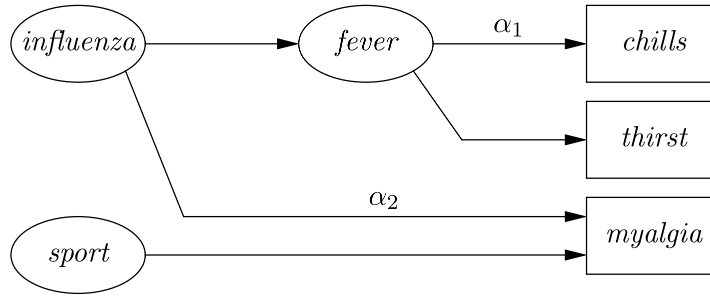

Figure 4.2: A knowledge base with causal relations.

$$
\begin{array}{l}
\mathit{fever} \wedge \alpha_1 \rightarrow \mathit{chills} \\
\mathit{influenza} \rightarrow \mathit{fever} \\
\mathit{fever} \rightarrow \mathit{thirst} \\
\mathit{influenza} \wedge \alpha_2 \rightarrow \mathit{myalgia} \\
\mathit{sport} \rightarrow \mathit{myalgia}
\end{array}
$$

For example, $\mathit{influenza} \wedge \alpha_2 \rightarrow \mathit{myalgia}$ means that influenza *may cause* myalgia; $\mathit{influenza} \rightarrow \mathit{fever}$ means that influenza *always causes* fever. For illustrative purposes, a causal knowledge base as given above is often depicted as a labelled, directed graph G, which is called a *causal net*, as shown in Figure 4.2. Suppose that the abductive diagnostic problem $\mathcal{P} = (\Sigma, F)$ must be solved, where the set of observed findings $F = \{\mathit{thirst}, \mathit{myalgia}\}$. Then, $C = \{\neg \mathit{chills}\}$. There are several solutions to this abductive diagnostic problem (for which the consistency and covering conditions are fulfilled):

$$
\begin{array}{l}
D_1 = \{\mathit{influenza}, \alpha_2\} \\
D_2 = \{\mathit{influenza}, \mathit{sport}\} \\
D_3 = \{\mathit{fever}, \mathit{sport}\} \\
D_4 = \{\mathit{fever}, \mathit{influenza}, \alpha_2\} \\
D_5 = \{\mathit{influenza}, \alpha_2, \mathit{sport}\} \\
D_6 = \{\mathit{fever}, \mathit{influenza}, \mathit{sport}\} \\
D_7 = \{\mathit{fever}, \mathit{influenza}, \alpha_2, \mathit{sport}\}
\end{array}
$$

Finally, note that, for example, the hypothesis $D = \{\alpha_1, \alpha_2, \mathit{fever}, \mathit{influenza}\}$ is incompatible with the consistency condition.

Several researchers have noted a close correspondence between abduction and the predicate completion of a logical theory, as originally proposed in connection with negation as finite failure in logic programming, i.e., the `not` predicate. Consider the following example.

EXAMPLE 4.3

Suppose that sport and influenza are two 'disorders'; this may be expressed in predicate logic as follows:

$$
\begin{array}{l}
\mathit{Disorder}(\mathit{sport}) \\
\mathit{Disorder}(\mathit{influenza})
\end{array}
$$

The following logical implication is equivalent to the conjunction of the two literals above:

$$\forall x((x = \mathit{sport} \vee x = \mathit{influenza}) \rightarrow \mathit{Disorder}(x))$$

assuming the presence of the logical axioms for equality, and also assuming that constants with different names are not equal. Suppose that sport and influenza are the *only* possible disorders. This can be expressed by adding the following logical implication:

$$\forall x(\mathit{Disorder}(x) \rightarrow (x = \mathit{sport} \vee x = \mathit{influenza})) \tag{4.7}$$

to the implication above. For example, adding $\mathit{Disorder}(\mathit{asthma})$ to logical implication (4.7) yields an inconsistency, because *asthma* is neither equal to *sport* nor equal to *influenza*: the conclusion

$$\mathit{asthma} = \mathit{sport} \vee \mathit{asthma} = \mathit{influenza}$$

cannot be satisfied. Now, suppose that the literal $\mathit{Disorder}(\mathit{asthma})$ is removed, but that '*asthma*' remains a valid constant symbol. Then, $\neg\mathit{Disorder}(\mathit{asthma})$ is a logical consequence of formula (4.7); this formula 'completes' the logical theory by stating that disorders not explicitly mentioned are assumed to be false. Formula (4.7) is called a *completion formula*.

The characterisation of abduction as deduction in a completed logical theory is natural, because computation of the predicate completion of a logical theory amounts to adding the only-if parts of the formulae to the theory, i.e. it 'reverses the arrow' which is exactly what happens when abduction is applied to derive conclusions. After all, abductive reasoning is reasoning in a direction reverse to logical implication. In an intuitive sense, predicate completion expresses that the only possible causes (defects) for observed findings are those appearing in the abnormality axioms; assumption literals are taken as implicit causes. Where the characterisation of abduction by means of the covering and consistency conditions may be viewed as a meta-level description of abductive diagnosis, the predicate completion can be taken as the object-level characterisation, i.e. in terms of the original axioms in $\mathcal{R}$. However, in contrast to the predicate completion in logic programming, predicate completion should only pertain to literals appearing as a consequence of the logical axioms in $\mathcal{R}$, i.e. finding literals and defect literals that can be derived from other defects and assumption literals. This set of defects and observable findings is called the set of *non-abducible* literals, denoted by A; the set $\Delta \backslash A$ is then called the set of *abducible* literals.

Let us denote the axiom set $\mathcal{R}$ by

$$\begin{aligned}\mathcal{R} = \{&\varphi_{1,1} \rightarrow a_1, \ldots, \varphi_{1,n_1} \rightarrow a_1,\\ &\vdots\\ &\varphi_{m,1} \rightarrow a_m, \ldots, \varphi_{m,n_m} \rightarrow a_m\}\end{aligned}$$

where $A = \{a_i \mid 1 \leq i \leq m\}$ is the set of non-abducible (finding or defect) literals and each $\varphi_{i,j}$ denotes a conjunction of defect literals, possibly including an assumption literal. The predicate completion of $\mathcal{R}$ with respect to the non-abducible literals A, denoted by $\mathrm{COMP}[\mathcal{R}; A]$ is defined as follows:

$$\begin{aligned}\text{COMP}[\mathcal{R}; A] = \mathcal{R} \cup \{a_1 &\rightarrow \varphi_{1,1} \vee \cdots \vee \varphi_{1,n_1},\\ &\vdots\\ a_m &\rightarrow \varphi_{m,1} \vee \cdots \vee \varphi_{m,n_m}\}\end{aligned}$$

The predicate completion of $\mathcal{R}$ makes explicit the fact that the only causes of non-abducible literals (findings and possibly also defects) are the defects and assumption literals given as a disjunct in the consequent. For example,

$$f_{ab} \rightarrow d_1 \vee \cdots \vee d_n$$

indicates that only the defects from the set $\{d_1, \ldots, d_n\}$ can be used to explain the observed finding f_{ab}.

Predicate completion of abnormality axioms with respect to a set of non-abducible literals can now be used to characterise diagnosis. Let ψ and ψ' be two logical formulae. It is said that ψ is *more specific than* ψ' iff $\psi \vDash \psi'$. Using the predicate completion of a set of abnormality axioms $\mathcal{R}$, we now have the following definition.

Definition 4.3 *(solution formula) Let $\mathcal{P} = (\Sigma, F)$ be an abductive diagnostic problem and let* COMP$[\mathcal{R}; A]$ *be the predicate completion of $\mathcal{R}$ with respect to A, the set of non-abducible literals in $\mathcal{P}$. A* solution formula *S for $\mathcal{P}$ is defined as the most specific formula consisting only of abducible literals, such that*

$$\text{COMP}[\mathcal{R}; A] \cup F \cup C \vDash S$$

where C is defined as in Equation (4.6).

Hence, abductive solutions become deductive solutions (cf. Section 1.3.1). A solution formula is obtained by applying the set of equivalences in COMP$[\mathcal{R}; A]$ to a set of observed findings F, augmented with those findings not observed, C, yielding a logical formula that includes all possible solutions according Equation (4.2), given the equivalences in COMP$[\mathcal{R}; A]$. The following theorem reveals an important relationship between the meta-level characterisation of abductive diagnosis, as presented in Definition (4.2), and the object-level characterisation of diagnosis in Definition 4.3.

THEOREM 4.1 *Let $\mathcal{P} = (\Sigma, F)$ be an abductive diagnostic problem, where $\Sigma = (\Delta, \Phi, \mathcal{R})$ is a causal specification. Let C be defined as in Definition 4.2, and let S be a solution formula for $\mathcal{P}$. Let $H \subseteq \Delta$ be a set of abducible literals, and let I be an interpretation of $\mathcal{P}$, such that for each abducible literal $a \in \Delta$: $\vDash_I a$ iff $a \in H$. Then, H is a solution to $\mathcal{P}$ iff $\vDash_I S$.*

Proof: ($\Rightarrow$): The set of defect and assumption literals H is a solution to $\mathcal{P}$, hence, for each $f \in F$: $\mathcal{R} \cup H \vDash f$, and for each $f' \in C$: $\mathcal{R} \cup H \nvDash \neg f'$. The solution formula S is the result of rewriting observed findings in E and non-observed findings in C using the equivalences in COMP$[\mathcal{R}; A]$ to a formula merely consisting of abducibles. Assume that S is in conjunctive normal form. Conjuncts in S are equivalent to observed findings $f \in F$, that are logically entailed by $\mathcal{R} \cup H$, or to non-observed findings $\neg f \in C$ that are consistent with $\mathcal{R} \cup H$. Hence, an interpretation I for which $\vDash_I H$, that falsifies each abducible in $\Delta \backslash H$, satisfying every $f \in F$ and each $\neg f \in C$ that has been rewritten, must satisfy this collection of conjuncts, i.e. S.
($\Leftarrow$): If S is in conjunctive normal form, S must be the result of rewriting observed findings $f \in$

F and non-observed findings in C to (negative or positive) abducibles, using the equivalences in COMP[$\mathcal{R}$; A]. Since an interpretation I that satisfies H and S must also satisfy each finding $f \in F$ and those $\neg f \in C$ that have been rewritten to S, it follows that I can be chosen such that $\models_I C$, i.e. H must be a solution to $\mathcal{P}$. $\diamondsuit$

This theorem reveals an important property of the abductive theory of diagnosis. Sometimes, a solution to an abductive diagnostic problem is capable of satisfying a solution formula in the technical, logical sense.

EXAMPLE 4.4

Reconsider the set of logical axioms given in Example 4.2. The predicate completion of $\mathcal{R}$ is equal to

$$\begin{aligned}
&\mathrm{COMP}[\mathcal{R}; \{\mathit{chills}, \mathit{thirst}, \mathit{myalgia}, \mathit{fever}\}] \\
&= \mathcal{R} \cup \{\mathit{chills} \rightarrow \mathit{fever} \wedge \alpha_1, \\
&\qquad \mathit{fever} \rightarrow \mathit{influenza}, \\
&\qquad \mathit{thirst} \rightarrow \mathit{fever}, \\
&\qquad \mathit{myalgia} \rightarrow (\mathit{influenza} \wedge \alpha_2) \vee \mathit{sport}\} \\
&= \{\mathit{chills} \leftrightarrow \mathit{fever} \wedge \alpha_1, \\
&\quad \mathit{fever} \leftrightarrow \mathit{influenza}, \\
&\quad \mathit{thirst} \leftrightarrow \mathit{fever}, \\
&\quad \mathit{myalgia} \leftrightarrow (\mathit{influenza} \wedge \alpha_2) \vee \mathit{sport}\}
\end{aligned}$$

Note that

$$\mathrm{COMP}[\mathcal{R}; \{\mathit{chills}, \mathit{thirst}, \mathit{myalgia}, \mathit{fever}\}] \cup F \cup C \models (\mathit{influenza} \wedge \alpha_2) \vee (\mathit{influenza} \wedge \mathit{sport})$$

given that $F = \{\mathit{thirst}, \mathit{myalgia}\}$ and $C = \{\neg \mathit{chills}\}$. Although

$$\mathrm{COMP}[\mathcal{R}; \{\mathit{chills}, \mathit{thirst}, \mathit{myalgia}, \mathit{fever}\}] \cup F \cup C \models \neg(\mathit{fever} \wedge \alpha_1)$$

the formula $\neg(\mathit{fever} \wedge \alpha_1)$, which is a logical consequence of $\neg\mathit{chills}$ and $\mathit{chills} \leftrightarrow (\mathit{fever} \wedge \alpha_1)$, is not part of the solution formula $S \equiv (\mathit{influenza} \wedge \alpha_2) \vee (\mathit{influenza} \wedge \mathit{sport})$, because the literal *fever* is non-abducible. It holds, in accordance with Theorem 4.1, that

$$\models_I H_i \;\Rightarrow\; \models_I (\mathit{influenza} \wedge \alpha_2) \vee (\mathit{influenza} \wedge \mathit{sport})$$

for $i = 1, 2, 5$, where H_i is a solution given in Example 4.2 consisting only of abducible literals, for suitable interpretations I. Here, it even holds that $H_i \models S$, because S does not contain any negative defects or assumption literals entailed by non-observed findings in C.

David Poole's, a Canadian researcher located in Vancouver, Prolog-based system AILog offers a general-purpose environment to experiment with hypothetical reasoning, and is also suitable as a basis for abductive and consistency-based diagnostic systems. See Section 4.5 for details.

4.3.2 Set-covering theory of diagnosis

Instead of choosing logic as the language for abductive diagnosis, as discussed above, others have adopted set theory as their formal language. This approach to the formalisation of diagnosis is referred to as the *set-covering theory of diagnosis*, or *parsimonious covering theory*. The treatment of the set-covering theory of diagnosis in the literature deals only with the modelling of restricted forms of abnormal behaviour of a system.

The specification of the knowledge involved in diagnostic problem solving consists of the enumeration of all findings that may be present (and observed) given the presence of each individual defect distinguished in the domain; the association between each defect and its associated set of observable findings is interpreted as an uncertain *causal relation* between the defect and each of the findings in the set of observable findings. Instead of the terms 'defect' and 'finding' the terms 'disorder' and 'manifestation' are employed in descriptions of the set-covering theory of diagnosis. In the following, we have chosen to uniformly employ the terms 'defect' and 'finding' instead. The basic idea of the theory with respect to diagnosis is that each finding in the set of observed findings in a given diagnostic situation must be causally related to at least one present defect; the collected set of present defects thus obtained can be taken as a diagnosis. As with the theory of diagnosis by Console and Torasso, this reasoning method is usually viewed as being abductive in nature, because the reasoning goes from findings to defects, using causal knowledge from defects to findings.

More formally, the triple $\mathcal{N} = (\Delta, \Phi, C)$ is called a *causal net* in the set-covering theory of diagnosis, where

- Δ is a set of possible *defects*,
- Φ is a set of elements called *observable findings*, and
- C is a binary relation

  $$C \subseteq \Delta \times \Phi$$

  called the *causation relation*.

A *diagnostic problem* in the set-covering theory of diagnosis is then defined as a pair $\mathcal{D} = (\mathcal{N}, E)$, where $E \subseteq \Phi$ is a *set of observed findings*. It is assumed that all defects $d \in \Delta$ are potentially present in a diagnostic problem, and all findings $f \in \Phi$ will be observed when present. In addition, all defects $d \in \Delta$ have a causally related observable findings $f \in \Phi$, and vice versa, i.e. $\forall d \in \Delta\, \exists f \in \Phi : (d, f) \in C$, and $\forall f \in \Phi\, \exists d \in \Delta : (d, f) \in C$. No explicit distinction is made in the theory between positive (present), negative (absent) and unknown defects, and positive (present), negative (absent) and unknown findings. The causation relation is often depicted by means of a labelled, directed acyclic graph, which, as $\mathcal{N}$, is called a *causal net*.

Let $\wp(X)$ denote the power set of the set X. It is convenient to write the binary causation relation C as two functions. Since in the next section, such functions are intensively employed, we adopt a notation that slightly generalises the notation originally proposed.[1] The first

[1]In the original definition of set-covering diagnosis, function e for singleton sets is called *effects*, and is defined for elements only. They also define an associated function *Effects*, which is defined on sets of defects, in terms of the *effects* function. This function is identical to our function e. Hence, the *effects* function is superfluous. Similarly, the functions corresponding to the function c are called *causes* and *Causes*.

function

$$e : \wp(\Delta) \rightarrow \wp(\Phi)$$

called the *effects function*, is defined as follows; for each $D \subseteq \Delta$:

$$e(D) = \bigcup_{d \in D} e(\{d\}) \tag{4.8}$$

where

$$e(\{d\}) = \{f \mid (d, f) \in C\}$$

and the second function

$$c : \wp(\Phi) \rightarrow \wp(\Delta)$$

called the *causes function*, is defined as follows; for each $E \subseteq \Phi$:

$$c(E) = \bigcup_{f \in E} c(\{f\})$$

where

$$c(\{f\}) = \{d \mid (d, f) \in C\}$$

Hence, knowledge concerning combinations of findings and defects is taken as being composed of knowledge concerning individual defects or findings, which is not acceptable in general. This is a strong assumption, because it assumes that no interaction occurs between defects.

A causal net can now be redefined, in terms of the effects function e above, as a triple $\mathcal{N} = (\Delta, \Phi, e)$.

Given a set of observed findings, diagnostic problem solving amounts to determining sets of defects – technically the term *cover* is employed – that account for *all* observed findings. Formally, a diagnosis is defined as follows.

Definition 4.4 *(set-covering diagnosis) Let $\mathcal{D} = (\mathcal{N}, E)$ be a diagnostic problem, where $\mathcal{N} = (\Delta, \Phi, e)$ is a causal net and E denotes a set of observed findings. Then, a* (set-covering) diagnosis *of $\mathcal{D}$ is a set of defects $D \subseteq \Delta$, such that:*

$$e(D) \supseteq E \tag{4.9}$$

In the set-covering theory of diagnosis the technical term ‘cover’ is employed instead of ‘diagnosis’; ‘diagnosis’ will be the name adopted in the lecture notes. Due to the similarity of condition (4.9) with the covering condition in the abductive theory of diagnosis, this condition is called the *covering condition* in the set-covering theory of diagnosis. Actually, set-covering diagnosis can be mapped to abductive diagnosis in a straightforward way, thus revealing that set-covering diagnosis is more restrictive than abductive diagnosis. Just by mapping each function value

$$e(\{d\}) = \{f_1, \ldots, f_n\}$$

to a collection of logical implications, taken as abnormality axioms $\mathcal{R}$ of a causal specification $\Sigma = (\Delta, \Phi, \mathcal{R})$, of the following form:

$$\begin{aligned} d \wedge \alpha_{f_1} &\rightarrow f_1 \\ d \wedge \alpha_{f_2} &\rightarrow f_2 \\ &\vdots \\ d \wedge \alpha_{f_n} &\rightarrow f_n \end{aligned}$$

abductive diagnosis for such restricted causal specifications and set-covering diagnosis coincide.

Since it is assumed that $e(\Delta) = \Phi$ is satisfied, i.e. any finding $f \in \Phi$ is a possible causal effect of at least one defect $d \in \Delta$, there exists a diagnosis for any set of observed findings E, because

$$e(\Delta) \supseteq E$$

always holds (explanation existence theorem).

A set of defects D is said to be an *explanation* of a diagnostic problem $\mathcal{D} = (\mathcal{N}, E)$, with E a set of observed findings, if D is a diagnosis of E and D satisfies some additional criteria. Various criteria, in particular so-called *criteria of parsimony*, sometimes called *Occam's razor* after the medieval, English Franciscan friar William of Ockam who formulated this in his work, are in use. The basic idea is that among the various diagnoses of a set of observable findings, those that satisfy certain criteria of parsimony are more likely than others. Let $\mathcal{D} = (\mathcal{N}, E)$ be a diagnostic problem, then some of the criteria are:

- *Minimal cardinality*: a diagnosis D of E is an explanation of $\mathcal{D}$ iff it contains the minimum number of elements among all diagnoses of E;
- *Irredundancy*: a diagnosis D of E is an explanation of $\mathcal{D}$ iff no proper subset of D is a diagnosis of E;
- *Relevance*: a diagnosis D of E is an explanation of $\mathcal{D}$ iff $D \subseteq c(E)$;
- *Most probable diagnosis*: a diagnosis D of E is an explanation of $\mathcal{D}$ iff $P(D|E) \geq P(D'|E)$ for any diagnosis D' of E.

In addition, some researchers define the concept of minimal-cost diagnosis. A diagnosis D of a set of observed findings E is called a *minimal-cost explanation* of $\mathcal{D}$ iff

$$\sum_{d \in D} cost(d) \leq \sum_{d \in D'} cost(d)$$

for each diagnosis D' of E, where *cost* is a function associating real values with defects $d \in \Delta$. The cost of a diagnosis may be anything, varying from financial costs to some subjective feeling of importance expressed by numbers. Interestingly, interpreting a cost function as the negative logarithm of probabilities, a minimal-cost diagnosis is identical to a most probable diagnosis in terms of probability theory.

Although not every diagnosis is an explanation, any diagnosis may be seen as a solution to a diagnostic problem, where diagnoses which represent explanations conform to more strict

conditions than diagnoses that do not. The term 'explanation' refers to the fact that a diagnosis in the set-covering theory of diagnosis can be stated, and thus be explained, in terms of cause-effect relationships. A better choice, in our opinion, would have been the adoption of the term 'explanation' for what is now called 'cover' in the theory, and to refer to what are now called 'explanations' by the name of 'parsimonious explanations'. To avoid confusion, the term 'explanation' will not be used in the sequel. Instead, we shall speak of a 'minimal-cardinality diagnosis', an 'irredundant diagnosis', a 'minimal-cost diagnosis' and so on.

For minimal cardinality, a diagnosis which consists of the smallest number of defects among all diagnoses is considered the most plausible diagnosis. Minimal cardinality is a suitable parsimony criterion in domains in which large combinations of defects are unlikely to occur. For example, in medicine, it is generally more likely that a patient has a single disorder than more than one disorder. Irredundancy expresses that it is not possible to leave out a defect from an explanation without losing the capability of explaining the complete set of observed findings, i.e.

$$e(D) \not\supseteq E$$

for each $D \subset D'$, where D' is an irredundant diagnosis. The relevance criterion states that every defect in an explanation has at least one observable finding in common with the set of observed findings. This seems an obvious criterion, but note that the notion of uncertain causal relation employed in the set-covering theory of diagnosis does not preclude situations in which a defect is present, although none of its causally related observable findings have been observed. These three definitions of the notion of explanation are based on general set-theoretical considerations. In contrast, the most probable diagnosis embodies some knowledge of the domain, in particular with respect to the strengths of the causal relationships. We shall not deal with such probabilistic extensions of the set-covering theory of diagnosis any further.

EXAMPLE 4.5

Consider the causal net $\mathcal{N} = (\Delta, \Phi, C)$, where the effects function e is defined by the causation relation C, i.e.

$$e(D) = \bigcup_{d \in D} e(\{d\})$$

where

$$e(\{d\}) = \begin{cases} \{cough, fever, sneezing\} & \text{if } d = influenza \\ \{cough, sneezing\} & \text{if } d = common\ cold \\ \{fever, dyspnoea\} & \text{if } d = pneumonia \end{cases}$$

It states, for example, that a patient with influenza will be coughing, sneezing and have a fever; a patient with a common cold will show the same findings, except fever, and a patient with pneumonia will have a fever and dyspnoea (shortness of breath). The associated graph representation G_C of C is shown in Figure 4.3. It holds, among others, that

$$e(\{influenza, common\ cold\}) = \{cough, fever, sneezing\}$$

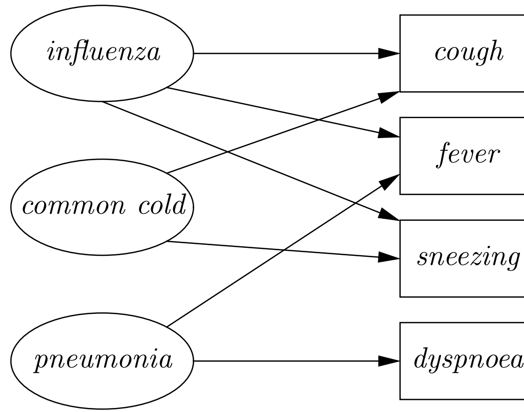

Figure 4.3: Causal net.

Based on the causal net C, the following causes function c is obtained:

$$c(E) = \bigcup_{o \in E} c(\{o\})$$

with

$$c(\{f\}) = \begin{cases} \{\mathit{influenza}, \mathit{common\ cold}\} & \text{if } f = \mathit{cough} \\ \{\mathit{influenza}, \mathit{pneumonia}\} & \text{if } f = \mathit{fever} \\ \{\mathit{influenza}, \mathit{common\ cold}\} & \text{if } f = \mathit{sneezing} \\ \{\mathit{pneumonia}\} & \text{if } f = \mathit{dyspnoea} \end{cases}$$

Suppose $\mathcal{D} = (\mathcal{N}, E)$ is a diagnostic problem, with $E = \{\mathit{cough}, \mathit{fever}\}$ a set of observed findings, then a diagnosis of $\mathcal{D}$ is

$$D_1 = \{\mathit{influenza}\}$$

but

$$D_2 = \{\mathit{influenza}, \mathit{common\ cold}\}$$
$$D_3 = \{\mathit{common\ cold}, \mathit{pneumonia}\}$$

and $D_4 = \{\mathit{influenza}, \mathit{common\ cold}, \mathit{pneumonia}\}$ are also diagnoses for E. All of these diagnoses are relevant diagnoses, because

$$c(\{\mathit{cough}, \mathit{fever}\}) \supseteq D_i$$

where $i = 1, \ldots, 4$. Irredundant diagnoses of E are D_1 and D_3. There is only one minimal cardinality diagnosis, viz. $D_1 = \{\mathit{influenza}\}$. Now suppose that $E = \{\mathit{cough}\}$, then for example $D = \{\mathit{influenza}, \mathit{pneumonia}\}$ would not have been a relevant diagnosis, because

$$c(\{\mathit{cough}\}) = \{\mathit{influenza}, \mathit{common\ cold}\} \not\supseteq D$$

Other, more domain-specific, definitions of the notion of explanation have only been developed recently. Such domain-specific knowledge can be effective in reducing the size of the set of diagnoses generated by a diagnostic system. For example, Tuhrim et al. demonstrated that the use of knowledge concerning the three-dimensional structure of the brain by means of a binary adjacency relation in a neurological diagnostic knowledge system, based on the set-covering theory of diagnosis, could increase the diagnostic accuracy of the system considerably.

Peng and Regia [5] have also shown that the causation relation C can be extended for the representation of multi-layered causal nets, in which defects are causally connected to each other, finally leading to observable findings. By computation of the reflexive, transitive closure of the causation relation, $C^\star$, the basic techniques discussed above immediately apply. The reflexive closure makes it possible to enter defects as observed findings, which are interpreted as already established defects, yielding a slight extension to the theory treated above.

4.4 Non-monotonic reasoning

From an AI point of view, one of the interesting features of abductive diagnosis, as is also true for consistency-based diagnosis, is that these are examples of *non-monotonic* reasoning. By this we mean that more knowledge does not always give yield more results, as is, in contrast, always true for standard logic (that underlies mathematics).

To get insight into the non-monotonic nature of model-based diagnosis, the key idea in abductive diagnosis is to reverse the logical implication symbol $\rightarrow$ and add the resulting formula to the causal model $\mathcal{R}$. The result is called the *completion*, as now any finding f observed can be deducibly associated to an explanation in terms of defects d. For example, assume that we have $d \rightarrow f$ (meaning 'd causes f'), then adding $d \leftarrow f$ yields $d \leftrightarrow f$. But now we have

$$\{d \leftrightarrow f\} \cup \{f\} \vdash d$$

so, we have derived the diagnosis (explanation for f) d by ordinary (monotonic) logical deduction. However, we needed the completion of the causal model to make it possible. See now the slides about the relationship between abduction and completion.

Understanding the non-monotonic nature of consistency-based diagnosis is more difficult. One common approach, which we have adopted in the lecture, is to map consistency-based diagnosis to a non-monotonic logic, such as *default logic*. Then is becomes possible to characterise the computation of a diagnosis as reasoning in this logic, which then by definition is non-monotonic.

In default logic, one adds special inference rules to what is already available to predicate logic. The general form of a *default* is as follows:

$$\frac{\textit{prerequisite} : \textit{justifications}}{\textit{consequent}}$$

where 'prerequisite' is a condition that most be true, as is the case with 'justifications', and only then 'consequent' can be derived, however, *only* when it is not the case that an inconsistency occurs.

There are various ways in which default logic can be used to model particulay ways of reasoning:

- Prototypical reasoning ("typically children have parents"):

$$\frac{\mathit{Child}(x) : \mathit{HasParents}(x)}{\mathit{HasParent}(x)}$$

- No-risk reasoning ("assume that the accused is innocent unless you know otherwise"):

$$\frac{\mathit{Accused}(x) : \mathit{Innocent}(x)}{\mathit{Innocent}(x)}$$

- Autoepistemic reasoning ("the tutorial will be on Wednesday, unless moved"):

$$\frac{: \mathit{Tutorial\text{-}on\text{-}Wednesday}}{\mathit{Tutorial\text{-}on\text{-}Wednesday}}$$

Reasoning in default logic starts with a *default theory* $\mathrm{KB} = (W, D)$, where W is a set of logical formulae, and D the set of defaults. Computation of what can be derived from KB is done using a derivation operator:

- $E = \mathrm{Th}(E)$ (so-called fixed point, we have reached a stable situation: nothing more can be derived);
- $W \subseteq E$ (hence, we do not get less than the predicate logical formulae we start with);
- E includes the maximal set of conclusions obtained by applying defaults in D;
- If $\frac{A\,:\,B_1,\ldots,B_n}{C} \in D$, $A \in E$ and $\neg B_1, \ldots, \neg B_n \notin E$ (E is consistent with the justification $B_1, \ldots, B_n$), then C is added to E, i.e., $C \in E$.

E is called an *extension* and Th is the derivation operator (deduction + default rule application). Note that if the set of default rules D is empty, then Th just amounts to computing all logical consequences of W (See Appendix A).

EXAMPLE 4.6

Consider the following simple default theory:

$$\mathrm{KB} = \left(\{P(a)\}, \left\{\frac{P(a) : Q(a)}{Q(a)}\right\}\right)$$

then

$$E = \mathrm{Th}(\{P(a), Q(a)\}) = \{P(a), Q(a)\}$$

A classical example concerns USA's former, not particular popular, president Richard Nixon, who was both a republican and a quaker, i.e., $\mathrm{KB} = (W, D)$, with

$$W = \{\mathrm{Republican}(\mathrm{Nixon}), \mathrm{Quaker}(\mathrm{Nixon})\}$$

and set of defaults:

$$D = \left\{\frac{\mathrm{Republican}(x) : \neg\mathrm{Pacifist}(x)}{\neg\mathrm{Pacifist}(x)}, \frac{\mathrm{Quaker}(x) : \mathrm{Pacifist}(x)}{\mathrm{Pacifist}(x)}\right\}$$

which express that "rebublicans are typically not pacifist" whereas "quakers are typically pacifists". These rules are clearly mutually exclusive. There are two extensions in this case:

- $E_1 = \{\text{Republican(Nixon)}, \text{Quaker(Nixon)}, \text{Pacifist(Nixon)}\}$
- $E_2 = \{\text{Republican(Nixon)}, \text{Quaker(Nixon)}, \neg\text{Pacifist(Nixon)}\}$

which can be looked on as two alternative solutions.

We only need a very simple default logic in the context of model-based diagnosis (so-called normal defaults). See Section 4.2 and the slides about consistency-based reasoning and default logic.

4.5 The AILog system

David Poole and colleagues have developed a theory and an implementation of a form of hypothetical reasoning, called *AILog*. AILog may be used as a framework of diagnosis, but it is not restricted in any way to diagnostic problem solving [6].

In AILog, a diagnostic problem must be specified in terms of a set of *facts*, denoted by FACTS, a set of *hypotheses*, denoted by HYP, and a set of *constraints*, denoted by C. The set of facts FACTS and constraints C are collections of arbitrary closed formulae in first-order logic; hypotheses act as a kind of defaults that might become instantiated, and assumed to hold true, in the reasoning process. A set $\text{FACTS} \cup H$ is called an *explanation* of a closed formula g, where H is a set of ground instances of hypothesis elements in HYP, iff:

(1) $\text{FACTS} \cup H \vDash g$, and

(2) $\text{FACTS} \cup H \cup C \nvDash \bot$.

```
assumable a1.
assumable a2.
assumable fever.
assumable influenza.
assumable sport.

chills  <- fever & a1.
fever   <- influenza.
thirst  <- fever.
myalgia <- influenza & a2.
myalgia <- sport.

false <- chills. % the set C
create_nogoods.
```

Figure 4.4: Specification of an abductive diagnostic problem in AILog.

On first sight, the framework looks a lot like the framework of abductive diagnosis discussed in Section 4.3.1, but it is much more general, mainly due to the unrestricted nature of its elements. In terms of the abductive theory of diagnosis, we would have called H a diagnosis, if the abnormality axioms $\mathcal{R}$ were taken as FACTS, the set of findings not observed as

constraints C, and the set of observed findings E as g. Obviously, because there is no fixed diagnostic interpretation in AILog, the framework can be used as a basis for various other notions of diagnosis, such as consistency-based diagnosis (just take $g \equiv \top$).

EXAMPLE 4.7

Figure 4.4 presents a specification of the abductive diagnostic problem from Example 4.2 in terms of the AILog implementation, where `C` denotes the set of findings assumed to be absent, C, taken as an integrity constraints in AILog. The following query:

```
ask thirst & myalgia.
```

yields the following results:

```
Answer is thirst & myalgia
Assuming [a2,fever,influenza]

Answer is thirst & myalgia
Assuming [fever,sport]

Answer is thirst & myalgia
Assuming [a2,influenza]

Answer is thirst & myalgia
Assuming [influenza,sport]
```

Theories are solutions in the abductive theory of diagnosis. Only a subset of the solutions mentioned in Example 4.2 are computed, because in AILog it is assumed that every observed finding need be explained only once by a diagnosis.

Chapter 5

Lecture 9 – Reasoning and Decision Making under Uncertainty

Since the early 1970s reasoning with uncertainty has been one of the most important topics on the AI research agenda. The reason is, of course, that there are very few real-world problems where there is no uncertainty involved. Uncertainty usually arises from incomplete knowledge about the world, e.g., noisy sensors (a robot may have to infer its location from imperfect sensors), tests that are imperfect, or measurements that cannot be taken because it is too dangerous (looking inside the brain of a patient, for example), or impossible (the composition of soil on Pluto, for example). Furthermore, contemporary theories on how the human brain functions rely heavily on probability theory.

5.1 Introduction

In the early days of AI research, *rule-based systems* were important, i.e., knowledge systems where knowledge bases were composed from rules of the form $A \rightarrow C$ (premise A implies conclusion C). To represent uncertainty, some measure of uncertainty was attached to the conclusion of rules, yielding: $A \rightarrow C_x$ with x some uncertainty measure, usually meaning that given that when A is known with *absolute* certainty, then C is known with a certainty equal to x.

The rule-based approaches were at the end of the 1980s slowly replaced by graphical representations of probability distributions, so-called *probabilistic graphical models*. In particular Bayesian networks have become important.

Rather ironically (because in the 1990s there were many AI researchers who thought that rule-based approached to reasoning with uncertainty were by definition unsound), the last few years there is revival happening of combining rule-based, logical approaches to reasoning with uncertainty and probabilistic graphical models. An example is Markov logic.

5.2 Rule-based uncertainty knowledge

In rule-based representations of uncertainty one needs:

- a way to *represent* uncertainty, and

- a method to propagate the uncertainty from evidence (observations) to conclusions, called *propagation rules*.

The representation of uncertainty is straightforward. Most methods simply associate a measure of uncertainty x to to conclusion of a logical rule, as follows:

$$e_1 \wedge \cdots \wedge e_n \to h_x$$

meaning that $e_1, e_2, \ldots, e_n$ are true (observed), then conclusion h is true with certainty x.

If we assume that the rules have a restricted syntax, for example only $\vee$ and $\wedge$ are allow in the premises of rules, then there are three (classes) of propagation rules that are required:

1. $f_\wedge$ and $f_\vee$: combining uncertain evidence in the premise of a rule;
2. f_{prop}: the propagation of uncertainty from the (uncertain) premise to the conclusion of a rule;
3. f_{co}: combining the uncertain conclusions of rules if they have the same conclusion.

The slides give definitions of these functions for a one popular method for uncertainty reasoning developed for the MYCIN knowledge system, called the *certainty-factor calculus*. Have a look at these definitions:

- $f_\wedge$: $\text{CF}(e_1 \wedge e_2, e') = \min\{\text{CF}(e_1, e'), \text{CF}(e_2, e')\}$
- $f_\vee$: $\text{CF}(e_1 \vee e_2, e') = \max\{\text{CF}(e_1, e'), \text{CF}(e_2, e')\}$
- f_{prop}: $\text{CF}(h, e') = \text{CF}(h, e) \cdot \max\{0, \text{CF}(e, e')\}$
- f_{co}: Let $\text{CF}(h, e'_1) = x$ and $\text{CF}(h, e'_2) = y$. Then

$$\text{CF}(h, e'_1 \textbf{ co } e'_2) = \begin{cases} x + y(1-x) & \text{if } x, y > 0 \\ \frac{x+y}{1-\min\{|x|,|y|\}} & \text{if } -1 < xy \leq 0 \\ x + y(1+x) & \text{if } x, y < 0 \end{cases}$$

For example, in the definition of f_{prop}, the max operator makes sure that only positive or zero uncertainties are propagated to the conclusion, as CFs are numbers between -1 and 1 (inclusive), and a condition that has a negative uncertainty attached fails (yielding 0). The fact that CFs can also be negative also explains why there are three entries in the definition of f_{co}. For positive CFs only the first entry is important, and this is the entry which we use when we give CFs a *probabilistic interpretation*.

5.3 Probabilistic graphical models

The rule-based representation enforces a restricted use of uncertainty: only from the premise to the conclusion of rules, and always is it necessary to have knowledge about any of the elements in the premise of the rules. So, already from a conceptual point of view there are reasons to find the early rule-based approaches too restrictive.

The breakthrough come from realising that reasoning with joint (multivariate) probability distributions becomes feasible if one is able to make assumptions of (statistical) independence

between random variables. In probabilistic graphical models, such *independence assumptions* are represented by means of a graph or network and the associated probability distribution respects those assumptions. One of the most successful probabilistic graphical models are *Bayesian networks*; their graph representation is that of an acyclic, directed graph, i.e., all links are arcs (*directed* edges or arrows), and there are no directed cycles in the graph. *Markov networks* are other examples of probabilistic graphical models; their graph consists of only *undirected* edges.

Formally, a Bayesian network is a pair $\mathcal{B} = (G, P)$, where $G = (V, E)$ is an acyclic directed graph, with V a set of vertices or nodes, $E \subseteq V \times V$ the set of arcs, and P the associated joint (multivariate) probability distribution that is defined such that there is for each vertex in G a random variable in P, and:

$$P(X_V) = \prod_{v \in V} P(X_v \mid X_{\pi(v)}) \tag{5.1}$$

where $\pi(v)$ is the set of parents of vertex v. The expression: $P(X_v \mid X_{\pi(v)})$ means that X_v is the random variable that corresponds to the vertex v, and $X_{\pi(v)}$ is the *set* of random variables that correspond to the set of parents of vertex v. $P(X_v \mid X_{\pi(v)})$ is then the conditional probability distribution.[1] The multiplication rule (5.1) says that of one multiplies all conditional probability distributions associated with each variable X_v, when a joint probability distribution is obtained.

Consider the following example. We have the Bayesian network with acyclic directed graph $1 \rightarrow 2$, and associated joint probability distribution:

$$P(X_1, X_2) = P(X_2 \mid X_1)P(X_1)$$

Note that we have followed the structure of the graph in our definition: vertex 2 has parent 1, and vertex 1 has no parents. This explains why we have the two factors: $P(X_2 \mid X_1)$ and $P(X_1)$. Now consider the following more extensive Bayesian network, also included in the slides.

EXAMPLE 5.1

Consider Figure 5.1, which shows a simplified version of a Bayesian network modelling some of the relevant variables in the diagnosis of two causes of fever. The presence of an arc between two vertices denotes the existence of a direct causal relationship or other influences; absence of an arc means that the variables do not influence each other directly. The following knowledge is represented in Figure 5.1: variable 'FL' is expressed to influence 'MY' and 'FE', as it is known that flu causes myalgia (muscle pain) and fever. In turn, fever causes a change in body temperature, represented by the random variable TEMP. Finally, pneumonia (PN) is another cause of fever.

For Figure 5.1, the conditional probability table

$$P(\text{FE} \mid \text{FL}, \text{PN})$$

has been assessed with respect to all possible values of the variables FE, FL and PN. In general, the graph associated with a Bayesian network mirrors the (in)dependences

[1] For convenience, we often abuse notation and write, e.g., $\pi(X_v)$ instead of $X_{\pi(v)}$.

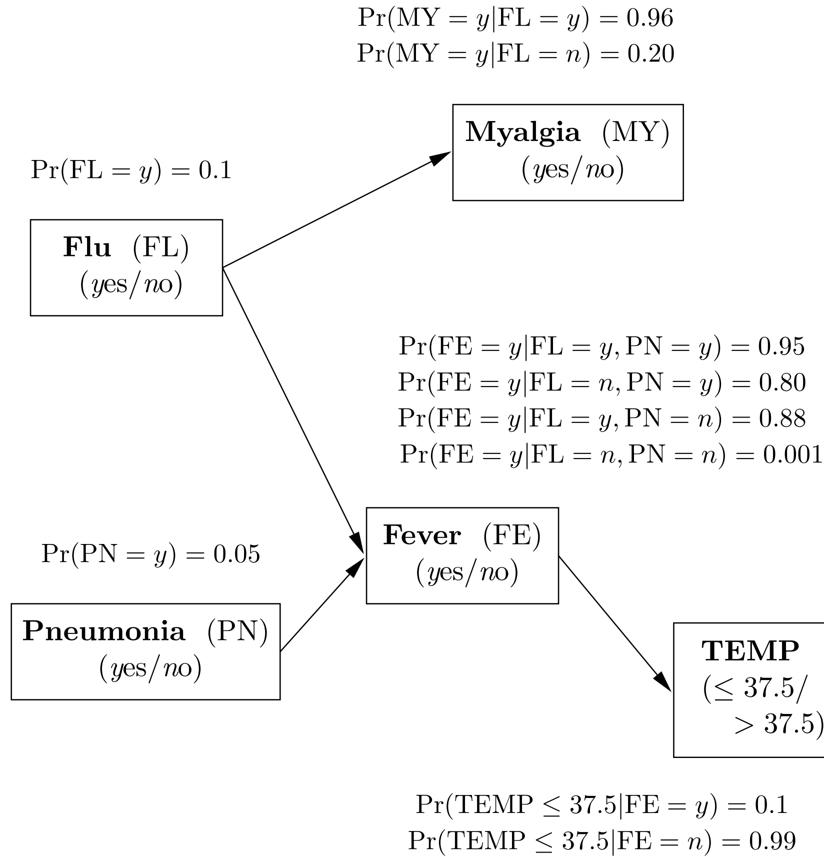

Figure 5.1: Bayesian network $\mathcal{B} = (G, P)$ with associated joint probability distribution P (only probabilities $P(X_v = y \mid X_{\pi(v)})$ are shown, as $P(X_v = n \mid X_{\pi(v)}) = 1 - P(X_v = y \mid X_{\pi(v)})$).

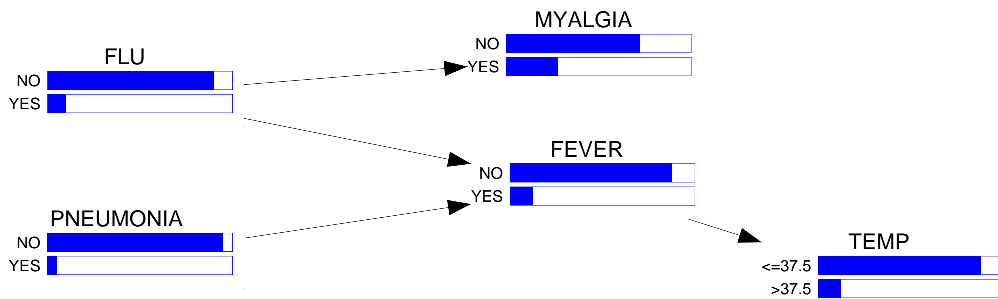

Figure 5.2: Prior marginal probability distributions for the Bayesian network shown in Figure 5.1.

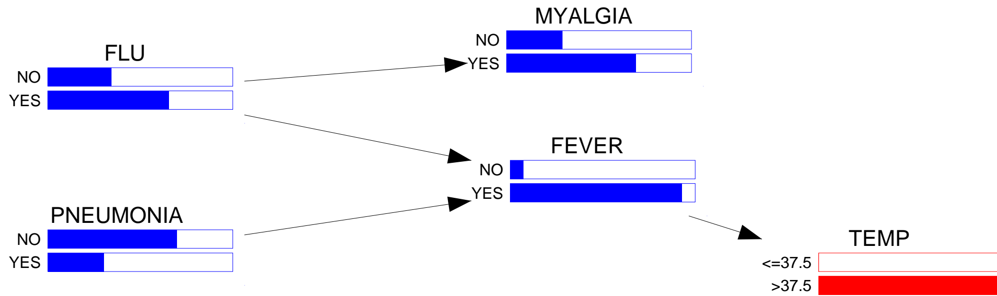

Figure 5.3: Posterior marginal probability distributions for the Bayesian network after entering evidence concerning body temperature. Note the increase in probabilities of the presence of both flu and pneumonia compared to Figure 5.2. It is also predicted that it is likely for the patient to have myalgia.

that are assumed to hold among variables in a domain. For example, given knowledge about presence or absence of fever, neither additional knowledge of flu nor of pneumonia is able to influence the knowledge about body temperature, since it holds that TEMP is conditionally independent of both PN and FL given FE. The marginal probability distribution $P(V_i)$ for every variable in the network can be computed; this is shown for the fever network in Figure 5.2. In addition, a once constructed Bayesian belief network can be employed to enter and process data of a specific case, i.e. specific values for certain variables, like TEMP, yielding an updated network. Figure 5.3 shows the updated Bayesian network after entering evidence about a patient's body temperature into the network shown in Figure 5.1. Entering evidence in a network is also referred to as *instantiating* the network.

We continue with the simple example. If we assume that we have the following definitions for $P(X_2 \mid X_1)$ and $P(X_1)$ yielding $P(X_1, X_2) = P(X_2 \mid X_1)P(X_1)$:

$$P(x_2 \mid x_1) = 0.2$$
$$P(x_2 \mid \neg x_1) = 0.3$$
$$P(x_1) = 0.4$$

(we use x_i as notation for $X_i = \mathit{yes}$ and $\neg x_i$ for $X_i = \mathit{no}$). Then we know that it must hold that $P(\neg x_2 \mid x_1) = 0.8$, as according to the axioms of probability theory $P(x_2 \mid x_1) + P(\neg x_2 \mid x_1) = 1$; similarly, $P(\neg x_2 \mid \neg x_1) = 0.7$ and $P(\neg x_1) = 0.6$. We therefore have the following joint probability distribution:

$$\begin{aligned}
P(x_1, x_2) &= P(x_2 \mid x_1)P(x_1) = 0.2 \times 0.4 = 0.08 \\
P(\neg x_1, x_2) &= P(x_2 \mid \neg x_1)P(\neg x_1) = 0.3 \times 0.6 = 0.18 \\
P(x_1, \neg x_2) &= P(\neg x_2 \mid x_1)P(x_1) = 0.8 \times 0.4 = 0.32 \\
P(\neg x_1, \neg x_2) &= P(\neg x_2 \mid \neg x_1)P(\neg x_1) = 0.7 \times 0.6 = 0.42
\end{aligned}$$

Note that $\sum_{X_1,X_2} P(X_1, X_2) = P(x_1, x_2) + P(\neg x_1, x_2) + P(x_1, \neg x_2) + P(\neg x_1, \neg x_2) = 1$, as with any probability distribution.

The interesting thing of joint probability distributions is that *any (conditional) probability distribution can be computed from them* using the marginalisation and conditioning rules. Using marginalisation, it is easy to compute $P(x_2)$, as we have that

$$P(x_2) = \sum_{X_1} P(X_1, x_2) = P(x_1, x_2) + P(\neg x_1, x_2) = 0.08 + 0.18 = 0.26$$

Using conditioning, it easy to compute $P(x_1 \mid x_2)$, as follows:

$$P(x_1 \mid x_2) = \frac{p(x_1, x_2)}{P(x_2)} = 0.08/0.26 = 4/13$$

In this case, I used the definition of conditional probability distributions.

This type of reasoning has been called *naive* probabilistic reasoning, as it may involve a lot of, also redundant, work. Probabilistic reasoning is usually done using special-purpose algorithms, such as (approximate) particle-based (sampling) algorithms, variable elimination, or Pearl's algorithm, which is based on a message-passing scheme.

A particularly important rule, which can be derived from the chain rule and commutativity of conjunction is Bayes' rule. We write $P(h, e) = P(h \mid e)P(e) = P(e \mid h)P(h)$. If $P(e) \neq 0$, you can divide the right hand sides by $P(e)$:

$$P(h \mid e) = \frac{P(e \mid h)P(h)}{P(e)}$$

This is Bayes' rule. It is an important rule since it allows quantities of interest $P(h \mid e)$ to be expressed in terms of a likelihood $P(e \mid h$ and a prior $P(h)$ which are often much easier to estimate. Bayes' rule can be interpreted as reasoning in the opposite direction of the arrow in a Bayesian network $h \rightarrow e$.

An interesting result, mentioned in the slides, is that it is possible to map the certainty-factor calculus of rule-based systems to Bayesian networks. This gives insight into the assumptions underlying the certainty-factor calculus. So, maybe, in the end, the designers of the rule-based uncertainty schemes were not as stupid as some lecturers sometimes say! What is your opinion about this?

5.4 Towards Decision Making

Logic and probability theory provide necessary and sufficient tools in order to allow an agent to reason in a rational way. However, in the real world, reasoning alone does not ensure the agent's survival. It is rather the actions that are executed based on an agent's inferences that implement optimal behaviour. This selection of optimal action based on an agent's knowledge is the subject matter of *decision theory.* Decision theory is also crucial in the development of expert systems. Often, a physian would like to get advice on which action to perform given observed symptoms rather than just knowing what disease is the most likely cause of the symptoms. Furthermore, sometimes actions themselves update our state of knowledge. Think for instance of laboratory tests which may or may not be performed for a particular patient. Our notation and examples are inspired by the textbook by David Poole [15], which is also freely available on his website: http://artint.info/. Chapter 13 and 14 of the AI book by Russell and Norvig [9] cover this topic too.

5.5 Preferences and utilities

An action $a \in \mathcal{A}$ will always result in an outcome $o \in \mathcal{O}$. Agents have preferences over outcomes and a rational agent will always choose that action which leads to the optimal outcome. If o_1 and o_2 are outcomes:

- $o_1 \succeq o_2$ means o_1 is at least as desirable as o_2
- $o_1 \sim o2$ means $o_1 \succeq o_2$ and $o_2 \succeq o_1$
- $o_1 \succ o_2$ means $o_1 \succeq o_2$ but not $o_2 \succeq o_1$

An action may not always lead to one particular outcome but rather to a probability distribution over outcomes. This is known as a *lottery*, written as

$$p_1 : o_1; p_2 : o_2; \ldots ; p_k : o_k$$

where the o_i are outcomes and $p_i > 0$ such that $\sum_i p_i = 1$. Preferences are formalized in terms of various axioms and a rational agent should obey these axioms. The following axioms hold:

- *Completeness*: $\forall o_1 \forall o_2 \;\; o_1 \succeq o_2$ or $o_2 \succeq o_1$. The rationale for this axiom is that an agent must act; if the actions available to it have outcomes o_1 and o_2 then, by acting, it is explicitly or implicitly preferring one outcome over the other.
- *Transitivity*: if $o_1 \succeq o_2$ and $o_2 \succeq o_3$ then $o_1 \succeq o_3$
- *Monotonicity*: if $o_1 \succ o_2$ and $p > q$ then $[p\colon o_1, 1-p\colon o_2] \succ [q\colon o_1, 1-q\colon o_2]$. Note that, in this axiom, $>$ between outcomes represents the agent's preference, whereas $>$ between p and q represents the familiar comparison between numbers.
- *Decomposability*: Indifference between lotteries (over lotteries) with the same probabilities and outcomes: $[p\colon o_1, 1-p\colon [q\colon o_2, 1-q\colon o_3]] \sim [p\colon o_1, (1-p)q\colon o_2, (1-p)(1-q)\colon o_3]$. This axiom formalizes the indifference between lotteries (no fun in gambling).
- *Substitutability*: if $o_1 \sim o_2$ then the agent is indifferent between lotteries that only differ by o_1 and o_2: $[p\colon o_1, 1-p\colon o_3] \sim [p\colon o_2, 1-p\colon o_3]$
- *Continuity*: suppose $o_1 \succ o_2$ and $o_2 \succ o_3$ then there exists a $p \in [0,1]$ such that $o_2 \sim [p\colon o_1, 1-p\colon o_3]$

If an agent is rational, then the preference of an outcome can be quantified using a *utility function*:

$$U\colon \mathcal{O} \rightarrow [0,1]$$

such that:

- $o_1 \succeq o_2$ if and only if $U(o_1) \geq U(o_2)$.
- $U([p_1\colon o_1, p_2\colon o_2, \ldots, p_k\colon o_k]) = \sum_{i=1}^{k} p_i \cdot U(o_i)$

People are often risk-aversive. What do you prefer? 50/50 chance for 0 or 1000000 or 300000 now? For this utility function, U(999000) approx 0.9997. Thus, given this utility function, the person would be willing to pay 1000 to eliminate a 0.03% chance of losing all of their money. This is why insurance companies exist. By paying the insurance company, say 600, the agent can change the lottery that is worth 999,000 to them into one worth 1,000,000 and the insurance companies expect to pay out, on average, about 300, and so expect to make 300. The insurance company can get its expected value by insuring enough houses. It is good for both parties.

5.6 Decision problems

What an agent should do depends on:

- The agent's ability: what options are available to it.
- The agent's beliefs: the ways the world could be, given the agent's knowledge. Sensing updates the agent's beliefs.
- The agent's preferences: what the agent wants and tradeoffs when there are risks.

Decision theory specifies how to trade off the desirability and probabilities of the possible outcomes for competing actions. In order to represent decisions, we make use of *decision variables*. They are like random variables that an agent gets to choose a value for. For a single decision variable, the agent can choose $D = d$ for any $d \in \mathrm{dom}(D)$. The expected utility of decision $D = d$ is

$$E(U \mid D = d) = \sum_{\omega \models d} P(\omega \mid d) U(\omega)$$

That is, we sum over all possible worlds that imply the decision and for each world, we take the product of the (conditional) probability of that world times the utility of that world. An optimal single decision is the decision $D = d_{\max}$ whose expected utility is maximal:

$$d_{\max} = \arg \max_{d \in \mathrm{dom}(D)} E(U \mid D = d)$$

One way to represent a decision problem is using a *decision tree*. Decision trees are a way to graphically organise a sequential decision process. It contains chance nodes (random variables) and decision nodes, each with branches for each of the alternative decisions. The utility of each branch is computed at the leaf of each branch and the expected utility of any decision is computed on the basis of the weighted summation of all branches from the decision to all leaves from that branch. Figure 5.4 shows a very simple decision tree, which depicts the problem of deciding whether or not to go to a party depending on the weather. The respectively utilities are computed as follows:

$$U(party) = \sum_{Rain} U(party, Rain) P(Rain) = -100 \times 0.6 + 500 \times 0.4 = 140$$

$$U(no-party) = \sum_{Rain} U(no-party, Rain) P(Rain) = 0 \times 0.6 + 50 \times 0.4 = 20$$

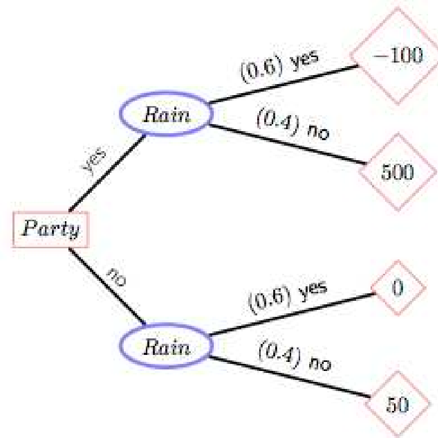

Figure 5.4: Deciding to go to the party or not. After [16].

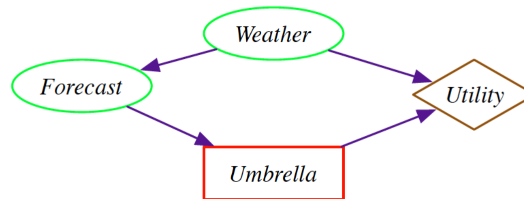

Figure 5.5: The umbrella network. After [16].

An intelligent agent doesn't carry out just one action or ignore intermediate information. A more typical scenario is where the agent: observes, acts, observes, acts, etc. Subsequent actions can depend on what is observed and what is observed depends on previous actions. Often the sole reason for carrying out an action is to provide information for future actions. Solving a sequence of decisions can be achieved using a decision tree by rolling back the tree. However, decision trees grow exponentially fast in the number of variables. A sequential decision problem consists of a sequence of decision variables $D_1, \ldots, D_n$ where each D_i has an information set of variables $\pi(D_i)$ whose value will be known at the time decision D_i is made. The decision nodes are totally ordered. This is the order the actions will be taken. All decision nodes that come before D_i are parents of decision node D_i. Thus the agent remembers its previous actions. Any parent of a decision node is a parent of subsequent decision nodes. Thus the agent remembers its previous observations. Total utility is given by the sum of the independent utilities (additive utility).

One way to represent a *finite* sequential decision problem is by means of an influence diagram. Influence diagrams extend belief networks to include decision variables and utility. They specify what information is available when the agent has to act and which variables the utility depends on. Figure 5.5 shows an example of an influence diagram concerning whether or not to wear an umbrella.

5.7 Optimal policies

What an agent should do under all circumstances is formalized by a *policy*. It is a sequence $\delta_1, \ldots, \delta_N$ of decision functions

$$\delta_i : dom(\pi(D_i)) \to dom(D_i)$$

meaning that when the agent has observed $O \in dom(\pi(D_i))$, it will do $\delta_i(O)$. The expected utility of policy δ is

$$E(u \mid \delta) = \sum_{\omega \models \delta} u(\omega) \times P(\omega)$$

and an *optimal* policy is one with the highest expected utility. Finding the optimal policy is equivalent to solving a decision problem. The recipe for finding an optimal policy using an influence diagram, exemplified with the umbrella network, is as follows:

- Let us take the conditional probability tables and the table for the utility (the so-called *factors*)

Weather	Value
norain	0.7
rain	0.3

Weather	Fcast	Value
norain	sunny	0.7
norain	cloudy	0.2
norain	rainy	0.1
rain	sunny	0.15
rain	cloudy	0.25
rain	rainy	0.6

Weather	Umb	Value
norain	take	20
norain	leave	100
rain	take	70
rain	leave	0

- In spirit of the value elimination algorithm, one can systematically remove factors by maximization (since we want the optimal policy) and computing new factors. For example, here we compute the factor for the utility of the combination of the decision and the forecast ($U(\text{Fcast}, \text{Umb})$). That is, $U(\text{sunny}, \text{take}) = P(\text{sunny}|\text{rain}) \times P(\text{rain}) \times U(\text{rain}, \text{take}) + P(\text{sunny}|\text{norain}) \times P(\text{norain}) \times U(\text{norain}, \text{take}) = 0.15 \times 0.3 \times 70 + 0.7 \times 0.7 \times 20 = 12.95$. This leaves use with:
  - the optimal decision function for D, $\arg\max_D f$
  - a new factor to use (e.g. in variable elimination), $\max_D f$ (or more generally; for computing the optimal decision)

f:

Fcast	Umb	Val
sunny	take	12.95
sunny	leave	49.0
cloudy	take	8.05
cloudy	leave	14.0
rainy	take	14.0
rainy	leave	7.0

$\max_{Umb} f$:

Fcast	Val
sunny	49.0
cloudy	14.0
rainy	14.0

$\arg\max_{Umb} f$:

Fcast	Umb
sunny	leave
cloudy	leave
rainy	take

- Such calculations are repeated untill there are no more decision nodes to consider.
- If multiple factors are computed, one can multiply the factors: this is the expected utility of the optimal policy (in this example we have only one factor to consider).

This approach to solving a sequential decision problem only holds for finite-horizon problems. That is, problems consisting of a fixed number of time steps. In case of infinite or indefinite horizon decision problems, we typically formalize them in terms of (partially observable) Markov decision processes, which make use of other solution methods such as value iteration or policy iteration. Such decision problems are common in, for example, planning problems where robots need to traverse a space in order to reach a goal.

Chapter 6

Lecture 10 – Probabilistic logic

There have been different recent proposals in the AI literature to combine logic and probability theory, where usually predicate logic is combined with probabilistic graphical models. David Poole has developed so-called *independent choice logic* (which later was integrated into AIlog). It combined Prolog-like logic with Bayesian networks. Another approach, developed by Williamson et al. makes use of *credal networks*, which are similar to Bayesian networks but reason over probability intervals instead of probabilities. The last few years *Markov logic* has had an enormous impact on the research area. The idea is to use predicate logic to *generate* Markov networks, i.e., joint probability distributions that have an associated *undirected* graph. Formalisms such as independent choice logic and Markov logic are examples of what is called *probabilistic logic*.

Section 14.6 of Russell and Norvig [9] features a brief introduction to probabilistic relational models. Also, Chapter 14 of the book by Poole [15] features the combination of logic and probability, especially in Section 14.3.

6.1 Probabilistic logic based on logical abduction

Various probabilistic logics, such as the independent choice logic, are based on logical abduction. The basic idea of these kind of logics is to define the probability of a query in terms of the probability of its *explanations* (sometimes called a *prediction* in theory of logical abduction) of a certain query (cf. Section 4.5) given a logic program. Probability of the explanations are defined by a very simple distribution, namely by a set of independent random variables, which makes it possible to (relatively) efficiently compute a probability. The nice thing about this approach is that it truely combines logical reasoning (finding the explanations) with probabilistic reasoning (computing the probability of the set of explanations).

Defining the probability distributions over the explanations is done by associating probabilities to hypotheses in a set Δ. In order to make sure that we end up with a valid probability distribution, we require a partitioning of this set into subsets $\Delta_1, \ldots, \Delta_n$, i.e., such that it holds that:

$$\bigcup_{i=1}^{n} \Delta_i = \Delta$$

and $\Delta_i \cap \Delta_j = \emptyset$ for all $i \neq j$. Each possible grounding of Δ_i, i.e. $\Delta_i\sigma$ with σ a substition, is associated to a random variable $X_{i,\sigma}$, i.e., $dom(X_{i,\sigma}) = \Delta_i\sigma$. While you could imagine that

every random variable is different, here we will assume that every grounding of $h \in \Delta$ has to have the same probability, i.e., for all substitutions σ, σ':

$$P(X_{i,\sigma} = h\sigma) = P(X_{i,\sigma'} = h\sigma')$$

Whereas each pair of random variables as we have just defined is assumed to be independent, the hypotheses *in the same partition* are dependent. Suppose for example, we have a random variable X with three possible hypotheses:

$$dom(X) = \{influenza, sport, not_sport_or_influenza\}$$

In each possible state (element of the sample space), each random variable is exactly in one state at the time, i.e., in this case, we assume that we either have influenza, or we sport, or neither, but we do not sport while we have influenza. In other words: sport and influenza are considered to be inconsistent.

To understand the space of explanations that we may consider is by picking a possible value for each random variable. In the language of the *independent choice logic*, this is called a *choice* (hence, the name). In order to make this work probabilistically, we need some slight restrictions on our logic program. First, it is not allowed to have two hypotheses in Δ that unify. Further, it is not allowed that an element from Δ unifies with a head of one of the clauses. Finally, mostly for convenience here, we will restrict ourselves to acyclic logic programs consisting of Horn clauses and substitutions that can be made using the constants in the program.

The probability distribution over Δ is now used to define a probability for arbitrary atoms. As mentioned earlier, this will be defined in terms of explanations, which are slightly different than we have seen before due to the probabilistic semantics. Given a causal specification $\Sigma = (\Delta, \Phi, \mathcal{R})$, a (probabilistic) explanation $E \subseteq \Delta\sigma$ for some formula $F \in \Phi$ is:

$$\mathcal{R} \cup E \models F$$
$$\mathcal{R} \cup C \cup E \not\models \bot$$

where

$$C = \{\bot \leftarrow h_1, h_2 \mid \Delta_i \text{ is one of the partitions of } \Delta, h_1, h_2 \in \Delta_i\}$$

and $\Delta\sigma$ grounded. Note that the consistency condition entails that we only pick at most one value for each random variable. The intuitive assumption that is now being made is that an atom is true if and only if at least one of its (grounded) explanations is true. Suppose $\mathcal{E}(F)$ is the set of all explanations for F, then we define:

$$F = \bigvee_{E_i \in \mathcal{E}(F)} E_i$$

Notice that this definition is equivalent to assuming Clarke's completion of the given theory (cf. Section 4.3.1).

Recall that an explanation E is called minimal if there does not exist an explanation E' such that $E' \subset E$. It is not difficult to see that we can restrict our attention to the set of *minimal explanations* $\mathcal{E}_m(F)$: by logical reasoning it holds that, if $E' \subset E$ then $E' \vee E = E'$, so it can be shown that $\mathcal{E}(F) = \mathcal{E}_m(F)$. We then have:

$$F = \bigvee_{E_i \in \mathcal{E}_m(F)} E_i$$

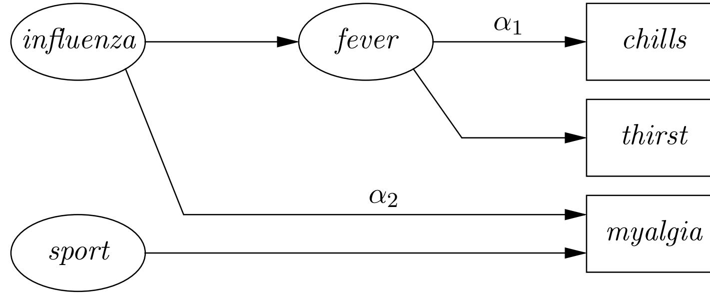

Figure 6.1: A knowledge base with causal relations.

Again, there is a close connection to the semantics of abduction, as $\bigvee_{E_i \in \mathcal{E}_m(F)} E_i$ is sometimes referred to as the *solution formula*. Of course, if two things are equal, then their probability must be equal:

$$P(F) = P(\bigvee_{E_i \in \mathcal{E}_m(F)} E_i)$$

It is now clear how we can solve the problem of computing the probability of F: first we find the (minimal) explanations of F and then we use the probability distribution defined over the hypotheses to compute the disjunction of the explanations.

EXAMPLE 6.2

Consider the causal specification $\Sigma = (\Delta, \Phi, \mathcal{R})$, with

$$\Delta = \{\mathit{influenza}, \mathit{sport}, \mathit{not_sport_or_influenza}, \alpha_1, \mathit{not_}\alpha_1, \alpha_2, \mathit{not_}\alpha_2\}$$

and

$$\Phi = \{\mathit{chills}, \mathit{thirst}, \mathit{myalgia}\}$$

and the set of logical formulae $\mathcal{R}$ that was presented before in context of abduction:

$$\begin{array}{l} \mathit{fever} \wedge \alpha_1 \rightarrow \mathit{chills} \\ \mathit{influenza} \rightarrow \mathit{fever} \\ \mathit{fever} \rightarrow \mathit{thirst} \\ \mathit{influenza} \wedge \alpha_2 \rightarrow \mathit{myalgia} \\ \mathit{sport} \rightarrow \mathit{myalgia} \end{array}$$

The corresponding *causal net*, is shown in Figure 6.1.

First we need to define a probability distribution over Δ. For example, we may assume to have three independent random variables X, Y, Z, such that:

$$\begin{array}{l} P(X = \mathit{sport}) = 0.3 \\ P(X = \mathit{influenza}) = 0.1 \\ P(X = \mathit{not_sport_or_influenza}) = 0.6 \\ P(Y = \alpha_1) = 0.9 \\ P(Y = \mathit{not_}\alpha_1) = 0.1 \\ P(Z = \alpha_2) = 0.7 \\ P(Z = \mathit{not_}\alpha_2) = 0.3 \end{array}$$

Note that explanations containing e.g., *sport* and *influenza* are inconsistent with this probability distribution, as X can only take the value of one of them (they are mutually exclusive).

Suppose we have interested in the probability of myalgia, i.e., $P(myalgia)$. The set of all minimal explanations for myalgia, i.e., $\mathcal{E}_m(myalgia)$ is $\{E_1, E_2\}$, where:

$$E_1 = \{influenza, \alpha_2\}$$
$$E_2 = \{sport\}$$

Clearly, there are many more explanations, e.g.,

$$E_3 = \{influenza, sport, \alpha_2\}$$
$$E_4 = \{influenza, \alpha_1, \alpha_2\}$$
$$E_5 = \{influenza, not_\alpha_1, \alpha_2\}$$
$$\ldots$$

Note that for example, the set:

$$E' = \{influenza, \alpha_1, not_\alpha_1, \alpha_2\}$$

is inconsistent, because α_1 and not_α_1 cannot both be true. Therefore, it is not an explanation.

Since we assumed that a formula is true if only if at least one of its explanations is true, the probability of myalgia is defined it terms of influenza and sport:

$$P(myalgia) = P((influenza \wedge \alpha_2) \vee sport)$$

Since $influenza \wedge \alpha_2$ and *sport* are mutually exclusive, the probability of the disjunction is the sum of the disjuncts, i.e.:

$$\begin{aligned} P(myalgia) &= P(influenza \wedge \alpha_2) + P(sport) \\ &= P(influenza)P(\alpha_2) + P(sport) \\ &= 0.1 \cdot 0.7 + 0.3 = 0.37 \end{aligned}$$

From a computational point of view, the question is how to obtain the relevant explanations. Observe that trying all subsets of groundings of Δ would be wildly inefficient: there are an exponential amount of explanations that we need to consider. Moreover, if we have a first-order theory, then there might be an infinite amount of groundings of our set of hypotheses. Luckily, it can be done more efficiently using logic programming techniques. In particular, it can be shown that explanations can be found using SLD resolution: each proof for a query q found by SLD that uses a consistent set of (ground) hypotheses is an explanation for q. Moreover, if we ensure that in each SLD proof for a given query, the hypotheses that we need to prove are grounded, all explanations can be found using SLD resolution.

EXAMPLE 6.3

Consider the causal specification of Example 6.2. If we assume the hypotheses are true, then we find two SLD resolution proofs for 'myalgia' (written in logic programming notation):

$$\cfrac{\cfrac{\leftarrow \mathit{myalgia} \quad \mathit{myalgia} \leftarrow \mathit{sport}}{\leftarrow \mathit{sport}} \qquad \mathit{sport} \leftarrow}{\Box}$$

and

$$\cfrac{\cfrac{\cfrac{\leftarrow \mathit{myalgia} \quad \mathit{myalgia} \leftarrow \mathit{influenza}, \alpha_2}{\leftarrow \mathit{influenza}, \alpha_2} \qquad \mathit{influenza} \leftarrow}{\leftarrow \alpha_2} \qquad \alpha_2 \leftarrow}{\Box}$$

As these are the only two SLD proofs, we have found all its explanations. The first proof uses the hypothesis *sport*, whereas the second proof uses the hypotheses *influenza* and α_2.

6.2 Probabilistic reasoning in AILog

The type of reasoning discussed in this chapter is also implemented in the AILog system. For the specification of random variables, the `prob` keyword is used, which comes in two forms. The first form is as you might expect for defining a random variable:

`prob` $a_1 : p_1, \ldots, a_n : p_n$

where a_i are atoms that can be assumed and p_i are probabilities such that each $p_i \in [0, 1]$ and $\sum_{i=1}^{n} p_i = 1$. The second form is:

`prob` $a : p$

where a is again an atom and $p \in [0, 1]$. Here we implicitly define a binary random variable X for every grounding of a such that $P(X = a\sigma) = p$. The other value for X simply has no name.

EXAMPLE 6.4

The causal specification and associated probability distribution as discussed in Example 6.2 is formalised as follows in AILog:

```
prob influenza : 0.1, sport : 0.3, not_influenza_or_sport : 0.6.
prob a1 : 0.9.
prob a2 : 0.7.

chills  <- fever & a1.
fever   <- influenza.
thirst  <- fever.
myalgia <- flu & a2.
myalgia <- sport.
```

To obtain a probability, the `predict` keyword can be used, i.e., to predict the probability of myalgia we obtain:

```
ailog: predict myalgia.
Answer: P(myalgia|Obs)=0.37.
  [ok,more,explanations,worlds,help]: explanations.
  0: ass([],[a2,influenza],0.06999999999999999)
  1: ass([],[sport],0.3)
```

Furthermore, it is possible to find conditional probability, e.g., the probability of myalgia given that we observe fever. To observe atoms, we use the `observe` keyword:

```
ailog: observe fever.
Answer: P(fever|Obs)=0.1.
  [ok,more,explanations,worlds,help]: ok.
ailog: predict myalgia.
Answer: P(myalgia|Obs)=0.6999999999999998.
```

Chapter 7

Lecture 11 – Logic for Dynamic Worlds

In the previous chapters we have dealt with *static* databases, i.e. where rules and facts were fixed. We did, however, see some examples of *new* or *additional* information was obtained and incorporated using *conditioning*, for example in Bayesian networks. Here we move on to a somewhat more general setting in which we use logic to reason about these changes. This is useful for playing games, predicting the future, generating plans for a robot and many other things. In addition, we can also use such formalisms to *interpret* dynamic information, for example coming from video data. In that case we talk about *activity recognition*, a very active area in AI.

The slides of the last two lectures contain most of the necessary information for understanding changing databases. The rest of the story can be found in the excellent book by Brachman and Levesque [1], of which you find two (draft) chapters on Blackboard and the course webpage. These two chapters contain the full story of STRIPS, situation calculus and planning in both formalisms. You will also need these descriptions for the practical exercises. In the following we briefly provide some additional information, in particular on the state-operator model which is not dealt with extensively in the Brachman and Levesque chapters. On the web page you also find a PDF containing several pages from Russell and Norvig's AI text book (2nd version) to provide a little more background on the fundamental *frame*, *ramification* and *qualification* problems.

7.1 States and Operators

The simplest way to represent dynamically changing worlds is to explicitly specify which *states* can occur (and how they are represented) and which *operators* (or: *actions*) can change the state of the world (and how). Dynamic worlds, actions and planning are much related, and we can start seeing this by looking at a simple example. Examples here stem mainly from Levesque's book [REF]. The MONKEY-BANANA problem was first proposed by John McCarthy, the seminal figure in situation calculus. This problem appears in many planning-related contexts, and in the lecture we have seen research studies where they use actual monkeys.

A monkey is in a room where a bunch of bananas is hanging from the ceiling, too

> *high to reach. In the corner of the room is a box, which is not under the bananas. The box is sturdy enough to support the monkey if he climbs on it, and light enough so that he can move it easily. If the box is under the bananas, and the monkey is on the box, he will be high enough to reach the bananas.*

Initially the monkey is on the ground, and the box is not under the bananas. There's a lot the monkey can do:

- Go somewhere else in the room (assuming the monkey is not standing on the box)
- Climb onto the box (assuming the monkey is at the box, but not on it)
- Climb off the box (assuming it is standing on the box)
- Push the box anywhere (assuming the monkey is at the box, but not on it)
- Grab the bananas (assuming the monkey is on the box, under the bananas)

Given these possibilities, it is fairly easy to find an informal plan for the monkey to get something to eat: 1) go to the box, 2) push it until it is right under the bananas, 3) climb onto it, and 4) grab the bananas. This amounts to the fastest plan, but many other possibilities exist.

Now in order to automaticallly find a plan for this problem, we need to formalize it as a search problem. Actually this is very similar to either planning or reasoning in this situation. In general, we need the following concepts in any such problem formulation:

- **States**: a form of *snapshots* of how the world can look like. In our example, a state consist of the locations of the monkey, the box and the bananas.
- **Operators**: actions that can change the state of the world. In our example, moving the box changes the location of the box (and the monkey).
- **Initial state**: the state of the world at the start of problem. In our example, the monkey is not on the box, and the box is not under the bananas.
- **Goal state**: the *desired* state we want to be in. This state is the goal of the whole planning process.

The activity of **planning** consists of computing a *sequence of operators* such that once that sequence is applied starting from the initial state, one will reach the goal state. Note that it is required that each operator should be possible to apply in each consequitive state. In other words, a state should be reachable from another state using an operator. A general planning algorithm is now simple to program, see the following algorithm:

A general planning algorithm in Prolog

```
plan(L) :- initial_state(I), goal_state(G), reachable(I, L, G).
%
reachable(S, [], S).
reachable(S1, [M|L], S3) :- legal_move(S1, M, S2), reachable(S2, L, S3).
```

Thus, a plan tries to find a sequence of legal moves such that the goal state is reachable from the initial state by the application of the sequence of operators.

Let us now first try to find a simple logical model of the problem. Let us assume that each state describes the location of the bananas (b), the monkey (m) and the box (l), as well as whether the monkey is on the box (o, which can take values `y` and `n`), and whether the monkey has the bananas h (which too can take the values `y` and `n`).

We can use this to form states in Prolog as *lists of atoms.* For example, we can represent the initial state as [`loc1`, `loc2`, `loc3`, `n`, `n`] and a goal state will (at least) have as last element in the list a '`y`'.

```
%
initial_state([loc1, loc2, loc3, n, n]).
%
goal_state([_, _, _, _, y]).
%
legal_move([B, M, M, n, H], climb_on, [B, M, M, y, H]).
legal_move([B, M, M, y, H], climb_off, [B, M, M, n, H]).
legal_move([B, B, B, y, n], grab, [B, B, B, y, y]).
legal_move([B, M, M, n, H], push(X), [B, X, X, n, H]).
legal_move([B, _, L, n, H], go(X), [B, X, L, n, H]).
%
```

We can run this simply by querying the `plan` predicate, but this might not be the best option:

```
? - plan(P).
ERROR : Outoflocalstack
```

Note that while planning, it does not matter that we do not yet know where to go to; the goal state will provide that information in the end.

Now, `plan(P)` is obviously a too general query, and it will – unboundedly – search for *any* sequence, , and consequently it will first make the plan longer and longer without actually looking for useful substitutions first (question: what's really happening here?). Better is to *bound* the plan by saying explicitly how many actions you want, for example exactly four:

```
?- plan([X,Y,Z,W]).
X = go(loc3),
Y = push(loc1),
Z = climb_on,
W = grab ;
false.
```

Or, we try to find plans of length five:

```
?- plan([X,Y,Z,W,V]).
X = go(loc3),
Y = push(loc1),
Z = climb_on,
W = grab,
```

```
V = climb_off ;
X = go(loc3),
Y = push(_G1281),
Z = push(loc1),
W = climb_on,
V = grab ;
X = go(loc3),
Y = push(loc1),
Z = go(loc1),
W = climb_on,
V = grab ;
X = go(_G1264),
Y = go(loc3),
Z = push(loc1),
W = climb_on,
V = grab ;
false.
```

Note that, technically, the first solution is ok, but it is not optimal (why?). Can you explain why the uninstantiated variables (e.g. `_G1264`) appear in the plan?

A better solution would be to search for small plans first, and extend the length of possible plans increasingly until a good plan is found.

```
bplan(L) :- tryplan([],L).

tryplan(L,L) :- plan(L).
tryplan(X,L) :- tryplan([_|X],L).

1 ?- bplan(L).
L = [go(loc3), push(loc1), climb_on, grab] ;
L = [go(loc3), push(loc1), climb_on, grab, climb_off] ;
L = [go(loc3), push(_G228), push(loc1), climb_on, grab] ;
L = [go(loc3), push(loc1), go(loc1), climb_on, grab] ;
L = [go(_G211), go(loc3), push(loc1), climb_on, grab] ;
L = [go(loc3), climb_on, climb_off, push(loc1), climb_on, grab] ;
L = [go(loc3), push(loc1), climb_on, climb_off, climb_on, grab] ;
L = [go(loc3), push(loc1), climb_on, grab, climb_off, climb_on] ;
L = [go(loc3), push(loc1), climb_on, grab, climb_off, push(_G202)] ;
L = [go(loc3), push(loc1), climb_on, grab, climb_off, go(_G202)] ;
L = [go(loc3), push(_G231), push(loc1), climb_on, grab, climb_off] ;
L = [go(loc3), push(_G231), push(_G248), push(loc1), climb_on, grab] ;
L = [go(loc3), push(_G231), push(loc1), go(loc1), climb_on, grab] ;
L = [go(loc3), push(loc1), go(loc1), climb_on, grab, climb_off] ;
L = [go(loc3), push(_G231), go(_G231), push(loc1), climb_on, grab] ;
L = [go(loc3), push(loc1), go(_G248), go(loc1), climb_on, grab] ;
L = [go(_G214), go(loc3), push(loc1), climb_on, grab, climb_off] ;
L = [go(_G214), go(loc3), push(_G248), push(loc1), climb_on, grab] ;
```

```
L = [go(_G214), go(loc3), push(loc1), go(loc1), climb_on, grab] ;
L = [go(_G214), go(_G231), go(loc3), push(loc1), climb_on, grab] ;
L = [go(loc3), climb_on, climb_off, push(loc1), climb_on, grab, climb_off]
```

Thus, state-operator models are very simple, and use fixed-order and fixed-size representations. However, they do support very easily all kinds of implementations of planning algorithms as we have seen. For example, it is also easy to add additional knowledge to the planner by creating a predicate `acceptable` to discard any unwanted actions, or to *guide* the planner to specific actions. We can then extend the planner using:

`reachable(S1,[M|L],S3) :- legal_move(S1,M,S2),acceptable(M,S1),reachable(S2,L,S3).`

Note that, because we are using a language such as Prolog here, we can basically add any kind of knowledge to the planning process, creating ever smarter planning algorithms.

7.2 STRIPS

A second type of representation is STRIPS, which forms the foundation for many kinds of *planning languages* such as ADL and PDDL. STRIPS can be formalized in propositional logic, but here we use a relational (first-order) version. A STRIPS operator is defined as

$$\langle Act, Pre, Add, Del \rangle$$

and features four components:

- **action name** *Act*: the name (plus arguments) of the action described in the operator
- **precondition** *Pre*: atoms that must be true in order to apply the action
- **delete list**: *Add*: atoms to be deleted from the current state (those becoming `false`) if the action is applied
- **add list**: *Del*: atoms to be added to the current state (those becoming `true`) if the action is applied

An example operator is

$$\begin{aligned} O = \langle & Go(x,y) \\ & \{At(Monkey,x), On(Monkey,Floor)\}, \\ & \{At(Monkey,x)\}, \{At(Monkey,Y)\}\rangle, \end{aligned}$$

which specifies the `go`-action in our monkey-banana problem.

STRIPS has an *operational* semantics, which means that there is a specific way how to *use* the formalism (for reasoning about the world, for planning, etc.). Let O be an operator and let S be a state, i.e. a set of ground relational atoms. The *operational semantics* of applying O to S is

- first find a *matching* of Pre and S, i.e. find a subset $S' \subseteq S$ and a substitution θ such that $Pre\theta \equiv S'$
- compute the new state as $S'' = (S \setminus Del\theta) \cup Add\theta$.

For our example action, let the current state be $S = \{On(Monkey, Floor), At(Monkey, Loc1), \ldots, etc.\}$
Taking $Go(Loc1, Loc2)$ spawns the new state $S' = \{On(Monkey, Floor), At(Monkey, Loc2), \ldots, etc.\}$
In Prolog it looks like this:

```
action(go(X,Y), [at(monkey,X), on(monkey,floor)],
    [at(monkey,X)], [at(monkey,Y)]).

action(push(B,X,Y),
    [at(monkey,X), at(B,X), on(monkey,floor), on(B,floor)],
    [at(monkey,X), at(B,X)], [at(monkey,Y), at(B,Y)]).

action(climbon(B),
    [at(monkey,X), at(B,X), on(monkey,floor), on(B,floor)],
    [on(monkey,floor)], [on(monkey,B)]).

action(grab(B),
    [on(monkey,box), at(box,X), at(B,X), status(B,hanging)],
    [status(B,hanging)], [status(B,grabbed)]).
```

A planning algorithm in Prolog for STRIPS can now operate on the *lists* specifying the states.

```
plan(State, Goal, Plan):-
    plan(State, Goal, [], Plan).

plan(State, Goal, Plan, Plan):-
  is_subset(Goal, State), nl,
  write_sol(Plan).

plan(State, Goal, Sofar, Plan):-
  action(A, Preconditions, Delete, Add),
  is_subset(Preconditions, State),
  \+ member(A, Sofar),
  delete_list(Delete, State, Remainder),
  append(Add, Remainder, NewState),
  plan(NewState, Goal, [A|Sofar], Plan).

test1(Plan):-
    plan([on(monkey,floor),on(box,floor),at(monkey,loc1),at(box,loc2),
           at(bananas,loc3),status(bananas,hanging)],
    [status(bananas,grabbed)],
                 Plan).
```

The slides and Brachman and Levesque's chapters provide more detail on the STRIPS formalism.

7.2.1 Situation calculus

The *situation calculus* is a full first-order logical language for specifying (and reasoning about) dynamic worlds. The logic has two *sorts*: *actions* and *situations*. An action is a predicate with arguments, such as $put(x, y)$, $walk(loc)$ and $pickup(r, x)$, with the intuitive meaning is that it denotes something that can change the state of the world. A *situation* is a term $do(s, a)$ denoting the state of the world after applying action a in situation s. A situation essentially is a structured term expressing the *history* of actions undertaken. A distinguished constant S_0 denotes the *initial situation* in which no actions have yet been applied. An example of a more elaborated situation is $do(put(A, B), do(put(B, C), S_0)$, which is the situation resulting from putting A on B after putting B on C in the initial situation.

Thus, situation calculus does not explicitly represent the state of the world, but only implicitly by representing the initial state of the world plus all actions that have been applied. Based on a rigorous formalization in terms of *effect axioms*, *precondition axioms* and *successor state axioms* this is all that is needed to reason about anything in the world. The slides and Brachman and Levesque's chapters contain all necessary details on this formalization and how planning can be seen as theorem proving (e.g. using resolution). In addition, it contains a description on how to add imperative constructs (such as if-then-else, while-loops, etc.) to the language to get a powerful logical programming language (Golog, see [11]). Furthermore, the slides contain examples in AILog that add *probabilistic* aspects to the language.

Chapter 8

Lecture 12 – AI Applications of Probabilistic Logic

Computer vision using (probabilistic) logical representations has the promise of being able to use both structured knowledge representation for high-level aspects of a domain (e.g. a house is *next to* another house) and probabilistic aspects for representing the inherent uncertain features of any vision domain. The lecture has covered many examples of how high-level knowledge aspects can be used to *help* in the interpretation of visual data, both for static images as well as dynamic video data. All needed information can be found in the slides. Here we briefly summarize two main approaches.

8.0.2 Vision as constraint satisfaction

As a first example in image interpretation using logic, Levesque describes a simple setting in Levesque's book [12] (http://www.cs.toronto.edu/ hector/tc-instructor.html). in which a picture is described using Prolog rules, and where all constraints on the possible values for parts of the image are added as Prolog rules. This way, knowledge representation is used to specify all constraints and knowledge about a domain, and the interpretation process consists of solving a *constraint satisfaction problem* (i.e. what are possible values for variables which stand for parts of the image?). The slides contain the Prolog code for this simple setting. It depicts in a simple program with which we can ask Prolog for an instantiation of

```
solution(R1,R2,R3,R4,R5)
```

such that, for example, `R1` = `water`, `R2` = `grass` and so on.

8.0.3 Vision using probabilistic explanations

Our last AI example of using KR and probability stems from the field of *depiction* and *image interpretation*. In his article on *probabilistic Horn abduction* [17], which you can find on the course webpage, Poole describes an application of probabilistic logic to the task of interpreting simple images. The key issue is to see pictures as being made up of **image objects** which could have been *caused* by **scene objects**. These scene objects contribute to the *meaning* (or: semantics) of the picture.

Following earlier work by Reiter and Mackworth, Poole assumes that for each image object I there is a corresponding scene object $\sigma(I)$ which it depicts. As an example, let us assume that images can contain *regions*, which can be *explained* using a scene object called *area*. Furthermore, let us assume that we have can have different types of areas that can be an explanation for seeing a region in the image. Here we consider only two different kinds of areas: *land* and *water*. Additional information (for example obtained by counting in all images in a certain domain) teaches us that 70 percent of all areas are water, and only 30 percent are land. We can model this as follows:

$$\mathit{region}(I) \leftarrow \mathit{area}(\sigma(I), T).$$

$$\mathit{disjoint}([\mathit{area}(S, \mathit{land}) : 0.3, \mathit{area}(S, \mathit{water}) : 0.7]).$$

In AILog we can model the second rule using the `prob` statement. Poole introduces several fragments of code to capture the whole image domain, see the paper and the slides.

An image is now one observation, which is actually a conjunction over all image objects (represented as ground atoms), for example $\texttt{chain}(\texttt{c1}) \wedge \texttt{chain}(\texttt{c2}) \wedge \ldots \wedge \texttt{bounds}(\texttt{c1}, \texttt{r2}) \ldots \wedge \texttt{tee}(\texttt{c3}, \texttt{c2}, \texttt{1}) \wedge \ldots$. In essence, we *condition* on this information. Interpreting an image (observation) now amounts to finding a (probabilistic-logical) explanation in terms of scene objects. Poole lists four possible explanations for the example image and shows that using the computed probabilities one can also compute conditional probabilities such as $P(\mathit{linear}(\sigma(c2), \mathit{river}) | \mathit{image})$.

Bibliography

[1] H.J. Levesque and R.J. Brachman. A fundamental tradeoff in knowledge representation and reasoning. In: *Readings in Knowledge Representation*, Morgan-Kaufmann, San Mateo, 1985.

[2] F. van Harmelen, V. Lifschitz, and B. Poter, *Handbook of Knowledge Representation*, Elsevier, Amsterdam, 2008.

[3] W. Hamscher, L. Console, and J. de Kleer, *Readings in Model-based Diagnosis*, Morgan Kaufmann, San Mateo, CA, 1992.

[4] J. de Kleer, A.K. Mackworth and R. Reiter (1992). Characterizing diagnoses and systems. *Artificial Intelligence*, **52**, 197–222.

[5] Y. Peng and J.A. Reggia (1990). *Abductive inference models for diagnostic problem solving*. New York: Springer-Verlag.

[6] D. Poole, R. Goebel and R. Aleliunas (1987). Theorist: a logical reasoning system for defaults and diagnosis. In *The Knowledge Frontier* (N. Cercone and G. Mc Calla, eds.), 331–352. Berlin: Springer-Verlag.

[7] R. Reiter (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, **32**, 57–95.

[8] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice Hall, 2003.

[9] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, 3rd edition, Prentice Hall, 2010.

[10] I. Bratko. *Prolog Programming for Artificial Intelligence*, 3rd ed. Addison-Wesley, Harlow, 2001.

[11] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming, Special issue on Reasoning about Action and Change*, 31:59–84, 1997.

[12] H. Levesque. *Thinking as Computation: A First Course* MIT Press, 2012

[13] J. Lloyd. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, Berlin, 1987.

[14] P.J.F. Lucas and L.C. van der Gaag. *Principles of Expert Systems*, Addison-Wesley, Wokingham, 1991.

[15] D. Poole. Artificial Intelligence: Foundations of Computational Agents, Cambridge University Press, 2010.

[16] D. Barber. Bayesian Reasoning and Machine Learning, Cambridge University Press, 2011.

[17] D. Poole (1993) *Probabilistic Horn Abduction and Bayesian Networks*, Artificial Intelligence 64, pp 81-129

Appendix A

Logic and Resolution

One of the earliest formalisms for the representation of knowledge is *logic*. The formalism is characterized by a well-defined syntax and semantics, and provides a number of *inference rules* to manipulate logical formulas on the basis of their form in order to derive new knowledge. Logic has a very long and rich tradition, going back to the ancient Greeks: its roots can be traced to Aristotle. However, it took until the present century before the mathematical foundations of modern logic were laid, amongst others by T. Skolem, J. Herbrand, K. Gödel, and G. Gentzen. The work of these great and influential mathematicians rendered logic firmly established before the area of computer science came into being.

Already from the early 1950s, as soon as the first digital computers became available, research was initiated on using logic for problem solving by means of the computer. This research was undertaken from different points of view. Several researchers were primarily interested in the mechanization of mathematical proofs: the efficient automated generation of such proofs was their main objective. One of them was M. Davis who, already in 1954, developed a computer program which was capable of proving several theorems from number theory. The greatest triumph of the program was its proof that the sum of two even numbers is even. Other researchers, however, were more interested in the study of human problem solving, more in particular in heuristics. For these researchers, mathematical reasoning served as a point of departure for the study of heuristics, and logic seemed to capture the essence of mathematics; they used logic merely as a convenient language for the formal representation of human reasoning. The classical example of this approach to the area of theorem proving is a program developed by A. Newell, J.C. Shaw and H.A. Simon in 1955, called the *Logic Theory Machine*. This program was capable of proving several theorems from the Principia Mathematica of A.N. Whitehead and B. Russell. As early as 1961, J. McCarthy, amongst others, pointed out that theorem proving could also be used for solving non-mathematical problems. This idea was elaborated by many authors. Well known is the early work on so-called *question-answering systems* by J.R. Slagle and the later work in this field by C.C. Green and B. Raphael.

After some initial success, it soon became apparent that the inference rules known at that time were not as suitable for application in digital computers as hoped for. Many AI researchers lost interest in applying logic, and shifted their attention towards the development of other formalisms for a more efficient representation and manipulation of information. The breakthrough came thanks to the development of an efficient and flexible inference rule in 1965, named *resolution*, that allowed applying logic for automated problem solving by the

computer, and theorem proving finally gained an established position in artificial intelligence and, more recently, in the computer science as a whole as well.

Logic can directly be used as a knowledge-representation formalism for building knowledge systems; currently however, this is done only on a small scale. But then, the clear semantics of logic makes the formalism eminently suitable as a point of departure for understanding what the other knowledge-representation formalisms are all about. In this chapter, we first discuss the subject of how knowledge can be represented in logic, departing from propositional logic, which although having a rather limited expressiveness, is very useful for introducing several important notions. First-order predicate logic, which offers a much richer language for knowledge representation, is treated in Section A.2. The major part of this chapter however will be devoted to the algorithmic aspects of applying logic in an automated reasoning system, and resolution in particular will be the subject of study.

A.1 Propositional logic

Propositional logic may be viewed as a representation language which allows us to express and reason with statements that are either *true* or *false*. Examples of such statements are:

'A full-adder is a logical circuit'
'10 is greater than 90'

Clearly, such statement need not be true. Statements like these are called *propositions* and are usually denoted in propositional logic by uppercase letters. Simple propositions such as P and Q are called *atomic propositions* or *atoms* for short. Atoms can be combined with so-called *logical connectives* to yield *composite propositions*. In the language of propositional logic, we have the following five connectives at our disposal:

negation:	$\neg$	(not)
conjunction:	$\wedge$	(and)
disjunction:	$\vee$	(or)
implication:	$\rightarrow$	(if then)
bi-implication:	$\leftrightarrow$	(if and only if)

For example, when we assume that the propositions G and D have the following meaning

G = 'A Bugatti is a car'
D = 'A Bugatti has 5 wheels'

then the composite proposition

$$G \wedge D$$

has the meaning:

'A Bugatti is a car *and* a Bugatti has 5 wheels'

However, not all formulas consisting of atoms and connectives are (composite) propositions. In order to distinguish syntactically correct formulas that do represent propositions from those that do not, the notion of a well-formed formula is introduced in the following definition.

Definition A.1 *A* well-formed formula *in propositional logic is an expression having one of the following forms:*

(1) An atom is a well-formed formula.

(2) If F is a well-formed formula, then $(\neg F)$ is a well-formed formula.

(3) If F and G are well-formed formulas, then $(F \wedge G)$, $(F \vee G)$, $(F \rightarrow G)$ and $(F \leftrightarrow G)$ are well-formed formulas.

(4) No other formula is well-formed.

EXAMPLE A.1

Both formulas $(F \wedge (G \rightarrow H))$ and $(F \vee (\neg G))$ are well-formed according to the previous definition, but the formula $(\rightarrow H)$ is not.

In well-formed formulas, parentheses may be omitted as long as no ambiguity can occur; the adopted priority of the connectives is, in decreasing order, as follows:

$\neg \wedge \vee \rightarrow \leftrightarrow$

In the following, the term formula is used as an abbreviation when a well-formed formula is meant.

EXAMPLE A.2

The formula $P \rightarrow Q \wedge R$ is the same as the formula $(P \rightarrow (Q \wedge R))$.

The notion of well-formedness of formulas only concerns the syntax of formulas in propositional logic: it does not express the formulas to be either *true* or *false*. In other words, it tells us nothing with respect to the semantics or meaning of formulas in propositional logic. The truth or falsity of a formula is called its *truth value*. The meaning of a formula in propositional logic is defined by means of a function $w : \text{PROP} \rightarrow \{true, false\}$ which assigns to each proposition in the set of propositions PROP either the truth value *true* or *false*. Consequently, the information that the atom P has the truth value *true*, is now denoted by $w(P) = true$, and the information that the atom P has the truth value *false*, is denoted by $w(P) = false$. Such a function w is called an interpretation function, or an *interpretation* for short, if it satisfies the following properties (we assume F and G to be arbitrary well-formed formulas):

(1) $w(\neg F) = true$ if $w(F) = false$, and $w(\neg F) = false$ if $w(F) = true$.

(2) $w(F \wedge G) = true$ if $w(F) = true$ and $w(G) = true$; otherwise $w(F \wedge G) = false$.

(3) $w(F \vee G) = false$ if $w(F) = false$ and $w(G) = false$; in all other cases, that is, if at least one of the function values $w(F)$ and $w(G)$ equals *true*, we have $w(F \vee G) = true$.

(4) $w(F \rightarrow G) = false$ if $w(F) = true$ and $w(G) = false$; in all other cases we have $w(F \rightarrow G) = true$.

(5) $w(F \leftrightarrow G) = true$ if $w(F) = w(G)$; otherwise $w(F \leftrightarrow G) = false$.

Table A.1: The meanings of the connectives.

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
-*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

Table A.2: Truth table for $P \rightarrow (\neg Q \wedge R)$.

P	Q	R	$\neg Q$	$\neg Q \wedge R$	$P \rightarrow (\neg Q \wedge R)$
true	*true*	*true*	*false*	*false*	*false*
true	*true*	*false*	*false*	*false*	*false*
true	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*false*	*true*
false	*true*	*false*	*false*	*false*	*true*
false	*false*	*true*	*true*	*true*	*true*
false	*false*	*false*	*true*	*false*	*true*

These rules are summarized in Table A.1. The first two columns in this table list all possible combinations of truth values for the atomic propositions F and G; the remaining columns define the meanings of the respective connectives. If w is an interpretation which assigns to a given formula F the truth value *true*, then w is called a *model* for F.

By repeated applications of the rules listed in table 2.1, it is possible to express the truth value of an arbitrary formula in terms of the truth values of the atoms the formula is composed of. In a formula containing n different atoms, there are 2^n possible ways of assigning truth values to the atoms in the formula.

EXAMPLE A.3

Table A.2 lists all possible combinations of truth values for the atoms in the formula $P \rightarrow (\neg Q \wedge R)$; for each combination, the resulting truth value for this formula is determined. Such a table where all possible truth values for the atoms in a formula F are entered together with the corresponding truth value for the whole formula F, is called a *truth table*.

Definition A.2 *A formula is called a* valid *formula if it is true under all interpretations. A valid formula is often called a* tautology*. A formula is called* invalid *if it is not valid.*

So, a valid formula is true regardless of the truth or falsity of its constituent atoms.

EXAMPLE A.4

The formula $((P \rightarrow Q) \wedge P) \rightarrow Q$ is an example of a valid formula. In the previous example we dealt with an invalid formula.

Definition A.3 *A formula is called* unsatisfiable *or* inconsistent *if the formula is false under all interpretations. An unsatisfiable formula is also called a* contradiction*. A formula is called* satisfiable *or* consistent *if it is not unsatisfiable.*

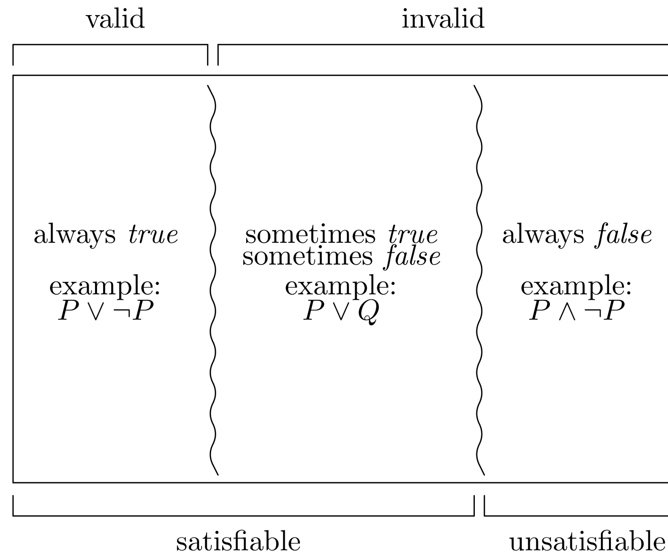

Figure A.1: Relationship between validity and satisfiability.

Table A.3: Truth table of $\neg(P \land Q)$ and $\neg P \lor \neg Q$.

P	Q	$\neg(P \land Q)$	$\neg P \lor \neg Q$
true	*true*	*false*	*false*
true	*false*	*true*	*true*
false	*true*	*true*	*true*
false	*false*	*true*	*true*

Note that a formula is valid precisely when its negation is unsatisfiable and vice versa.

EXAMPLE A.5

The formulas $P \land \neg P$ and $(P \to Q) \land (P \land \neg Q)$ are both unsatisfiable.

Figure A.1 depicts the relationships between the notions of valid, invalid, and satisfiable, and unsatisfiable formulas.

Definition A.4 *Two formulas F and G are called* equivalent*, written as $F \equiv G$, if the truth values of F and G are the same under all possible interpretations.*

Two formulas can be shown to be equivalent by demonstrating that their truth tables are identical.

EXAMPLE A.6

Table A.3 shows that $\neg(P \land Q) \equiv \neg P \lor \neg Q$.

Using truth tables the logical equivalences listed in Table 2.4 can easily be proven. These equivalences are called *laws of equivalence*. Law (a) is called the *law of double negation*; the laws (b) and (c) are called the *commutative laws*; (d) and (e) are the so-called *associative*

Table A.4: Laws of equivalence.

$\neg(\neg F) \equiv F$	(a)
$F \vee G \equiv G \vee F$	(b)
$F \wedge G \equiv G \wedge F$	(c)
$(F \wedge G) \wedge H \equiv F \wedge (G \wedge H)$	(d)
$(F \vee G) \vee H \equiv F \vee (G \vee H)$	(e)
$F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$	(f)
$F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$	(g)
$F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$	(h)
$F \rightarrow G \equiv \neg F \vee G$	(i)
$\neg(F \wedge G) \equiv \neg F \vee \neg G$	(j)
$\neg(F \vee G) \equiv \neg F \wedge \neg G$	(k)

laws, and (f) and (g) are the *distributive laws*. The laws (j) and (k) are known as the *laws of De Morgan*. These laws often are used to transform a given well-formed formula into a logically equivalent but syntactically different formula.

In the following, a conjunction of formulas is often written as a set of formulas, where the elements of the set are taken as the conjunctive subformulas of the given formula.

EXAMPLE A.7

The set $S = \{F \vee G, H\}$ represents the following formula: $(F \vee G) \wedge H$.

Truth tables can be applied to determine whether or not a given formula follows logically from a given set of formulas. Informally speaking, a formula logically follows from a set of formulas if it is satisfied by all interpretations satisfying the given set of formulas; we say that the formula is a logical consequence of the formulas in the given set. The following is a formal definition of this notion.

Definition A.5 *A formula G is said to be a* logical consequence *of the set of formulas $F = \{F_1, \ldots, F_n\}$, $n \geq 1$, denoted by $F \models G$, if for each interpretation w for which $w(F_1 \wedge \cdots \wedge F_n) = true$, we have $w(G) = true$.*

EXAMPLE A.8

The formula R is a logical consequence of the set of formulas $\{P \wedge \neg Q, P \rightarrow R\}$. Thus we can write $\{P \wedge \neg Q, P \rightarrow R\} \models R$.

Note that another way of stating that two formulas F and G are logically equivalent, that is, $F \equiv G$, is to say that both $\{F\} \models G$ and $\{G\} \models F$ hold. This tells us that the truth value of F and G are explicitly related to each other, which can also be expressed as $\models (F \leftrightarrow G)$.

Satisfiability, validity, equivalence and logical consequence are *semantic* notions; these properties are generally established using truth tables. However, for deriving logical consequences from of a set of formulas for example, propositional logic provides other techniques than using truth tables as well. It is possible to derive logical consequences by *syntactic* operations only. A formula which is derived from a given set of formulas then is guaranteed

to be a logical consequence of that set if the syntactic operations employed meet certain conditions. Systems in which such syntactic operations are defined, are called (*formal*) *deduction systems*. Various sorts of deduction systems are known. An example of a deduction system is an *axiomatic system*, consisting of a formal language, such as the language of propositional logic described above, a set of *inference rules* (the syntactic operations) and a set of *axioms*. In Section 2.4 we shall return to the subject of logical deduction.

A.2 First-order predicate logic

In propositional logic, atoms are the basic constituents of formulas which are either *true* or *false*. A limitation of propositional logic is the impossibility to express general statements concerning similar cases. *First-order predicate logic* is more expressive than propositional logic, and such general statements can be specified in its language. Let us first introduce the language of first-order predicate logic. The following symbols are used:

- *Predicate symbols*, usually denoted by uppercase letters. Each predicate symbol has associated a natural number n, $n \geq 0$, indicating the number of arguments the predicate symbol has; the predicate symbol is called an *n-place* predicate symbol. 0-place or *nullary* predicate symbols are also called (*atomic*) *propositions*. One-place, two-place and three-place predicate symbols are also called *unary*, *binary* and *ternary* predicate symbols, respectively.

- *Variables*, usually denoted by lowercase letters from the end of the alphabet, such as x, y, z, possibly indexed with a natural number.

- *Function symbols*, usually denoted by lowercase letters halfway the alphabet. Each function symbol has associated a natural number n, $n \geq 0$, indicating its number of arguments; the function symbol is called *n-place*. Nullary function symbols are usually called *constants*.

- The *logical connectives* which have already been discussed in the previous section.

- Two *quantifiers*: the *universal quantifier* $\forall$, and the *existential quantifier* $\exists$. The quantifiers should be read as follows: if x is a variable, then $\forall x$ means 'for each x' or 'for all x', and $\exists x$ means 'there exists an x'.

- A number of *auxiliary symbols* such as parentheses and commas.

Variables and functions in logic are more or less similar to variables and functions in for instance algebra or calculus.

Before we define the notion of an atomic formula in predicate logic, we first introduce the notion of a term.

Definition A.6 *A* term *is defined as follows:*

(1) A constant is a term.

(2) A variable is a term.

(3) If f is an n-place function symbol, $n \geq 1$, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

(4) Nothing else is a term.

So, a term is either a constant, a variable or a function of terms. Recall that a constant may also be viewed as a nullary function symbol. An atomic formula now consists of a predicate symbol and a number of terms to be taken as the arguments of the predicate symbol.

Definition A.7 *An* atomic formula*, or* atom *for short, is an expression of the form* $P(t_1, \ldots, t_n)$*, where* P *is an* n*-place predicate symbol,* $n \geq 0$*, and* $t_1, \ldots, t_n$ *are terms.*

EXAMPLE A.9

If P is a unary predicate symbol and x is a variable, then $P(x)$ is an atom. $Q(f(y), c, g(f(x), z))$ is an atom if Q is a ternary predicate symbol, c is a constant, f a unary function symbol, g a binary function symbol, and x, y and z are variables. For the same predicate symbols P and Q, $P(Q)$ is not an atom, because Q is not a term but a predicate symbol.

Composite formulas can be formed using the five connectives given in Section 2.1, together with the two quantifiers $\forall$ and $\exists$ just introduced. As was done for propositional logic, we now define the notion of a well-formed formula in predicate logic. The following definition also introduces the additional notions of free and bound variables.

Definition A.8 *A* well-formed formula *in predicate logic, and the set of* free variables *of a well-formed formula are defined as follows:*

(1) An atom is a well-formed formula. The set of free variables of an atomic formula consists of all the variables occurring in the terms in the atom.

(2) Let F *be a well-formed formula with an associated set of free variables. Then,* $(\neg F)$ *is a well-formed formula. The set of free variables of* $(\neg F)$ *equals the set of free variables of* F*.*

(3) Let F *and* G *be well-formed formulas and let for each of these formulas a set of free variables be given. Then,* $(F \vee G)$*,* $(F \wedge G)$*,* $(F \rightarrow G)$ *and* $(F \leftrightarrow G)$ *are well-formed formulas. The set of free variables of each of these last mentioned formulas is equal to the union of the sets of free variables of* F *and* G*.*

(4) If F *is well-formed formula and* x *is an element of the set of free variables of* F*, then both* $(\forall x F)$ *and* $(\exists x F)$ *are well-formed formulas. The set of free variables of each of these formulas is equal to the set of free variables of* F *from which the variable* x *has been removed. The variable* x *is called* bound *by the quantifier* $\forall$ *or* $\exists$*.*

(5) Nothing else is a well-formed formula.

Note that we have introduced the notion of a formula in the preceding definition only from a purely syntactical point of view: nothing has been said about the meaning of such a formula.

Parentheses will be omitted from well-formed formulas as long as ambiguity cannot occur; the quantifiers then have a higher priority than the connectives.

Definition A.9 *A well-formed formula is called a* closed formula*, or a* sentence*, if its set of free variables is empty; otherwise it is called an* open formula*.*

EXAMPLE A.10

The set of free variables of the formula $\forall x \exists y(P(x) \rightarrow Q(y,z))$ is equal to $\{z\}$. So, only one of the three variables in the formula is a free variable. The formula $\forall x(P(x) \vee R(x))$ has no free variables at all, and thus is an example of a sentence.

In what follows, we shall primarily be concerned with closed formulas; the term formula will be used to mean a closed formula, unless explicitly stated otherwise.

In the formula $\forall x(A(x) \rightarrow G(x))$ all occurrences of the variable x in $A(x) \rightarrow G(x)$ are governed by the associated universal quantifier; $A(x) \rightarrow G(x)$ is called the *scope* of this quantifier.

EXAMPLE A.11

The scope of the universal quantifier in the formula

$$\forall x(P(x) \rightarrow \exists y R(x,y))$$

is $P(x) \rightarrow \exists y R(x,y)$; the scope of the existential quantifier is the subformula $R(x,y)$.

In propositional logic, the truth value of a formula under a given interpretation is obtained by assigning either the truth value *true* or *false* to each of its constituent atoms according to this specific interpretation. Defining the semantics of first-order predicate logic is somewhat more involved than in propositional logic. In predicate logic, a structure representing the 'reality' is associated with the *meaningless* set of symbolic formulas: in a structure the objects or elements of the domain of discourse, or domain for short, are enlisted, together with functions and relations defined on the domain.

Definition A.10 *A* structure *S is a tuple*

$$S = (D, \{\bar{f}_i^n : D^n \rightarrow D, n \geq 1\}, \{\bar{P}_i^m : D^m \rightarrow \{true, false\}, m \geq 0\})$$

having the following components:

(1) A non-empty set of elements D, called the domain *of S;*

(2) A set of functions *defined on D^n, $\{\bar{f}_i^n : D^n \rightarrow D, n \geq 1\}$;*

(3) A non-empty set of mappings, called predicates, *from D^m to the set of truth values $\{true, false\}$, $\{\bar{P}_i^m : D^m \rightarrow \{true, false\}, m \geq 0\}$.*

The basic idea underlying the definition of a structure is that we associate functions $\bar{f}_i^n$ to function symbols f_i and predicates $\bar{P}_i^m$ to predicate symbols P_i. Hence, we have to express how a given meaningless formula should be interpreted in a given structure: it is not possible to state anything about the truth value of a formula as long as it has not been prescribed which elements from the structure are to be associated with the elements in the formula.

EXAMPLE A.12

Consider the formula $A(c)$. We associate the predicate having the intended meaning 'is a car' with the predicate symbol A. The formula should be *true* if the constant representing a Bugatti is associated with c; on the other hand, the same formula should be *false* if the constant representing a Volvo truck is associated with c. However, if we associate the predicate 'Truck' with A, the truth values of $A(c)$ for the two constants should be opposite to the ones mentioned before.

In the following definition, we introduce the notion of an assignment, which is a function that assigns elements from the domain of a structure to the variables in a formula.

Definition A.11 *An* assignment *(*valuation*) v to a set of formulas F in a given structure S with domain D is a mapping from the set of variables in F to D.*

The interpretation of (terms and) formulas in a structure S under an assignment v now consists of the following steps. First, the constants in the formulas are assigned elements from D. Secondly, the variables are replaced by the particular elements from D that have been assigned to them by v. Then, the predicate and function symbols occurring in the formulas are assigned predicates and functions from S. Finally, the truth values of the formulas are determined.

Before the notion of an interpretation is defined more formally, a simple example in which no function symbols occur, is given. For the reader who is not interested in the formal aspects of logic, it suffices to merely study this example.

EXAMPLE A.13

The open formula

$$F = A(x) \to O(x)$$

contains the unary predicate symbols A and O, and the free variable x. Consider the structure S consisting of the domain $D = \{bugatti, volvo\text{-}truck, alfa\text{-}romeo\}$ and the set of predicates comprising of the following elements:

- a unary predicate *Car*, with the intented meaning 'is a car', defined by $Car(bugatti) = true$, $Car(alfa\text{-}romeo) = true$ and $Car(volvo\text{-}truck) = false$, and
- the unary predicate *FourWheels* with the intended meaning 'has four wheels', defined by $FourWheels(bugatti) = false$, $FourWheels(volvo\text{-}truck) = false$ and $FourWheels(alfa\text{-}romeo) = true$.

Let us take for the predicate symbol A the predicate *Car*, and for the predicate symbol O the predicate *FourWheels*. It will be obvious that the atom $A(x)$ is *true* in S under any assignment v for which $Car(v(x)) = true$; so, for example for the assignment $v(x) = alfa\text{-}romeo$, we have that $A(x)$ is *true* in S under v. Furthermore, F is *true* in the structure S under the assignment v with $v(x) = alfa\text{-}romeo$, since $A(x)$ and $O(x)$ are both *true* in S under v. On the other hand, F is *false* in the structure S under the assignment v' with $v'(x) = bugatti$, because $Car(bugatti) = true$ and $FourWheels(bugatti) = false$ in S. Now, consider the closed formula

$$F' = \forall x(A(x) \to O(x))$$

and again the structure S. It should be obvious that F' is *false* in S.

Table A.5: Laws of equivalence for quantifiers.

$\neg\exists x P(x) \equiv \forall x \neg P(x)$	(a)
$\neg\forall x P(x) \equiv \exists x \neg P(x)$	(b)
$\forall x(P(x) \wedge Q(x)) \equiv \forall x P(x) \wedge \forall x Q(x)$	(c)
$\exists x(P(x) \vee Q(x)) \equiv \exists x P(x) \vee \exists x Q(x)$	(d)
$\forall x P(x) \equiv \forall y P(y)$	(e)
$\exists x P(x) \equiv \exists y P(y)$	(f)

Definition A.12 *An* interpretation *of terms in a structure $S = (D, \{\bar{f}_i^n\}, \{\bar{P}_i^m\})$ under an assignment v, denoted by I_v^S, is defined as follows:*

(1) $I_v^S(c_i) = d_i$, $d_i \in D$, where c_i is a constant.

(2) $I_v^S(x_i) = v(x_i)$, where x_i is a variable.

(3) $I_v^S(f_i^n(t_1, \ldots, t_n)) = \bar{f}_i^n(I_v^S(t_1), \ldots, I_v^S(t_n))$, where $\bar{f}_i^n$ is a function from S associated with the function symbol f_i^n.

The truth value of a formula in a structure S under an assignment v for a given interpretation I_v^S is obtained as follows:

(1) $I_v^S(P_i^m(t_1, \ldots, t_m)) = \bar{P}_i^m(I_v^S(t_1), \ldots, I_v^S(t_m))$, meaning that an atom $P_i^m(t_1, \ldots, t_m)$ is true in the structure S under the assignment v for the interpretation I_v^S if $\bar{P}_i^m(I_v^S(t_1), \ldots, I_v^S(t_m))$ is true, where $\bar{P}_i^m$ is the predicate from S associated with P_i^m.

(2) If the truth values of the formulas F and G have been determined, then the truth values of $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$ and $F \leftrightarrow G$ are defined by the meanings of the connectives as listed in Table A.1.

(3) $\exists x F$ is true under v if there exists an assignment v' differing from v at most with regard to x, such that F is true under v'.

(4) $\forall x F$ is true under v if for each v' differing from v at most with regard to x, F is true under v'.

The notions valid, invalid, satisfiable, unsatisfiable, logical consequence, equivalence and model have meanings in predicate logic similar to their meanings in propositional logic. In addition to the equivalences listed in Table A.4, predicate logic also has some laws of equivalence for quantifiers, which are given in Table A.5. Note that the properties $\forall x(P(x) \vee Q(x)) \equiv \forall x P(x) \vee \forall x Q(x)$ and $\exists x(P(x) \wedge Q(x)) \equiv \exists x P(x) \wedge \exists x Q(x)$ do *not* hold.

We conclude this subsection with another example.

EXAMPLE A.14

We take the unary (meaningless) predicate symbols C, F, V, W and E, and the constants a and b from a given first-order language. Now, consider the following formulas:

(1) $\forall x(C(x) \rightarrow V(x))$

(2) $F(a)$

(3) $\forall x(F(x) \to C(x))$

(4) $\neg E(a)$

(5) $\forall x((C(x) \land \neg E(x)) \to W(x))$

(6) $F(b)$

(7) $\neg W(b)$

(8) $E(b)$

Consider the structure S in the reality with a domain consisting of the elements and *bugatti*, which are assigned to the constants a and b, respectively. The set of predicates in S comprises the unary predicates *Car*, *Fast*, *Vehicle*, *FourWheels*, and *Exception*, which are taken for the predicate symbols C, F, V, W, and E, respectively. The structure S and the mentioned interpretation have been carefully chosen so as to satisfy the above-given closed formulas, for instance by giving the following intended meaning to the predicates:

Car	=	'is a car'
Fast	=	'is a fast car'
Vehicle	=	'is a vehicle'
FourWheels	=	'has four wheels'
Exception	=	'is an exception'

In the given structure S, the formula numbered 1 expresses the knowledge that every car is a vehicle. The fact that an alfa-romeo is a fast car, has been stated in formula 2. Formula 3 expresses that every fast car is a car, and formula 4 states that an alfa-romeo is not an exception to the rule that cars have four wheels, which has been formalized in logic by means of formula 5. A Bugatti is a fast car (formula 6), but contrary to an alfa-romeo it does not have 4 wheels (formula 7), and therefore is an exception to the last mentioned rule; the fact that Bugattis are exceptions is expressed by means of formula 8.

It should be noted that in another structure with another domain and other predicates, the formulas given above might have completely different meanings.

A.3 Clausal form of logic

Before turning our attention to reasoning in logic, we introduce in this section a syntactically restricted form ofpredicate logic, called the *clausal form of logic*, which will play an important role in the remainder of this chapter. This restricted form however, can be shown to be as expressive as full first-order predicate logic. The clausal form of logic is often employed, in particular in the fields of theorem proving and logic programming.

We start with the definition of some new notions.

Definition A.13 *A* literal *is an atom, called a* positive literal*, or a negation of an atom, called a* negative literal*.*

Definition A.14 *A* clause *is a closed formula of the form*

$$\forall x_1 \cdots \forall x_s (L_1 \vee \cdots \vee L_m)$$

where each L_i*,* $i = 1, \ldots, m$*,* $m \geq 0$*, is a literal, with* $L_i \neq L_j$ *for each* $i \neq j$*, and* $x_1, \ldots, x_s$*,* $s \geq 0$*, are variables occurring in* $L_1 \vee \cdots \vee L_m$*. If* $m = 0$*, the clause is said to be the* empty clause*, denoted by* $\Box$*.*

The empty clause $\Box$ is interpreted as a formula which is always *false*, in other words, $\Box$ is an unsatisfiable formula.

A clause

$$\forall x_1 \cdots \forall x_s (A_1 \vee \cdots \vee A_k \vee \neg B_1 \vee \cdots \vee \neg B_n)$$

where $A_1, \ldots, A_k, B_1, \ldots, B_n$ are atoms and $x_1, \ldots, x_s$ are variables, is equivalent to

$$\forall x_1 \cdots \forall x_s (B_1 \wedge \cdots \wedge B_n \rightarrow A_1 \vee \cdots \vee A_k)$$

as a consequence of the laws $\neg F \vee G \equiv F \rightarrow G$ and $\neg F \vee \neg G \equiv \neg(F \wedge G)$, and is often written as

$$A_1, \ldots, A_k \leftarrow B_1, \ldots, B_n$$

The last notation is the more conventional one in logic programming. The commas in $A_1, \ldots, A_k$ each stand for a disjunction, and the commas in $B_1, \ldots, B_n$ indicate a conjunction. $A_1, \ldots, A_k$ are called the *conclusions* of the clause, and $B_1, \ldots, B_n$ the *conditions*.

Each well-formed formula in first-order predicate logic can be translated into a set of clauses, which is viewed as the conjunction of its elements. As we will see, this translation process may slightly alter the meaning of the formulas. We shall illustrate the translation process by means of an example. Before proceeding, we define two normal forms which are required for the translation process.

Definition A.15 *A formula* F *is in* prenex normal form *if* F *is of the form*

$$Q_1 x_1 \cdots Q_n x_n M$$

where each Q_i*,* $i = 1, \ldots, n$*,* $n \geq 1$*, equals one of the two quantifiers* $\forall$ *and* $\exists$*, and where* M *is a formula in which no quantifiers occur.* $Q_1 x_1 \ldots Q_n x_n$ *is called the* prefix *and* M *is called the* matrix *of the formula* F*.*

Definition A.16 *A formula* F *in prenex normal form is in* conjunctive normal form *if the matrix of* F *is of the form*

$$F_1 \wedge \cdots \wedge F_n$$

where each F_i*,* $i = 1, \ldots, n$*,* $n \geq 1$*, is a disjunction of literals.*

EXAMPLE A.15

Consider the following three formulas:

$\forall x(P(x) \vee \exists y Q(x, y))$
$\forall x \exists y \forall z((P(x) \wedge Q(x, y)) \vee \neg R(z))$
$\forall x \exists y((\neg P(x) \vee Q(x, y)) \wedge (P(y) \vee \neg R(x)))$

The first formula is not in prenex normal form because of the occurrence of an existential quantifier in the 'inside' of the formula. The other two formulas are both in prenex normal form; moreover, the last formula is also in conjunctive normal form.

The next example illustrates the translation of a well-formed formula into a set of clauses. The translation scheme presented in the example however is general and can be applied to any well-formed formula in first-order predicate logic.

EXAMPLE A.16

Consider the following formula:

$$\forall x(\exists y P(x, y) \vee \neg \exists y(\neg Q(x, y) \rightarrow R(f(x, y))))$$

This formula is transformed in eight steps, first into prenex normal form, subsequently into conjunctive normal form, amongst others by applying the laws of equivalence listed in the tables A.4 and A.5, and finally into a set of clauses.

Step 1. Eliminate all implication symbols using the equivalences $F \rightarrow G \equiv \neg F \vee G$ and $\neg(\neg F) \equiv F$:

$$\forall x(\exists y P(x, y) \vee \neg \exists y(Q(x, y) \vee R(f(x, y))))$$

If a formula contains bi-implication symbols, these can be removed by applying the equivalence

$$F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$$

Step 2. Diminish the scope of the negation symbols in such a way that each negation symbol only governs a single atom. This can be accomplished by using the equivalences $\neg \forall x F(x) \equiv \exists x \neg F(x)$, $\neg \exists x F(x) \equiv \forall x \neg F(x)$, $\neg(\neg F) \equiv F$, together with the laws of De Morgan:

$$\forall x(\exists y P(x, y) \vee \forall y(\neg Q(x, y) \wedge \neg R(f(x, y))))$$

Step 3. Rename the variables in the formula using the equivalences $\forall x F(x) \equiv \forall y F(y)$ and $\exists x F(x) \equiv \exists y F(y)$, so that each quantifier has its own uniquely named variable:

$$\forall x(\exists y P(x, y) \vee \forall z(\neg Q(x, z) \wedge \neg R(f(x, z))))$$

Formulas only differing in the names of their bound variables are called *variants*.

Step 4. Eliminate all existential quantifiers. For any existentially quantified variable x not lying within the scope of a universal quantifier, all occurrences of x in the formula within the scope of the existential quantifier can be replaced by a new, that is, not previously used, constant symbol c. The particular existential quantifier may then be removed. For instance, the elimination of the existential quantifier in the formula

$\exists x P(x)$ yields a formula $P(c)$. However, if an existentially quantified variable y lies within the scope of one or more universal quantifiers with the variables $x_1, \ldots, x_n$, $n \geq 1$, then the variable y may be functionally dependent upon $x_1, \ldots, x_n$. Let this dependency be represented explicitly by means of a new n-place function symbol g such that $g(x_1, \ldots, x_n) = y$. All occurrences of y within the scope of the existential quantifier then are replaced by the function term $g(x_1, \ldots, x_n)$, after which the existential quantifier may be removed. The constants and functions introduced in order to allow for the elimination of existential quantifiers are called *Skolem functions*.

The existentially quantified variable y in the example lies within the scope of the universal quantifier with the variable x, and is replaced by $g(x)$:

$$\forall x(P(x, g(x)) \vee \forall z(\neg Q(x, z) \wedge \neg R(f(x, z))))$$

Note that by replacing the existentially quantified variables by Skolem functions, we lose logical equivalence. Fortunately, it can be shown that a formula F is satisfiable if and only if the formula F', obtained from F by replacing existentially quantified variables in F by Skolem functions, is satisfiable as well. In general, the satisfiability of F and F' will not be based on the same model, since F' contains function symbols not occurring in F. In the following, it will become evident that this property is sufficient for our purposes.

Step 5. Transform the formula into prenex normal form, by placing all the universal quantifiers in front of the formula:

$$\forall x \forall z(P(x, g(x)) \vee (\neg Q(x, z) \wedge \neg R(f(x, z))))$$

Note that this is allowed because by step 3 each quantifier applies to a uniquely named variable; this means that the scope of all quantifiers is the entire formula.

Step 6. Bring the matrix in conjunctive normal form using the distributive laws:

$$\forall x \forall z((P(x, g(x)) \vee \neg Q(x, z)) \wedge (P(x, g(x)) \vee \neg R(f(x, z))))$$

Step 7. Select the matrix by disregarding the prefix:

$$(P(x, g(x)) \vee \neg Q(x, z)) \wedge (P(x, g(x)) \vee \neg R(f(x, z)))$$

All variables in the matrix are now implicitly considered to be universally quantified.

Step 8. Translate the matrix into a set of clauses, by replacing formulas of the form $F \wedge G$ by a set of clauses $\{F', G'\}$, where F' and G' indicate that F and G are now represented using the notational convention of logic programming:

$$\{P(x, g(x)) \leftarrow Q(x, z)), P(x, g(x)) \leftarrow R(f(x, z))i\}$$

or the notation used in automated theorem proving:

$$\{P(x, g(x)) \vee \neg Q(x, z),\ P(x, g(x)) \vee \neg R(f(x, z))\}$$

We conclude this subsection with the definition of a special type of clause, a so-called Horn clause, which is a clause containing at most one positive literal.

Definition A.17 *A* Horn clause *is a clause having one of the following forms:*

(1) $A \leftarrow$

(2) $\leftarrow B_1, \ldots, B_n$, $n \geq 1$

(3) $A \leftarrow B_1, \ldots, B_n$, $n \geq 1$

A clause of the form 1 is called a unit clause*; a clause of form 2 is called a* goal clause*.*

Horn clauses are employed in the programming language Prolog. We will return to this observation in Section 2.7.2.

A.4 Reasoning in logic: inference rules

In the Sections 2.1 and 2.2 we described how a meaning could be attached to a meaningless set of logical formulas. This is sometimes called the *declarative semantics* of logic. The declarative semantics offers a means for investigating for example whether or not a given formula is a logical consequence of a set of formulas. However, it is also possible to answer this question without examining the semantic contents of the formulas concerned, by applying so-called *inference rules*. Contrary to truth tables, inference rules are purely syntactic operations which only are capable of modifying the form of the elements of a given set of formulas. Inference rules either add, replace or remove formulas; most inference rules discussed in this book however add new formulas to a given set of formulas. In general, an inference rule is given as a schema in which a kind of meta-variables occur that may be substituted by arbitrary formulas. An example of such a schema is shown below:

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

The formulas above the line are called the *premises*, and the formula below the line is called the *conclusion* of the inference rule. The above-given inference rule is known as *modus ponens*, and when applied, removes an implication from a formula. Another example of an inference rule, in this case for introducing a logical connective, is the following schema:

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

Repeated applications of inference rules give rise to what is called a *derivation* or *deduction*. For instance, modus ponens can be applied to draw the conclusion S from the two formulas $P \wedge (Q \vee R)$ and $P \wedge (Q \vee R) \rightarrow S$. It is said that there exists a derivation of the formula S from the set of clauses $\{P \wedge (Q \vee R), P \wedge (Q \vee R) \rightarrow S\}$. This is denoted by:

$$\{P \wedge (Q \vee R), P \wedge (Q \vee R) \rightarrow S\} \vdash S$$

The symbol $\vdash$ is known as the *turnstile*.

EXAMPLE A.17

Consider the set of formulas $\{P, Q, P \wedge Q \rightarrow S\}$. If the inference rule

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

is applied to the formulas P and Q, the formula $P \wedge Q$ is derived; the subsequent application of modus ponens to $P \wedge Q$ and $P \wedge Q \rightarrow S$ yields S. So,

$$\{P, Q, P \wedge Q \rightarrow S\} \vdash S$$

Now that we have introduced inference rules, it is relevant to investigate how the declarative semantics of a particular class of formulas and its *procedural semantics*, described by means of inference rules, are interrelated: if these two notions are related to each other, we are in the desirable circumstance of being able to assign a meaning to formulas which have been derived using inference rules, simply by our knowledge of the declarative meaning of the original set of formulas. On the other hand, when starting with the known meaning of a set of formulas, it will then be possible to derive only formulas which can be related to that meaning. These two properties are known as the soundness and the completeness, respectively, of a collection of inference rules.

More formally, a collection of inference rules is said to be *sound* if and only if for each formula F derived by applying these inference rules on a given set of well-formed formulas S of a particular class (for example clauses), we have that F is a logical consequence of S. This property can be expressed more tersely as follows, using the notations introduced before:

if $S \vdash F$ then $S \vDash F$.

In other words, a collection of inference rules is sound if it preserves truth under the operations of a derivation. This property is of great importance, because only by applying sound inference rules it is possible to assign a meaning to the result of a derivation.

EXAMPLE A.18

The previously discussed inference rule modus ponens is an example of a sound inference rule. From the given formulas F and $F \rightarrow G$, the formula G can be derived by applying modus ponens, that is, we have $\{F, F \rightarrow G\} \vdash G$. On the other hand, if $F \rightarrow G$ and F are both *true* under a particular interpretation w, then from the truth Table 2.1 we have that G is *true* under w as well. So, G is a logical consequence of the two given formulas: $\{F, F \rightarrow G\} \vDash G$.

The reverse property that by applying a particular collection of inference rules, each logical consequence F of a given set of formulas S can be derived, is called the *completeness* of the collection of inference rules:

if $S \vDash F$ then $S \vdash F$.

EXAMPLE A.19

The collection of inference rules only consisting of modus ponens is not complete for all well-formed formulas in propositional logic. For example, it is not possible to derive the formula P from $\neg Q$ and $P \vee Q$, although P is a logical consequence of the two formulas. However, by combining modus ponens with other inference rules, it is possible to obtain a complete collection of inference rules.

The important question now arises if there exists a mechanical proof procedure, employing a particular sound and complete collection of inference rules, which is capable of determining whether or not a given formula F can be derived from a given set of formulas S. In 1936, A. Church and A.M. Turing showed, independently, that such a general proof procedure does not exist for first-order predicate logic. This property is called the *undecidability* of first-order predicate logic. All known proof procedures are only capable of deriving F from S (that is, are able to prove $S \vdash F$) if F is a logical consequence of S (that is, if $S \models F$); if F is not a logical consequence of S, then the proof procedure is not guaranteed to terminate.

However, for propositional logic there do exist proof procedures which always terminate and yield the right answer: for checking whether a given formula is a logical consequence of a certain set of formulas, we can simply apply truth tables. So, propositional logic is decidable.

The undecidability of first-order predicate logic has not refrained the research area of automated theorem proving from further progress. The major result of this research has been the development of an efficient and flexible inference rule, which is both sound and complete for proving inconsistency, called *resolution*. This is sometimes called *refutation completeness* (see below). However, the resolution rule is only suitable for manipulating formulas in clausal form. Hence, to use this inference rule on a set of arbitrary logical formulas in first-order predicate logic, it is required to translate each formula into the clausal form of logic by means of the procedure discussed in Section 2.3. This implies that resolution is *not* complete for *unrestricted* first-order predicate logic. The formulation of resolution as a suitable inference rule for automated theorem proving in the clausal form of logic has been mainly due to J.A. Robinson, who departed from earlier work by D. Prawitz. The final working-out of resolution in various algorithms, supplemented with specific implementation techniques, has been the work of a large number of researchers. Resolution is the subject of the remainder of this chapter.

A.5 Resolution and propositional logic

We begin this section with a brief, informal sketch of the principles of resolution. Consider a set of formulas S in clausal form. Suppose we are given a formula G, also in clausal form, for which we have to prove that it can be derived from S by applying resolution. Proving $S \vdash G$ is equivalent to proving that the set of clauses W, consisting of the clauses in S supplemented with the negation of the formula G, that is $W = S \cup \{\neg G\}$, is unsatisfiable. Resolution on W now proceeds as follows: first, it is checked whether or not W contains the empty clause $\Box$; if this is the case, then W is unsatisfiable, and G is a logical consequence of S. If the empty clause $\Box$ is not in W, then the resolution rule is applied on a suitable pair of clauses from W, yielding a new clause. Every clause derived this way is added to W, resulting in a new set of clauses on which the same resolution procedure is applied. The entire procedure is repeated until some generated set of clauses has been shown to contain the empty clause $\Box$, indicating unsatisfiability of W, or until all possible new clauses have been derived.

The basic principles of resolution are best illustrated by means of an example from propositional logic. In Section 2.6 we turn our attention to predicate logic.

EXAMPLE A.20

Consider the following set of clauses:

$$\{C_1 = P \vee R, C_2 = \neg P \vee Q\}$$

These clauses contain *complementary* literals, that is, literals having opposite truth values, namely P and $\neg P$. Applying resolution, a new clause C_3 is derived being the disjunction of the original clauses C_1 and C_2 in which the complementary literals have been cancelled out. So, application of resolution yields the clause

$$C_3 = R \vee Q$$

which then is added to the original set of clauses.

The resolution principle is described more precisely in the following definition.

Definition A.18 *Consider the two clauses C_1 and C_2 containing the literals L_1 and L_2 respectively, where L_1 and L_2 are complementary. The procedure of* resolution *proceeds as follows:*

(1) Delete L_1 from C_1 and L_2 from C_2, yielding the clauses C'_1 and C'_2;

(2) Form the disjunction C' of C'_1 and C'_2;

(3) Delete (possibly) redundant literals from C', thus obtaining the clause C.

The resulting clause C is called the resolvent *of C_1 and C_2. The clauses C_1 and C_2 are said to be the* parent clauses *of the resolvent.*

Resolution has the important property that when two given parent clauses are *true* under a given interpretation, their resolvent is *true* under the same interpretation as well: resolution is a sound inference rule. In the following theorem we prove that resolution is sound for the case of propositional logic.

THEOREM A.1 *(*soundness of resolution*) Consider two clauses C_1 and C_2 containing complementary literals. Then, any resolvent C of C_1 and C_2 is a logical consequence of $\{C_1, C_2\}$.*

Proof: We are given that the two clauses C_1 and C_2 contain complementary literals. So, it is possible to write C_1 and C_2 as $C_1 = L \vee C'_1$ and $C_2 = \neg L \vee C'_2$ respectively for some literal L. By definition, a resolvent C is equal to $C'_1 \vee C'_2$ from which possibly redundant literals have been removed. Now, suppose that C_1 and C_2 are both *true* under an interpretation w. We then have to prove that C is *true* under the same interpretation w as well. Clearly, either L or $\neg L$ is *false*. Suppose that L is *false* under w, then C_1 obviously contains more than one literal, since otherwise C_1 would be *false* under w. It follows that C'_1 is *true* under w. Hence, $C'_1 \vee C'_2$, and therefore also C, is *true* under w. Similarly, it can be shown that the resolvent is *true* under w if it is assumed that L is *true*. So, if C_1 and C_2 are *true* under w then C is *true* under w as well. Hence, C is a logical consequence of C_1 and C_2. $\diamondsuit$

Resolution is also a complete inference rule. Proving the completeness of resolution is beyond the scope of this book; we therefore confine ourselves to merely stating the property.

EXAMPLE A.21

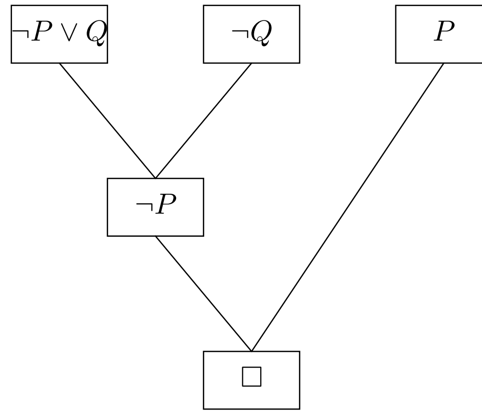

Figure A.2: A refutation tree.

In the definition of a clause in Section 2.3, it was mentioned that a clause was not allowed to contain duplicate literals. This condition appears to be a necessary requirement for the completeness of resolution. For example, consider the following set of formulas:

$$S = \{P \vee P, \neg P \vee \neg P\}$$

It will be evident that S is unsatisfiable, since $P \vee P \equiv P$ and $\neg P \vee \neg P \equiv \neg P$. However, if resolution is applied to S then in every step the tautology $P \vee \neg P$ is derived. It is not possible to derive the empty clause $\Box$.

Until now we have used the notion of a derivation only in an intuitive sense. Before giving some more examples, we define the notion of a derivation in a formal way.

Definition A.19 *Let S be a set of clauses and let C be a single clause. A* derivation *of C from S, denoted by $S \vdash_{\mathcal{B}} C$, is a finite sequence of clauses $C_1, C_2, \ldots, C_n$, $n \geq 1$, where each C_k either is a clause in S or a resolvent with parent clauses C_i and C_j, $i < k$, $j < k$, $i \neq j$, from the sequence, and $C = C_n$. If $C_n = \Box$, then the derivation is said to be a* refutation *of S, indicating that S is unsatisfiable.*

EXAMPLE A.22

Consider the following set of clauses:

$$S = \{\neg P \vee Q, \neg Q, P\}$$

From $C_1 = \neg P \vee Q$ and $C_2 = \neg Q$ we obtain the resolvent $C_3 = \neg P$. From the clauses C_3 and $C_4 = P$ we derive $C_5 = \Box$. So, S is unsatisfiable. The sequence of clauses C_1, C_2, C_3, C_4, C_5 is a refutation of S. Note that it is not the only possible refutation of S. In general, a set S of clauses may have more than one refutation.

Notice that by the choice of the empty clause $\Box$ as a formula that is *false* under all interpretations, which is a *semantic* notion, the *proof-theoretical* notion of a refutation has obtained a suitable meaning. A derivation can be depicted in a graph, called a *derivation graph*. In the case of a refutation, the vertices in the derivation graph may be restricted to those clauses and

resolvents which directly or indirectly contribute to the refutation. Such a derivation graph has the form of a tree and is usually called a *refutation tree*. The leaves of such a tree are clauses from the original set, and the root of the tree is the empty clause $\square$. The refutation tree for the derivation discussed in the previous example is shown in Figure A.2. Note that another refutation of S gives rise to another refutation tree.

A.6 Resolution and first-order predicate logic

An important feature of resolution in first-order predicate logic, taking place in the basic resolution method, is the manipulation of terms. This has not been dealt with in the previous section, where we only had atomic propositions, connectives and auxiliary symbols as building blocks for propositional formulas. In this section, we therefore first discuss the manipulation of terms, before we provide a detailed description of resolution in first-order predicate logic.

A.6.1 Substitution and unification

The substitution of terms for variables in formulas in order to make these formulas syntactically equal, plays a central role in a method known as *unification*. We first introduce the notion of substitution formally and then discuss its role in unification.

Definition A.20 *A* substitution σ *is a finite set of the form*

$$\{t_1/x_1, \ldots, t_n/x_n\}$$

where each x_i *is a variable and where each* t_i *is a term not equal to* x_i*,* $i = 1, \ldots, n$*,* $n \geq 0$*; the variables* $x_1, \ldots, x_n$ *differ from each other. An element* t_i/x_i *of a substitution* σ *is called a* binding *for the variable* x_i*. If none of the terms* t_i *in a substitution contains a variable, we have a so-called* ground substitution*. The substitution defined by the empty set is called the* empty substitution*, and is denoted by* ϵ*.*

Definition A.21 *An* expression *is a term, a literal, a conjunction of literals or a disjunction of literals; a* simple expression *is a term or an atom.*

A substitution σ can be applied to an expression E, yielding a new expression $E\sigma$ which is similar to E with the difference that the variables in E occurring in σ have been replaced by their associated terms.

Definition A.22 *Let* $\sigma = \{t_1/x_1, \ldots, t_n/x_n\}$*,* $n \geq 0$*, be a substitution and* E *an expression. Then,* $E\sigma$ *is an expression obtained from* E *by simultaneously replacing all occurrences of the variables* x_i *by the terms* t_i*.* $E\sigma$ *is called an* instance *of* E*. If* $E\sigma$ *does not contain any variables, then* $E\sigma$ *is said to be a* ground instance *of* E*.*

EXAMPLE A.23

Let $\sigma = \{a/x, w/z\}$ be a substitution and let $E = P(f(x, y), z)$ be an expression. Then, $E\sigma$ is obtained by replacing each variable x in E by the constant a and each variable z by the variable w. The result of the substitution is $E\sigma = P(f(a, y), w)$. Note that $E\sigma$ is not a ground instance.

The application of a substitution to a single expression can be extended to a set of expressions, as demonstrated in the following example.

EXAMPLE A.24

Application of the substitution $\sigma = \{a/x, b/z\}$ to the set of expressions $\{P(x, f(x, z)), Q(x, w)\}$ yields the following set of instances:

$$\{P(x, f(x, z)), Q(x, w)\}\sigma = \{P(a, f(a, b)), Q(a, w)\}$$

The first element of the resulting set of instances is a ground instance; the second one is not ground, since it contains the variable w.

Definition A.23 *Let* $\theta = \{t_1/x_1, \ldots, t_m/x_m\}$ *and* $\sigma = \{s_1/y_1, \ldots, s_n/y_n\}$, $m \geq 1$, $n \geq 1$, *be substitutions. The* composition *of these substitutions, denoted by* $\theta\sigma$, *is obtained by removing from the set*

$$\{t_1\sigma/x_1, \ldots, t_m\sigma/x_m, s_1/y_1, \ldots, s_n/y_n\}$$

all elements $t_i\sigma/x_i$ *for which* $x_i = t_i\sigma$, *and furthermore, all elements* s_j/y_j *for which* $y_j \in \{x_1, \ldots, x_m\}$

The composition of subtitutions is *associative*, i.e., for any expression E and substitutions ϕ, θ and σ we have that $E(\phi\sigma)\theta = E\phi(\sigma\theta)$; the operation is not commutative. Let $\theta = \{t_1/x_1, \ldots, t_m/x_m\}$ be a substitution, and let V be the set of variables occurring in $\{t_1, \ldots, t_m\}$, then θ is *idempotent*, i.e., $E(\theta\theta) = E\theta$, iff $V \cap \{x_1, \ldots, x_m\} = \varnothing$.

Note that the last definition gives us a means for replacing two substitutions by a single one, being the composition of these substitutions. However, it is not always necessary to actually compute the composition of two subsequent substitutions σ and θ before applying them to an expression E: it can easily be proven that $E(\sigma\theta) = (E\sigma)\theta$. The proof of this property is left to the reader as an exercise (see Exercise 2.11); here, we merely give an example.

EXAMPLE A.25

Consider the expression $E = Q(x, f(y), g(z, x))$ and the two substitutions $\sigma = \{f(y)/x, z/y\}$ and $\theta = \{a/x, b/y, y/z\}$. We compute the composition $\sigma\theta$ of σ and θ: $\sigma\theta = \{f(b)/x, y/z\}$. Application of the compound substitution $\sigma\theta$ to E yields the instance $E(\sigma\theta) = Q(f(b), f(y), g(y, f(b)))$. We now compare this instance with $(E\sigma)\theta$. We first apply σ to E, resulting in $E\sigma = Q(f(y), f(z), g(z, f(y)))$. Subsequently, we apply θ to $E\sigma$ and obtain the instance $(E\sigma)\theta = Q(f(b), f(y), g(y, f(b)))$. So, for the given expression and substitutions, we have $E(\sigma\theta) = (E\sigma)\theta$.

In propositional logic, a resolvent of two parent clauses containing complementary literals, such as P and $\neg P$, was obtained by taking the disjunction of these clauses after cancelling out such a pair of complementary literals. It was easy to check for complementary literals in this case, since we only had to verify equality of the propositional atoms in the chosen

literals and the presence of a negation in exactly one of them. Now, suppose that we want to compare the two literals $\neg P(x)$ and $P(a)$ occurring in two different clauses in first-order predicate logic. These two literals are 'almost' complementary. However, the first literal contains a variable as an argument of its predicate symbol, whereas the second one contains a constant. It is here where substitution comes in. Note that substitution can be applied to make expressions syntactically equal. Moreover, the substitution which is required to obtain syntactic equality of two given expressions also indicates the difference between the two. If we apply the substitution $\{a/x\}$ to the example above, we obtain syntactic equality of the two atoms $P(x)$ and $P(a)$. So, the two literals $\neg P(x)$ and $P(a)$ become complementary after substitution.

The *unification algorithm* is a general method for comparing expressions; the algorithm computes, if possible, the substitution that is needed to make the given expressions syntactically equal. Before we discuss the algorithm, we introduce some new notions.

Definition A.24 *A substitution σ is called a* unifier *of a given set of expressions $\{E_1, \ldots, E_m\}$ if $E_1\sigma = \cdots = E_m\sigma, m \geq 2$. A set of expressions is called* unifiable *if it has a unifier.*

Definition A.25 *A unifier θ of a unifiable set of expressions $E = \{E_1, \ldots, E_m\}$, $m \geq 2$, is said to be a* most general unifier *(mgu) if for each unifier σ of E there exists a substitution λ such that $\sigma = \theta\lambda$.*

A set of expressions may have more than one most general unifier; however, a most general unifier is unique but for a renaming of the variables.

EXAMPLE A.26

Consider the set of expressions $\{R(x, f(a, g(y))), R(b, f(z, w))\}$. Some possible unifiers of this set are $\sigma_1 = \{b/x, a/z, g(c)/w, c/y\}$, $\sigma_2 = \{b/x, a/z, f(a)/y, g(f(a))/w\}$ and $\sigma_3 = \{b/x, a/z, g(y)/w\}$. The last unifier is also a most general unifier: by the composition of this unifier with the substitution $\{c/y\}$ we get σ_1; the second unifier is obtained by the composition of σ_3 with $\{f(a)/y\}$.

The unification algorithm, more precisely, is a method for constructing a most general unifier of a finite, non-empty set of expressions. The algorithm considered in this book operates in the following manner. First, the left-most subexpressions in which the given expressions differ is computed. Their difference is placed in a set, called the *disagreement set*. Based on this disagreement set a ('most general') substitution is computed, which is subsequently applied to the given expressions, yielding a partial or total equality. If no such substitution exists, the algorithm terminates with the message that the expressions are not unifiable. Otherwise, the procedure proceeds until each element within each of the expressions has been processed. It can be proven that the algorithm either terminates with a failure message or with a most general unifier of the finite, unifiable set of expressions.

EXAMPLE A.27

Consider the following set of expressions:

$$S = \{Q(x, f(a), y), Q(x, z, c), Q(x, f(a), c)\}$$

The left-most subexpression in which the three expressions differ is in the second argument of the predicate symbol Q. So, the first disagreement set is $\{f(a), z\}$. By means of the substitution $\{f(a)/z\}$ the subexpressions in the second argument position are made equal. The next disagreement set is $\{y, c\}$. By means of the substitution $\{c/y\}$ these subexpressions are also equalized. The final result returned by the unification algorithm is the unifier $\{f(a)/z, c/y\}$ of S. It can easily be seen that this unifier is a most general one.

The following section shows an implementation of the unification algorithm. In the next section we discuss the role of unification in resolution in first-order predicate logic.

A.6.2 Resolution

Now that we have dealt with the subjects of substitution and unification, we are ready for a discussion of resolution in first-order predicate logic. We start with an informal introduction to the subject by means of an example.

EXAMPLE A.28

Consider the following set of clauses:

$$\{C_1 = P(x) \vee Q(x), C_2 = \neg P(f(y)) \vee R(y)\}$$

As can be seen, the clauses C_1 and C_2 do not contain complementary literals. However, the atoms $P(x)$, occurring in C_1, and $P(f(y))$, occurring in the literal $\neg P(f(y))$ in the clause C_2, are unifiable. For example, if we apply the substitution $\sigma = \{f(a)/x, a/y\}$ to $\{C_1, C_2\}$, we obtain the following set of instances:

$$\{C_1\sigma = P(f(a)) \vee Q(f(a)), C_2\sigma = \neg P(f(a)) \vee R(a)\}$$

The resulting instances $C_1\sigma$ and $C_2\sigma$ *do* contain complementary literals, namely $P(f(a))$ and $\neg P(f(a))$ respectively. As a consequence, we are now able to find a resolvent of $C_1\sigma$ and $C_2\sigma$, being the clause

$$C_3' = Q(f(a)) \vee R(a)$$

The resolution principle in first-order predicate logic makes use of the unification algorithm for constructing a most general unifier of two suitable atoms; the subsequent application of the resulting substitution to the literals containing the atoms, renders them complementary. In the preceding example, the atoms $P(x)$ and $P(f(y))$ have a most general unifier $\theta = \{f(y)/x\}$. The resolvent obtained after applying θ to C_1 and C_2, is

$$C_3 = Q(f(y)) \vee R(y)$$

The clause C_3' from the previous example is an instance of C_3, the so-called *most general clause*: if we apply the substitution $\{a/y\}$ to C_3, we obtain the clause C_3'.

It should be noted that it is necessary to rename different variables having the same name in both parent clauses before applying resolution, since the version of the unification

algorithm discussed in the previous section is not capable of distinguishing between equally named variables actually being the same variable, and equally named variables being different variables because of their occurrence in different clauses.

EXAMPLE A.29

Consider the atoms $Q(x, y)$ and $Q(x, f(y))$ occurring in two different clauses. In this form our unification algorithm reports failure in unifying these atoms (due to the occur check). We rename the variables x and y in $Q(x, f(y))$ to u and v respectively, thus obtaining the atom $Q(u, f(v))$. Now, if we apply the unification algorithm again to compute a most general unifier of $\{Q(u, f(v)), Q(x, y)\}$, it will come up with the (correct) substitution $\sigma = \{u/x, f(v)/y\}$.

We already mentioned in Section 2.3 that the meaning of a formula is left unchanged by renaming variables. We furthermore recall that formulas only differing in the names of their (bound) variables are called variants.

From the examples presented so far, it should be clear by now that resolution in first-order predicate logic is quite similar to resolution in propositional logic: literals are cancelled out from clauses, thus generating new clauses. From now on, cancelling out a literal L from a clause C will be denoted by $C\backslash L$.

Definition A.26 *Consider the parent clauses C_1 and C_2, respectively containing the literals L_1 and L_2. If L_1 and $\neg L_2$ have a most general unifier σ, then the clause $(C_1\sigma\backslash L_1\sigma) \vee (C_2\sigma\backslash L_2\sigma)$ is called a* binary resolvent *of C_1 and C_2. Resolution in which each resolvent is a binary resolvent, is known as* binary resolution.

A pair of clauses may have more than one resolvent, since they may contain more than one pair of complementary literals. Moreover, not every resolvent is necessarily a binary resolvent: there are more general ways for obtaining a resolvent. Before giving a more general definition of a resolvent, we introduce the notion of a factor.

Definition A.27 *If two or more literals in a clause C have a most general unifier σ, then the clause $C\sigma$ is said to be a* factor *of C.*

EXAMPLE A.30

Consider the following clause:

$$C = P(g(x), h(y)) \vee Q(z) \vee P(w, h(a))$$

The literals $P(g(x), h(y))$ and $P(w, h(a))$ in C have a most general unifier $\sigma = \{g(x)/w, a/y\}$. So,

$$C\sigma = P(g(x), h(a)) \vee Q(z) \vee P(g(x), h(a)) = P(g(x), h(a)) \vee Q(z)$$

is a factor of C. Note that one duplicate literal $P(g(x), h(a))$ has been removed from $C\sigma$.

The generalized form of resolution makes it possible to cancel out more than one literal from one or both of the parent clauses by first computing a factor of one or both of these clauses.

EXAMPLE A.31

Consider the following set of clauses:

$$\{C_1 = P(x) \vee P(f(y)) \vee R(y), C_2 = \neg P(f(a)) \vee \neg R(g(z))\}$$

In the clause C_1 the two literals $P(x)$ and $P(f(y))$ have a most general unifier $\sigma = \{f(y)/x\}$. If we apply this substitution σ to the clause C_1, then one of these literals can be removed:

$$\begin{aligned}(P(x) \vee P(f(y)) \vee R(y))\sigma &= P(f(y)) \vee P(f(y)) \vee R(y) \\ &= P(f(y)) \vee R(y)\end{aligned}$$

The result is a factor of C_1. The literal $P(f(y))$ in $C_1\sigma$ can now be unified with the atom $P(f(a))$ in the literal $\neg P(f(a))$ occurring in C_2, using the substitution $\{a/y\}$. We obtain the resolvent

$$C_3 = R(a) \vee \neg R(g(z))$$

Note that a total of three literals has been removed from C_1 and C_2. The reader can easily verify that there are several other resolvents from the same parent clauses:

- By taking $L_1 = P(x)$ and $L_2 = \neg P(f(a))$ we get the resolvent $P(f(y)) \vee R(y) \vee \neg R(g(z))$;
- Taking $L_1 = P(f(y))$ and $L_2 = \neg P(f(a))$ results in the resolvent $P(x) \vee R(a) \vee \neg R(g(z))$;
- By taking $L_1 = R(y)$ and $L_2 = \neg R(g(z))$ we obtain $P(x) \vee P(f(g(z))) \vee \neg P(f(a))$.

We now give the generalized definition of a resolvent in which the notion of a factor is incorporated.

Definition A.28 *A* resolvent *of the parent clauses C_1 and C_2 is one of the following binary resolvents:*

(1) A binary resolvent of C_1 and C_2;

(2) A binary resolvent of C_1 and a factor of C_2;

(3) A binary resolvent of a factor of C_1 and C_2;

(4) A binary resolvent of a factor of C_1 and a factor of C_2.

The most frequent application of resolution is refutation: the derivation of the empty clause $\square$ from a given set of clauses. The following procedure gives the general outline of this resolution algorithm.

```
procedure Resolution(S)
  clauses ← S;
  while □ ∉ clauses do
    {c_i, c_j} ← SelectResolvable(clauses);
```

```
            resolvent ← Resolve(c_i, c_j);
            clauses ← clauses ∪ {resolvent}
        od
    end
```

This algorithm is non-deterministic. The selection of parent clauses c_i and c_j can be done in many ways; how it is to be done has not been specified in the algorithm. Several different strategies have been described in the literature, each of them prescribing an unambiguous way of choosing parent clauses from the clause set. Such strategies are called the *control strategies* of resolution or *resolution strategies*. Several of these resolution strategies offer particularly efficient algorithms for making computer-based theorem proving feasible. Some well-known strategies are: *semantic resolution*, which was developed by J.R. Slagle in 1967, *hyperresolution* developed by J.A. Robinson in 1965, and various forms of *linear resolution*, such as *SLD resolution*, in the development of which R.A. Kowalski played an eminent role. At present, SLD resolution in particular is a strategy of major interest, because of its relation to the programming language Prolog.

A.7 Resolution strategies

Most of the basic principles of resolution have been discussed in the previous section. However, one particular matter, namely the efficiency of the resolution algorithm, has not explicitly been dealt with as yet. It is needless to say that the subject of efficiency is an important one for automated reasoning.

Unfortunately, the general refutation procedure introduced in Section A.6.3 is quite inefficient, since in many cases it will generate a large number of redundant clauses, that is, clauses not contributing to the derivation of the empty clause.

EXAMPLE A.32

Consider the following set of clauses:

$$S = \{P, \neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\}$$

To simplify referring to them, the clauses are numbered as follows:

(1) P

(2) $\neg P \vee Q$

(3) $\neg P \vee \neg Q \vee R$

(4) $\neg R$

If we apply the resolution principle by systematically generating all resolvents, without utilizing a more specific strategy in choosing parent clauses, the following resolvents are successively added to S:

(5) Q (using 1 and 2)

(6) $\neg Q \vee R$ (using 1 and 3)

(7) $\neg P \vee R$ (using 2 and 3)

(8) $\neg P \vee \neg Q$ (using 3 and 4)
(9) R (using 1 and 7)
(10) $\neg Q$ (using 1 and 8)
(11) $\neg P \vee R$ (using 2 and 6)
(12) $\neg P$ (using 2 and 8)
(13) $\neg P \vee R$ (using 3 and 5)
(14) $\neg Q$ (using 4 and 6)
(15) $\neg P$ (using 4 and 7)
(16) R (using 5 and 6)
(17) $\neg P$ (using 5 and 8)
(18) R (using 1 and 11)
(19) $\square$ (using 1 and 12)

This derivation of the empty clause $\square$ from S has been depicted in Figure A.3 by means of a derivation graph. As can be seen, by systematically generating all resolvents in a straightforward manner, fifteen of them were obtained, while, for instance, taking the two resolvents

($5'$) $\neg P \vee R$ (using 2 and 3)
($6'$) R (using 1 and $5'$)

would lead directly to the derivation of the empty clause:

($7'$) $\square$ (using 4 and $6'$)

In the latter refutation, significantly less resolvents were generated.

The main goal of applying a resolution strategy is to restrict the number of redundant clauses generated in the process of resolution. This improvement in efficiency is achieved by incorporating particular algorithmic refinements in the resolution principle.

A.8 Applying logic for building intelligent systems

In the preceding sections, much space has been devoted to the many technical details of knowledge representation and automated reasoning using logic. In the present section, we shall indicate how logic can actually be used for building a logic-based intelligent system. As logic offers suitable basis for model-based systems, more about the use of logic in the context of intelligent systems will said in Chapter 6.

In the foregoing, we have seen that propositional logic offers rather limited expressiveness, which in fact is too limited for most real-life applications. First-order predicate logic offers much more expressive power, but that alone does not yet render the formalism suitable for building intelligent systems. There are some problems: any automated reasoning method for full first-order logic is doomed to have a worst-case time complexity at least as bad as that of checking satisfiability in propositional logic, which is known to be NP-complete (this

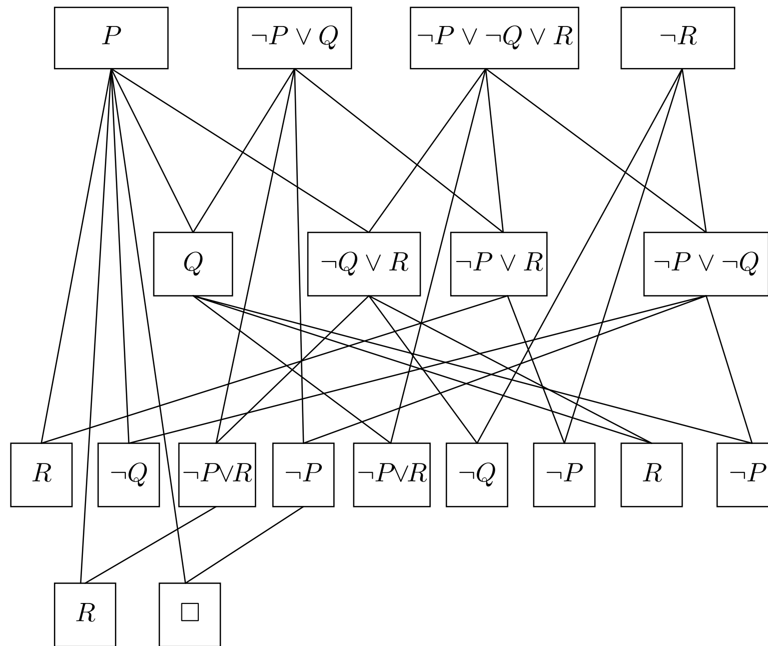

Figure A.3: Refutation of $\{P, \neg P \vee Q, \neg P \vee \neg Q \vee R, \neg R\}$.

means that no one has been able to come up with a better deterministic algorithm than an exponentially time-bounded one, although it has not been proven that better ones do not exist). Furthermore, we know that first-order predicate logic is undecidable; so, it is not even sure that an algorithm for checking satisfiability will actually terminate. Fortunately, the circumstances are not always as bad as that. A worst-case characterization seldom gives a realistic indication of the time an algorithm generally will spend on solving an arbitrary problem. Moreover, several suitable syntactic restrictions on first-order formulas have been formulated from which a substantial improvement of the time complexity of the algorithm is obtained; the Horn clause format we have paid attention to is one such restriction.

Since syntactic restrictions are only acceptable as far as permitted by a problem domain, we consider some examples, such as the logical circuit introduced in Chapter 1. However, first we discuss special standard predicates that are often used in modelling practical applications.

A.8.1 Reasoning with equality and ordering predicates

The special binary predicate symbols $>$ (*ordering predicate*) and $=$ *equality predicate* are normally specified in infix position, since this is normal mathematical practice. Both equality and the ordering predicates have a special meaning, which is described by means of a collection of axioms. The meaning of the equality predicate is defined by means of the following four axioms:

E_1 (*reflexivity*): $\forall x(x = x)$

E_2 (*symmetry*): $\forall x \forall y(x = y \rightarrow y = x)$

E_3 (*transitivity*): $\forall x \forall y \forall z(x = y \wedge y = z \rightarrow x = z)$

E_4 (*substitutivity*): $\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n((x_1 = y_1 \wedge \ldots \wedge x_n = y_n) \to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))$, and $\forall x_1 \ldots \forall x_n \forall y_1 \ldots \forall y_n((x_1 = y_1 \wedge \ldots \wedge x_n = y_n \wedge P(x_1, \ldots, x_n)) \to P(y_1, \ldots, y_n))$

Axiom E_1 states that each term in the domain of discourse is equal to itself; axiom E_2 expresses that the order of the arguments of the equality predicate is irrelevant. Axiom E_3 furthermore states that two terms which are equal to some common term, are equal to each other. Note that axiom E_2 follows from the axioms E_1 and E_3; nevertheless, it is usually mentioned explicitly. The three axioms E_1, E_2 and E_3 together imply that equality is an *equivalence relation*. Addition of axiom E_4 renders it a *congruence relation*. The first part of axiom E_4 states that equality is preserved under the application of a function; the second part expresses that equal terms may be substituted for each other in formulas.

EXAMPLE A.33

Consider the following set of clauses S:

$$S = \{\neg P(f(x), y) \vee Q(x, x), P(f(a), a), a = b\}$$

Suppose that, in addition, we have the equality axioms. If we add the clause $\neg Q(b, b)$ to S, the resulting set of clauses will be unsatisfiable. This can easily be seen informally as follows: we have $P(f(a), a) \equiv P(f(b), a)$ using the given clause $a = b$ and the equality axiom E_4. Now, we replace the atom $P(f(a), a)$ by the equivalent atom $P(f(b), a)$ and apply binary resolution.

The explicit addition of the equality axioms to the other formulas in a knowledge base suffices for rendering equality available for use in an intelligent system. However, it is well known that proving theorems in the presence of the equality axioms can be very inefficient, since many redundant clauses may be generated using resolution. Again, several refinements of the (extended) resolution principle have been developed to overcome the inefficiency problem. For dealing with equality, the resolution principle has for example been extended with an extra inference rule: *paramodulation*. Informally speaking, the principle of paramodulation is the following: if clause C contains a term t and if we have a clause $t = s$, then derive a clause by substituting s for a single occurrence of t in C. Therefore, in practical realizations equality is often only present implicitly in the knowledge base, that is, it is used as a 'built-in' predicate.

Another, more restrictive way to deal with equality is *demodulation*, which adds directionality to equality, meaning that one side of the equality may be replaces (rewitten) by the other side, but not the other way around.

Definition A.29 *A* demodulator *is a positive unit clause with equality predicate of the form* $(l = r)$*, where* l *and* r *are terms. Let* $C \vee L^t$ *a clause; where* L^t *indicates that the literal* L *contains the term* t*. Let* σ *be a substitution such that* $l\sigma = t$*, then* demodulatie *is defined as the inference rule:*

$$\frac{C \vee L^t, (l = r)}{C \vee L^{t \to r\sigma}}$$

where $L^{t \to r\sigma}$ *indicates that term* t *is rewritten to* $r\sigma$*.*

Thus a demodulator $(l = r)$ can be interpreted as a *rewrite rule* $l \rightarrow r$ with a particular orientation (here from left to right).

EXAMPLE A.34

Consider the demodulator

$$(brother(father(x)) = uncle(x))$$

and the clause

$$(age(brother(father(John))) = 55)$$

Application of the demodulator yields

$$(age(uncle(John)) = 55)$$

The applied substitution was $\sigma = \{Jan/x\}$.

In many real-life applications, a universally quantified variable ranges over a finite domain $D = \{c_i \mid i = 1, \ldots, n, n \geq 0\}$. The following property usually is satisfied: $\forall x(x = c_1 \vee x = c_2 \vee \ldots \vee x = c_n)$, with $c_i \neq c_j$ if $i \neq j$. This property is known as the *unique names assumption*; from this assumption we have that objects with different names are different.

EXAMPLE A.35

Consider the following set of clauses S:

$$S = \{\neg P(x) \vee x = a\}$$

We suppose that the equality axioms as well as the unique name assumption hold. Now, if we add the clause $P(b)$ to S, we obtain an inconsistency, since the derivable clause $b = a$ contradicts with the unique name assumption.

The ordering predicates $<$ and $>$ define a *total order* on the set of real numbers. They express the usual, mathematical 'less than' and 'greater than' binary relations between real numbers. Their meaning is defined by means of the following axioms:

O_1 (*irreflexivity*): $\forall x \neg(x < x))$

O_2 (*antisymmetry*): $\forall x \forall y(x < y \rightarrow \neg(y < x))$

O_3 (*transitivity*): $\forall x \forall y \forall z((x < y \wedge y < z) \rightarrow x < z)$

O_4 (*trichonomy law*): $\forall x \forall y((x < y \vee x = y \vee x > y)$

Axiom O_1 states that no term is less that itself; axiom O_2 expresses that reversing the order of the arguments of the predicate $<$ reverses the meaning. Axiom O_3 furthermore states that if a term is less than some other term, and this term is less than a third term, then the first term is less than the third one as well. Note that axiom O_2 follows from O_1 and O_3. The axioms O_1, O_2 and O_3 concern the ordering predicate $<$. The axioms for the ordering predicate

$>$ are similar to these: we may just substitute $>$ for $<$ to obtain them. Axiom O_4 states that a given term is either less than, equal to or greater than another given term. Again, in practical realizations, these axioms usually are not added explicitly to the knowledge base, but are assumed to be present implicitly as 'built-in' predicates or as evaluable predicates, that after it has been checked that all variables are instantiated to constants, are evaluated as an expression, returning true or false.

Exercises

(A.1) Consider the interpretation $v : PROP \to \{true, false\}$ in propositional logic, which is defined by $v(P) = false$, $v(Q) = true$ and $v(R) = true$. What is the truth value of the formula $((\neg P) \wedge Q) \vee (P \to (Q \vee R))$ given this interpretation v?

(A.2) For each of the following formulas in propositional logic determine whether it is valid, invalid, satisfiable, unsatisfiable or a combination of these, using truth tables:

(a) $P \vee (Q \to \neg P)$

(b) $P \vee (\neg P \wedge Q \wedge R)$

(c) $P \to \neg P$

(d) $(P \wedge \neg Q) \wedge (\neg P \vee Q)$

(e) $(P \to Q) \to (Q \to P)$

(A.3) Suppose that $F_1, \ldots, F_n$, $n \geq 1$, and G are formulas in propositional logic, such that the formula G is a logical consequence of $\{F_1, \ldots, F_n\}$. Construct the truth table of the implication $F_1 \wedge \cdots \wedge F_n \to G$. What do you call such a formula?

(A.4) Prove the following statements using the laws of equivalence for propositional logic:

(a) $P \to Q \equiv \neg P \to \neg Q$

(b) $P \to (Q \to R \equiv (P \wedge Q) \to R$

(c) $(P \wedge \neg Q) \to R \equiv (P \wedge \neg R) \to Q$

(d) $P \vee (\neg Q \vee R) \equiv (\neg P \wedge Q) \to R$

(A.5) Prove that the proposition $((P \to Q) \to P) \to P$, known as Peirce's law, is a tautology, using the laws of equivalence in propositional logic and the property that for any propositions π and ϕ, the formula $\pi \vee \neg\phi \vee \phi$ is a tautology.

(A.6) In each of the following cases, we restrict ourselves to a form of propositional logic only offering a limited set of logical connectives. Prove by means of the laws of equivalence that every formula in full propositional logic can be translated into a formula only containing the given connectives:

(a) the connectives $\neg$ and $\vee$.

(b) the connective $|$ which is known as the Sheffer stroke; its meaning is defined by the truth table given in Table A.6.

Table A.6: Meaning of Sheffer stroke.

F	G	$F \mid G$
true	*true*	*false*
true	*false*	*true*
false	*true*	*true*
false	*false*	*true*

(A.7) Consider the following formula in first-order predicate logic: $\forall x(P(x) \vee Q(y))$. Suppose that the following structure

$$S = (\{2,3\}, \varnothing, \{A : \{2,3\} \to \{\mathit{true}, \mathit{false}\}, B : \{2,3\} \to \{\mathit{true}, \mathit{false}\})$$

is given. The predicates A and B are associated with the predicate symbols P and Q, respectively. Now, define the predicates A and B, and a valuation v in such a way that the given formula is satisfied in the given structure S and valuation v.

(A.8) Consider the following statements. If a statement is correct, then prove its correctness using the laws of equivalence; if it is not correct, then give a counterexample.

(a) $\forall x P(x) \equiv \neg\exists x \neg P(x)$
(b) $\forall x \exists y P(x,y) \equiv \forall y \exists x P(x,y)$
(c) $\exists x(P(x) \to Q(x)) \equiv \forall x P(x) \to \exists x Q(x)$
(d) $\forall x(P(x) \vee Q(x)) \equiv \forall x P(x) \vee \forall x Q(x)$

(A.9) Transform the following formulas into the clausal form of logic:

(a) $\forall x \forall y \exists z(P(z,y) \wedge (\neg P(x,z) \to Q(x,y)))$
(b) $\exists x(P(x) \to Q(x)) \wedge \forall x(Q(x) \to R(x)) \wedge P(a)$
(c) $\forall x(\exists y(P(y) \wedge R(x,y)) \to \exists y(Q(y) \wedge R(x,y)))$

(A.10) For each of the following sets of clauses, determine whether or not it is satisfiable. If a given set is unsatisfiable, then give a refutation of the set using binary resolution; otherwise give an interpretation satisfying it:

(a) $\{\neg P \vee Q, P \vee \neg R, \neg Q, \neg R\}$
(b) $\{\neg P \vee Q \vee R, \neg Q \vee S, P \vee S, \neg R, \neg S\}$
(c) $\{P \vee Q, \neg P \vee Q, P \vee \neg Q, \neg P \vee \neg Q\}$
(d) $\{P \vee \neg Q, Q \vee R \vee \neg P, Q \vee P, \neg P\}$

(A.11) Let E be an expression and let σ and θ be substitutions. Prove that $E(\sigma\theta) = (E\sigma)\theta$.

(A.12) For each of the following sets of expressions, determine whether or not it is unifiable. If a given set if unifiable, then compute a most general unifier:

(a) $\{P(a,x,f(x)), P(x,y,x)\}$
(b) $\{P(x,f(y),y), P(w,z,g(a,b))\}$
(c) $\{P(x,z,y), P(x,z,x), P(a,x,x)\}$

(d) $\{P(z, f(x), b), P(x, f(a), b), P(g(x), f(a), y)\}$

(A.13) Use binary resolution to show that each one of the following sets of clauses is unsatisfiable:

(a) $\{P(x, y) \vee Q(a, f(y)) \vee P(a, g(z)), \neg P(a, g(x)) \vee Q(a, f(g(b))), \neg Q(x, y)\}$

(b) $\{append(nil, x, x), append(cons(x, y), z, cons(x, u)) \vee \neg append(y, z, u), \neg append(cons(1, cons(2, nil)), cons(3, nil), x)\}$

(c) $\{R(x, x), R(x, y) \vee \neg R(y, x), R(x, y) \vee \neg R(x, z) \vee \neg R(z, y), R(a, b), \neg R(b, a)\}$

Remark. The first three clauses in exercise (c) define an equivalence relation.

(A.14) Consider the set of clauses $\{\neg P, P \vee Q, \neg Q, R\}$. We employ the set-of-support resolution strategy. Why do we not achieve a refutation if we set the set of support initially to the clause R?

(A.15) Develop a logic knowledge base for a problem domain you are familiar with.