LOGIC FOR PROBLEM SOLVING, REVISITED

Chapter · November 2014

CITATION
1

READS
107

1 author:

Robert Kowalski
Imperial College London
146 PUBLICATIONS **13,551** CITATIONS

SEE PROFILE

LOGIC FOR PROBLEM SOLVING

Robert Kowalski
Imperial College London
12 October 2014

PREFACE

It has been fascinating to reflect and comment on the way the topics addressed in this book have developed over the past 40 years or so, since the first version of the book appeared as lecture notes in 1974. Many of the developments have had an impact, not only in computing, but also more widely in such fields as mathematical, philosophical and informal logic. But just as interesting (and perhaps more important) some of these developments have taken different directions from the ones that I anticipated at the time.

I have organised my comments, chapter by chapter, at the end of the book, leaving the 1979 text intact. This should make it easier to compare the state of the subject in 1979 with the subsequent developments that I refer to in my commentary.

Although the main purpose of the commentary is to look back over the past 40 years or so, I hope that these reflections will also help to point the way to future directions of work. In particular, I remain confident that the logic-based approach to knowledge representation and problem solving that is the topic of this book will continue to contribute to the improvement of both computer and human intelligence for a considerable time into the future.

SUMMARY

When I was writing this book, I was clear about the problem-solving interpretation of Horn clauses, and I knew that Horn clauses needed to be extended. However, I did not know what extensions would be most important. In particular, I was torn between the extension of Horn clauses given by the full clausal form of first-order logic, and the extension given by logic programs with negative conditions. It has turned out that the latter extension has been much more widely accepted.

Semantics of Logic Programs

In Chapter 11, I explored two different kinds of semantics for negative conditions of logic programs - the object-level semantics (also called the completion semantics) in which logic programs are definitions in if-and-only-if form, and the meta-level semantics in which negative conditions are interpreted as meaning that the corresponding positive condition cannot be shown. Over the years, the meta-level semantics has overshadowed the object-level one.

These days, the meta-level semantics is more commonly understood in model-theoretic terms: In the simplest case, a logic program with negative conditions is understood as determining a canonical model, and solving a goal is understood as finding an instance of the goal that is true in the model. However, in Answer Set Programming, a program can have many such models. The program itself represents the goal, and different models represent different solutions of the goal.

This shift over time, from the completion semantics (in which solving a goal is interpreted as proving a theorem in first-order logic) to the (meta-level) model-theoretic semantics, is part of a wider shift in logic-based computing: from theorem-proving to model-generation. This shift is also associated with a better understanding of the relationship between logic programs and integrity constraints and with a greater appreciation of integrity constraints.

Goals as Integrity Constraints

When I was writing the book, I considered integrity constraints as supporting the development and maintenance of logic programs, along the lines suggested in Chapter 9. But back then, the semantics of integrity constraints was even more confused than the semantics of logic programs with negative conditions.

Today, I believe that the relationship between logic programs and integrity constraints is like the relationship between an agent's beliefs and the agent's goals. If logic programs represent the beliefs of an intelligent agent, then integrity constraints, in the form of sentences of full first-order logic, represent the agent's goals. In a similar way to that in which an agent's beliefs support the agent in maintaining and achieving its goals, logic programs support integrity constraints, by helping to generate models in which the integrity constraints are true.

I was struggling to understand the relationship between goals and beliefs when I was writing Chapter 13. Moreover, I did not appreciate then the importance of actions in generating new information that, together with existing beliefs, can help to make an agent's goals true. The information systems of Chapter 13 are open to assimilate changes in the world, but they are passive, accepting those changes without having the power to generate changes of their own.

In recent years, the notion of an intelligent agent that can both assimilate information and generate actions has become the dominant paradigm in artificial intelligence. It has also become the inspiration for much of my own work, in which the task of an intelligent agent is to generate actions, to make the agent's goals true in the model determined by the agent's beliefs about the world.

CHAPTER 1

Although the focus of the book is on understanding logical inference in problem solving terms, I wanted to emphasize the declarative nature of logic by devoting the first two chapters to the topics of semantics and knowledge representation. None the less, despite my good intentions, I fear that my treatment of semantics in this chapter was inadequate.

The Importance of Model-theoretic Semantics

It is misleading to say, as I do on page 1, that "logic is not concerned with the truth, falsity or acceptability of individual sentences". On the contrary, logic *is concerned*, for example, with the fact that there is no interpretation I that makes the sentence p *and not* p true. More generally, given a sentence s and an interpretation I, logic *is concerned* with checking whether I is a *model* of s, namely an interpretation that makes s true. Given a set of sentences S, logic is also concerned both with determining whether S is *satisfied* by an interpretation I, and in generating interpretations I that are models of S.

Over the last 45 years or so, much of my own research has been concerned with attempting to find the right balance between semantics and inference. Most of the time, my research has been dominated by a syntactic view, which elevates inference over semantics. The most extreme manifestation of this view was the paper "Logic without Model Theory" [Kowalski, 1995]. Since then, having argued the case against model theory and having considered the arguments and the counter-arguments, I have modified my position, embracing a form of model theory that can be regarded as reconciling the syntactic and semantic views. The most recent example of this conversion is the paper "Reactive Computing as Model Generation" [Kowalski and Sadri, 2014]. I will document some of the reasons for this change in some of my commentary on the later chapters of this book.

The version of model theory that I now advocate is equivalent to the one presented on pages 14 and 15, but with a different emphasis on *satisfiability* and *truth*, rather than on inconsistency. The new version still has a somewhat syntactic character in its restriction of interpretations to *Herbrand interpretations*, which are sets of variable-free (also called *ground*) atomic sentences. But it is firmly committed to the notion of *truth*, building on the definition of truth for ground atomic sentences: A ground atom A is true in a Herbrand interpretation I if (and only if) $A \in I$.

Restricting interpretations to Herbrand interpretations I does away with any mystery about the nature of the individuals and relationships that belong to I. An individual belongs to a Herbrand interpretation I of a set of sentences S if and only if it belongs to the "universe of discourse" (also called the *Herbrand universe*), which is the set of all the ground terms that can be constructed from the language of S. If that universe is too small for some purpose, then it can be enlarged by including in the language extra constants that do not occur explicitly

in S. Thus a ground term t of the language is interpreted as the individual t in I. A predicate symbol P of the language is interpreted as the relation that is the set of all tuples $(t_1, \ldots t_n)$ such that $P(t_1, \ldots t_n) \in I$.

Semantic Trees

Rereading this chapter, I am surprised that I did not present the semantic tree model generation and proof procedure. It takes only a few lines, and it was the topic of my first publication [Kowalski and Hayes, 1968], building on [Robinson, 1968]. More importantly, it is one of the simplest proof procedures for the clausal form of logic imaginable.

Given a set of clauses S and an enumeration $A_1, \ldots A_n, \ldots$ of all the (possibly infinitely many) ground atoms of the language of S (also called the *Herbrand base* of S), the method explores a binary tree representing the set of all Herbrand interpretations of S:

> The root node has no label. Given a partial branch, from the root node to a node N, assigning truth values to the atoms $A_1, \ldots A_n$, if A_n is not the last atom in the enumeration, then the node N has two children. One child has a label that assigns true to A_{n+1}, and the other child has a label that assigns false to A_{n+1}.
>
> The semantic tree procedure generates the tree, starting at the root node, terminating a branch at a failure node N if the assignment of truth values to the atoms $A_1, \ldots A_n$ along the branch from the root to N falsifies some clause C in S. (It does so by checking whether there exists a substitution σ such that all the positive atoms A in $C\sigma$ are assigned false and all the atoms A of the negative literals *not* A in $C\sigma$ are assigned true.)

The semantic tree procedure is a sound and complete refutation procedure: If a set of clauses has no model, then the procedure terminates in a finite number of steps with a finite subtree all of whose leaf nodes are failure nodes.

The semantic tree procedure is also a sound and complete model generation procedure: If a set of clauses has a model, then the procedure will generate the Herbrand model M containing all the ground atomic sentences that are true in the model. If M is finite, then the procedure will generate M in a finite number of steps.

We will see later, in the commentary of Chapter 7, that the semantic tree method can be used to provide a simple proof of the completeness of the resolution rule of inference. Moreover, as [Baumgartner, 2000] points out, the method is closely related to the Davis-Putnam-Logemann-Loveland [1962] (DPLL) algorithm used in *SAT solvers*, to check whether a set of sentences in propositional logic is satisfiable. Due in large part to the development of efficient SAT solvers since the mid-1990s, they have become a core technology in computer science, with applications in

such areas as combinatorial optimization, the automation of electronic design, and hardware and software verification.

CHAPTER 2

Since the publication of the book, the use of clausal logic as a declarative representation language has expanded greatly, especially in the fields of databases and artificial intelligence.

Logic and Databases

In 1979, there was hardly any overlap between these two fields, the big exception being the workshop on logic and databases organised by Minker, Gallaire and Nicholas in 1977. But most of the workshop's participants were researchers in artificial intelligence, keen to apply the advances in logic developed for artificial intelligence to database applications. There was little contact with researchers working in mainstream database theory.

One of the concerns of the database community at the time was that relational databases do not support the definition of recursive relations, such as the ancestor relation:

$$Ancestor(x, y) \leftarrow Parent(x, y)$$
$$Ancestor(x, y) \leftarrow Ancestor(x, z), Ancestor(z, y)$$

(missing from my family relationships example in Chapter 1, but mentioned later in Chapters 7 and 10). Aho and Ullman [1979] proposed to remedy this deficiency of relational databases by extending the relational algebra with fixed point operators. But researchers working in artificial intelligence and logic programming did not see the need for fixed point operators, and were content to write down such definitions in the obvious way presented here. Harel [1980] published a harsh review of the logic and databases workshop proceedings [Gallaire and Minker, 1978], criticising it for claiming that first-order logic could be used to *define* recursive relations in such a simple way.

It took years to reconcile these two different intuitions about what it means to define a relation, and it is too early in my commentary to go into the issues here. Suffice it to say, for the moment, that the problem is touched upon in the 1979 book on page 31, and that I will discuss these issues in my commentary on Chapter 11.

Despite this early tussle with the database community, the field of *deductive databases*, represented by the papers in the 1978 workshop proceedings, prospered in the 1980s. Much of this work was concerned with developing semantics and proof procedures for integrity checking, to which I also contributed [Sadri and Kowalski, 1988]. However, during the same period, there emerged a new Datalog community, influenced by logic programming and deductive databases, but with its roots firmly in the database field.

The distinguishing feature of Datalog is that it restricts deductive databases to clauses that do not contain function symbols. This restriction means that the Herbrand universe and Herbrand base of a Datalog program are always finite, and query evaluation can be implemented in a way that always terminates.

The deductive database and Datalog fields also tended to view the semantics of databases in different terms. In the field of deductive databases, a database is viewed as a theory, and query evaluation is viewed as deriving answers that are theorems, which are logically implied by the database. However, in Datalog, influenced by relational database theory, a database represented in logical form is viewed as defining a Herbrand interpretation, and query evaluation is viewed as determining the truth value of the query. We will see later that these two views are equivalent for Horn clause databases and for queries that are existentially quantified conjunctions of atomic formulae [van Emden and Kowalski, 1976]. However, they differ for more general queries, and this is related to the two conflicting notions of definition mentioned above.

Deductive databases and Datalog went into decline in the 1990s and early 2000s, but Datalog experienced a revival in the mid 2000s "with a wide range of new applications, including data integration, declarative networking, program analysis, information extraction, network monitoring, security, optimizations, and cloud computing" [Green et al, 2013].

In both deductive databases and Datalog, databases are defined by *logic programs*, which in Chapter 5 are identified with sets of Horn clauses. But these days - and for many years now - (*normal*) *logic programs* are understood more generally as referring to sets of sentences (also called *clauses*), which have the same form as Horn clauses, but which can also have negative conditions:

$$A_0 \leftarrow A_1, ..., A_n, not\ B_1, ..., not\ B_m \text{ where } n \geq 0 \text{ and } m \geq 0.$$

Keith Clark [1978] started working on negation in logic programming around 1976, when I was about half way through writing the book. As a result, I did not deal with normal logic programs in the 1979 book until Chapter 11. In my commentary, I will discuss the effect that using normal logic programs would have had on the earlier chapters of the book.

Legislation as Logic Programs

I introduce the term *logic program* in Chapter 5 in the context of the procedural interpretation of Horn clauses. However, in both deductive databases and Datalog, logic programs have a purely declarative flavour. This declarative reading of logic programs is also shared with its application to the representation of law, which came into prominence with the representation of the British Nationality Act as a logic program [Sergot et al, 1986]. Although many readers will already be familiar with this application, for those who are not, it is easiest to explain it by means of some examples. Here is the very first sentence of the British Nationality Act of 1981:

> 1.-(1) A person born in the United Kingdom after commencement shall be a British citizen if at the time of the birth his father or mother is -
>
> (a) a British citizen; or
> (b) settled in the United Kingdom.

The sentence is very close to its logical representation as one sentence in the standard form of logic, but four sentences written as Horn clauses.

The example is not entirely typical, because most legislation has the structure of rules and exceptions. Here is a simplified example:

> A person can be deprived of British citizenship, if the person received that citizenship through naturalisation, and the application for naturalisation was fraudulent. A person cannot be deprived of British citizenship, if the deprivation would make the person stateless.

The literal translation into clausal form produces two clauses, say:

$$CanDeprive(x) \leftarrow Naturalised(x), Fraudulent(x)$$
$$\leftarrow CanDeprive(x), WouldBeStateless(x)$$

However, in the situation exemplified by $Naturalised(John)$, $Fraudulent(John)$, $WouldBeStateless(John)$, this literal translation leads to inconsistency.

Obviously, the literal translation does not do justice to the law's intention, because the inconsistency is not intended. Here is a better translation of the intended meaning into ordinary clausal form:

$$CanDeprive(x), Exception(x) \leftarrow Naturalised(x), Fraudulent(x)$$
$$Exception(x) \leftarrow WouldBeStateless(x)$$
$$\leftarrow CanDeprive(x), Exception(x)$$

The better translation introduces a new predicate $Exception(x)$ expressing that the person x is an exception to the rule. The third clause expresses, in effect, that a person cannot be deprived of citizenship if the person is an exception.

It is even more natural to translate the two sentences into normal logic programming form:

$$CanDeprive(x) \leftarrow Naturalised(x), Fraudulent(x), not\ Exception(x)$$
$$Exception(x) \leftarrow WouldBeStateless(x)$$

Logic and the World Wide Web

Deductive databases, Datalog and the representation of law illustrate how logic programming has developed as a declarative representation language since the publication of this book. One other area, discussed in this chapter, that has also developed significantly is the use of binary relations in knowledge representation, especially in the context of the World Wide Web.

The basis for much of this work is the RDF (Resource Description Framework) data model, which represents assertions about WWW resources as triples, which encode binary relationships along with the name of the relation itself. The Web Ontology Language (OWL), endorsed by the World Wide Web Consortium, embeds RDF in Description Logic [Baader, 2013], which focuses on the logic of unary relations, which represent *concepts* and binary relations, which represent *roles*.

A good overview of the most important influences on the design of OWL from RDF, Description Logics and frames is [Horrocks et al, 2003]. The semantic web rule language SWRL [Horrocks et al, 2004] is a proposal to combine OWL with rules expressed as Horn clauses.

CHAPTER 3

This is one of my favourite chapters. I particularly like its characterisation of reasoning with Horn clauses as filling in a triangle, with the goal at the top and the facts or assumptions at the bottom. A similar metaphor underlies the *pyramid principle* [Minto, 2010], which is "a technique for working out your thinking on any subject, so that you can present it clearly to someone else". The fact that the pyramid principle is designed for human use, rather than for computer implementation, supports my claim in the Preface, that "although the inference methods in this book were originally designed for use by computers, they can also be used by human beings". This claim is also the topic of my second book [Kowalski, 2011].

The Relationship between Inconsistency and Logical Consequence

There is one change I would make to this Chapter, if I were writing it today. Instead of describing filling in the triangle as showing inconsistency, I would describe it as showing logical consequence. The explanation in terms of inconsistency was motivated by viewing Horn clauses as a special case of the clausal form of logic, and by viewing top-down and bottom-up reasoning as special cases of the resolution rule. Because resolution is refutation-complete, but not deduction-complete, to show that a goal G is a logical consequence of a set S of clauses in the general case, it is necessary to convert the negation of G into a set G' of clauses and show that $S \cup G'$ is inconsistent.

I should have exploited the fact that refutation completeness for Horn clauses can be translated into *deduction completeness* for the kind of reasoning needed to fill in the triangle. To understand this kind of reasoning better, it is useful to distinguish between Horn clauses that have exactly one conclusion, called *definite clauses*, and Horn clauses of the form $\leftarrow A_1, ..., A_n$ having no conclusion at all, called *denials*.

But if we ignore the arrow $\leftarrow$ in the front of a denial, and interpret all the variables $x_1, ..., x_n$ as existentially quantified, then the denial can be interpreted positively as expressing the goal of showing that:

there exist $x_1, ..., x_n$
such that A_1 and ... and A_n

This goal G is the opposite (or "dual") of the denial.

As a consequence of the refutation completeness of hyper-resolution (or bottom-up reasoning), if a set S of definite clauses is inconsistent with a denial of the form $\leftarrow A_1, ..., A_n$, then there exists a substitution σ and bottom-up derivations of assertions $A_1\sigma, ..., A_n\sigma$, where all of the variables in each of the $A_i\sigma$ are universally quantified. As a consequence of the refutation completeness of linear resolution (or top-up reasoning), there also exists a top-down derivation $C_1, ..., C_m$, starting from the goal $C_1 = G$ and ending with the empty conjunction of subgoals $C_m = \Box$. Moreover, there exists a substitution σ that can be extracted from the derivation, such that S logically implies the conjunction $G\sigma = A_1\sigma$ &...& $A_n\sigma$.

In both cases, all variables that are left in $G\sigma$ are universally quantified. Therefore the bottom-up and top-down derivations actually show that S logically implies $G\sigma\theta$ for all further substitutions θ. In other words, not only do the derivations solve the goal G by finding values for the variables in G, but they typically find whole classes of solutions represented by the ground instances of the variables in $G\sigma$.

Minimal Model Semantics

Replacing the interpretation of filling in the triangle as showing inconsistency by its interpretation as showing logical implication would make the inference methods presented in this chapter more natural, and the claim that these methods are congenial for human use more convincing. However, there is one other change that would also make this chapter more up-to-date, namely adding the model-theoretic view of bottom-up and top-down reasoning as determining the truth value of a goal clause G in the intended model M of a set S of definite clauses.

It is strange that I did not mention the model-theoretic view anywhere in the book, despite the fact that Maarten van Emden and I had been working on it since 1974 [van Emden and Kowalski, 1976]. The best explanation I can think of is that the proof-theoretic view was the core of the lecture notes for the book, completed in March 1974, before the model-theoretic view was fully developed. Moreover, as I mentioned in my commentary on Chapter 1, it took me a long time to fully appreciate the model-theoretic point of view.

The connection between the model-theoretic and proof-theoretic views is given by the minimal model theorem [van Emden and Kowalski, 1976]:

> Given a set S of definite clauses, let H be the Herbrand base of S, and let $M = \{A \in H \mid S \vdash A\}$, then M is a Herbrand model of S, and for any other Herbrand model M' of S, $M \subseteq M'$. In other words, M is the unique minimal model of S.

Here $\vdash$ can be any provability relation that is complete for deriving ground atoms A from definite clauses S. In particular, it can be hyper-resolution [Robinson,

1965b] or SLD-resolution [Kowalski, 1974]. In fact, because of the restricted form of S and A, the provability relation can be the special case of hyper-resolution in which the only rule of inference is:

> From C in S and ground atoms $A_1, ..., A_n$, derive ground atom A_0, where $A_0 \leftarrow A_1, ..., A_n$ is a ground instance of C.

Let $S \vdash_{Horn} A$ be the provability relation corresponding to this one rule of inference. The proof of the minimal model theorem is embarrassingly simple:

> To show that $M = \{A \in H \mid S \vdash_{Horn} A\}$ is a model of S, it is necessary to show that every clause C in S is true in M. But this is the case if for every ground instance $A_0 \leftarrow A_1, ..., A_n$ of C, if $A_1, ..., A_n$ are in M, then A_0 is also in M. But this is obviously the case, because there exists a one step derivation of A_0 from C and $A_1, ..., A_n$.
>
> To show that M is the minimal model of S, it suffices to show that, if $A \in M$, and M' is a Herbrand model of S, then $A \in M'$. But if $A \in M$, then $S \vdash_{Horn} A$ by definition. Therefore because $\vdash_{Horn}$ is complete for definite clauses S and ground atoms A, it follows that $S \models A$, which means that A is true in all models of S. Therefore A is true in M'.

We will see in later commentaries that much of the research on the semantics of logic programming since 1979 has been concerned with extending this minimal model semantics of definite clauses to more general, normal logic programs.

CHAPTER 4

This chapter also has its origins in the 1974 lecture notes. But its focus on problem-solving avoids the semantic issues of logical implication and model-theoretic truth, which affect the earlier chapters.

As was my intention in the preceding chapters, I wanted to present a view of logic for problem-solving that would be relevant for both human use and computer applications. As Sherlock Holmes explained to Dr. Watson, in *A Study in Scarlet*:

> In solving a problem of this sort, the grand thing is to be able to reason backward. That is a very useful accomplishment, and a very easy one, but people do not practise it much. In the everyday affairs of life it is more useful to reason forward, and so the other comes to be neglected. There are fifty who can reason synthetically for one who can reason analytically.

As mentioned briefly in the introduction to Chapter 3 of the 1979 book, backward reasoning and analytical reasoning are synonymous with top-down reasoning, while forward and synthetic reasoning are synonymous with bottom-up reasoning.

I confess that I have been disappointed by the apparent lack of attention that the problem-solving interpretation has received outside computing circles. It was this disappointment that led to my second attempt [Kowalski, 2011] with its provocative subtitle, *How to be Artificially Intelligent*.

Prolog and Other Implementations of Logic Programming

However, together with the procedural interpretation in Chapter 5, the problem-solving interpretation in this chapter has been fairly well received in the world of computing, thanks in large part to the impact of the programming language Prolog. Indeed, if I were to rewrite this book from scratch, one of the biggest changes I would consider would be to rewrite clauses in Prolog syntax, and to discuss their Prolog execution.

The impact of the problem-solving and procedural interpretations of logic reached its zenith with the Fifth Generation Computer Systems (FGCS) Project. The FGCS Project [Moto-Oka,1982] was a ten year programme beginning in 1982, sponsored by the Japanese Ministry of International Trade and Industry and involving all the major Japanese computer manufacturers. Its objective was to develop a new generation of computers employing massive parallelism and oriented towards artificial intelligence applications. Logic programming was chosen as the foundation for the software to fill the gap between the applications and the hardware.

The story of the FGCS Project is long and complicated. However, it is possible to argue that the FGCS Project was ahead of its time. One support for this argument is the success of the AI system Watson developed at IBM [Ferrucci et al, 2010], which competed on the quiz show *Jeopardy!* and defeated two former, human winners in 2011. According to [Lally et al, 2012] most of the rule-based question analysis components in Watson are implemented in Prolog.

In expert hands, Prolog can perform wonders. But for a beginner, its depth-first search strategy can cause big trouble. Consider the following variant of the very first example in Chapter 1. Here is one way of rewriting it in Prolog notation, in which predicate symbols, function symbols and constants start with a lower case letter, variables start with an upper case letter, :− stands for ← and every clause ends in a full stop:

$$likes(bob, X) :- likes(X, bob).$$
$$likes(bob, logic).$$
$$:- likes(bob, X).$$

The third clause represents the goal of finding an X liked by bob.

Execution by Prolog fails to find the solution $X = logic$, because its use of depth-first search generates an infinite loop: It starts by using the first clause to reduce the goal $likes(bob, X)$ to the subgoal $likes(X, bob)$. It then uses the first clause again, but this time to reduce $likes(X, bob)$ to the subgoal $likes(bob, bob)$. It then repeatedly uses the first clause to reduce $likes(bob, bob)$ to $likes(bob, bob)$, without getting a chance to solve the problem by using the second clause. If the

order of the two clauses is reversed, Prolog finds the solution, and if only one solution is desired then it terminates. But if all solutions are desired, then it encounters the infinite branch, and goes into the same infinite loop.

The problem of infinite loops is a major theme of this chapter. However, I ignored the difficulty it causes for programs written in Prolog. It was not until the development of XSB Prolog [Chen and Warren, 1996; Swift and Warren, 2012], building upon the loop detection strategies of Brough and Walker [1984] and the tabling technique of Sato and Tamaki [1986] that a sufficiently practical solution of these problems became available.

These days, in addition to the use of tabling, there also exist other techniques for efficiently executing logic programs and for avoiding infinite loops. They include the model generation methods of Answer Set programming (ASP) [Brewka et al, 2011] and the bottom-up execution strategies of Datalog [Ceri et al, 1989]. However, these techniques ignore the problem-solving and procedural interpretations of logic programs, and focus on purely declarative knowledge representation.

These days, ASP and Datalog are booming, whereas Prolog is just keeping its head above water. In my opinion, part of the problem lies with a lack of appreciation that, with the use of tabling to avoid most infinite loops and to overcome other inefficiencies, it is possible to write purely declarative programs in Prolog. I was involved in the implementation of just such a program, which has been used by WHO and UNICEF since 2009, to implement a set of logical rules to assist in estimating global, country by country, infant immunization coverage [Burton et al, 2012; Kowalski and Burton, 2012].

The WHO/UNICEF program was first written in a variant of Prolog without tabling, and was intolerably inefficient - not because of infinite loops, but because it recomputed the same estimates over and over. Rerunning the same program in XSB with tabling speeded up the program by a factor of about 100.

I admit that some of my collaborators found the logic-based syntax of Prolog intimidating. But, as I tried to suggest in Chapter 2, and as many other researchers have pointed out, logic is compatible with a variety of more user-oriented notations, including graphical representations like semantic networks.

CHAPTER 5

Whereas Chapter 4 was concerned with top-down reasoning as a problem-solving technique for use by humans or by computers, this chapter is concerned more narrowly with its use for execution by computers.

Concurrent and Parallel Logic Programming

At the time I was writing the book, much of the effort in the logic programming group at Imperial College was devoted to developing IC-Prolog [Clark and McCabe, 1979; Clark et al, 1982]. IC-Prolog avoided the non-logical features of standard Prolog, and added negation, set expressions and a rich set of control

facilities, for example to control the order of executing subgoals (procedure calls) depending on the pattern of their input-output parameters (i.e. their *mode* of use), and to execute subgoals in parallel.

One offshoot of IC-Prolog was the Relational Language of Clark and Gregory [1981], which was the first in a series of concurrent logic programming languages, including Concurrent Prolog [Shapiro, 1987], Parlog [Clark and Gregory, 1986], and Guarded Horn Clauses (GHC) [Ueda, 1986]. In these languages, a program is a set of guarded Horn clauses of the form:

$$P \leftarrow G_1,\ \ldots\ , G_n \mid A_1,\ \ldots\ , A_m \quad \text{where } n \geq 0 \text{ and } m \geq 0.$$

The conjunction $G_1,\ \ldots\ , G_n$ is the *guard* of the clause, and $\mid$ is the *commitment* operator. Declaratively, the clause is read as an ordinary Horn clause:

$$P \text{ if } G_1 \text{ and } \ldots \text{ and } G_n \text{ and } A_1 \text{ and } \ldots \text{ and } A_m.$$

Procedurally, when a goal is activated, all of the procedures whose head P matches the goal are invoked, and their guards are evaluated in parallel. If the guards of several clauses succeed, then an arbitrary commitment is made to one of the clauses (typically to one of the clauses whose guard first succeeds), and execution proceeds with the remaining body of the clause. The procedure calls in the remaining body can also be executed in parallel. Thus concurrent logic programming includes a form of *and-parallelism*, *or-parallelism* and *don't care non-determinism*.

According to [Ueda, 1999], there was a big controversy within the FGCS Project about whether to base the software on Prolog-style logic programming, which is closer to AI applications, or on concurrent logic programming, which is closer to parallel computer implementations. Concurrent logic programming won the competition early in the Project.

One of the most serious competitors to the FGCS approach was the Gigalips Project [Lusk et al, 1990], which developed an or-parallel implementation of Prolog. The Project was an informal consortium involving The Swedish Institute of Computer Science (SICS), Argonne National Laboratory and the University of Manchester (later moved to the University of Bristol).

Neither the FGCS nor the Gigalips approaches to parallelism were widely accepted. However, in recent years, parallel execution of conventional programs over networks of processors has produced huge gains of efficiency. Moreover, the methods used to implement these systems are reminiscent of the ones used earlier to implement logic programs in parallel. For example, the MapReduce programming model used in Google [Dean and Ghemawat, 2008] is similar to the use of top-down reasoning to decompose goals into subgoals, solving the subgoals in parallel, then collecting and combining the solutions bottom-up. I find it hard to believe that these methods will not be applied to logic programming again in the future.

Answer Set Programming

Parallel execution of logic programs, and more generally different ways of executing logic programs, are one of the two main topics of this chapter. The other main

topic is the ways in which the same problem can be represented and solved by different logic programs. I gave examples of different representations of the path-finding and list-sorting problems. I also discussed the different possibilities of representing data in general: by means of terms or assertions; and I suggested that the representation by assertions is more powerful in many cases.

In recent years, Answer Set Programming (ASP) has taken the idea of representing data by assertions much further. In ASP, the input data is represented by assertions, and the output is represented by Herbrand models.

ASP exploits the fact that a normal logic program (with negative conditions) can have several "intended" Herbrand models, and that different models can represent different outputs for the same input. However, the basic idea of ASP also applies to Horn clause programs, which have a unique intended minimal model given any set of assertions as input. The Horn clause definition of *Ancestor* is a simple example:

$Ancestor(x, y) \leftarrow Parent(x, y)$
$Ancestor(x, y) \leftarrow Ancestor(x, z), Ancestor(z, y)$

Suppose we are given as input data the assertions:

$Parent(Zeus, Ares)$
$Parent(Hera, Ares)$
$Parent(Ares, Harmonia)$

Then the parenthood assertions together with the ancestor assertions:

$Ancestor(Zeus, Ares)$
$Ancestor(Hera, Ares)$
$Ancestor(Ares, Harmonia)$
$Ancestor(Zeus, Harmonia)$
$Ancestor(Hera, Harmonia)$

constitute the unique minimal model of the input data and the definition of *Ancestor*. The *Ancestor* assertions in the model can be regarded as the output corresponding to the input.

Given the same input data and the program:

$Mother(x, y) \leftarrow Parent(x, y),\ not\ Father(x, y)$
$Father(x, y) \leftarrow Parent(x, y),\ not\ Mother(x, y)$

instead of the *Ancestor* program, there are eight minimal models (or answer sets), corresponding to the different alternatives for whether the parents *Zeus, Hera* and *Ares* are mothers or fathers.

It is too early in the commentary to discuss the ASP semantics here. Suffice it for now to underline that ASP treats program execution as model generation, and not as inference. In fact, the model generation techniques used in ASP are similar to those used in SAT solvers, and can similarly be viewed as optimizations of the semantic tree method described in the commentary on Chapter 1.

Transaction Logic Programming

Transaction Logic Programming [Bonner and Kifer, 1993] builds on similar ideas of representing data by means of assertions, but uses logic programs both to define data and to manipulate it. The data is represented in a deductive database, and the logic programs that manipulate it do so by querying and updating the database. Such deductive databases are clearly much higher-level than the data structures found in conventional programming languages.

As this chapter argues and as subsequent developments have shown, logic programming can support both terms and assertions as data structures. Transaction Logic shows that logic programming can also support deductive databases as data structures. However, I like to think that the Transaction Logic approach points in the direction of the programming languages of the future.

CHAPTER 6

This is another one of my favourite chapters. Like Chapter 3, it emphasizes the difference between reasoning top-down and reasoning bottom-up, this time in the context of the planning problem. Of course, much progress has been made in the area of representing and reasoning about actions since the writing of this book. However, logic programming has continued to play a useful role in these later developments. In particular, the *Holds* predicate and negation as failure have become standard tools in the knowledge representation tool kit Using these tools, here is how I would represent the situation calculus in logic programming form today, but employing the notational conventions of the 1979 book:

$$Holds(f, result(a, s)) \leftarrow Poss(a, s), Initiates(a, f, s)$$
$$Holds(f, result(a, s)) \leftarrow Poss(a, s),\ Holds(f, s), not\, Terminates(a, f, s)$$

Here $Initiates(a, f, s)$ expresses that the action a performed in state s initiates f in the resulting state $result(a, s)$; and $terminates(a, f, s)$ expresses that a terminates f. Together, the two clauses assert that a fact holds in the state resulting from an action either if it was initiated by the action or if it held in the previous state and was not terminated by the action. The main difference from the representation used in the book is the explicit use of negation in the second clause.

The Event Calculus

Since the publication of the book, probably the most influential development to which I have contributed is the event calculus [Kowalski and Sergot, 1986], which combines some of the ideas about the representation of events in Chapter 2 with the ideas of representing change in this chapter.

The event calculus replaces states by linearly ordered time points and uses a predicate $Happens(e, t)$ to express that an event e happens at a time t. An event can be either an external event or an action. Here is a simplified version of the

event calculus, using similar predicates to those used in the situation calculus representation above, and using an extension of logic programming form in which conditions can be formulas of full first-order logic [Lloyd and Topor, 1984]:

$$Holds(f, t2) \leftarrow Happens(e1, t1), Initiates(e1, f, t1), t1 < t2,$$
$$not\ \exists e\ \exists t\ [Happens(e, t), Terminates(e, f, t), t1 \leq t < t2]$$

The single clause asserts that a fact holds at a time $t2$ if it was initiated by an event taking place at an earlier time $t1$ and it was not terminated by an event taking place between $t1$ and $t2$.

The Lloyd-Topor transformation converts this clause into normal logic programming form by using an auxiliary predicate. For example:

$$Holds(f, t2) \leftarrow Happens(e1, t1), Initiates(e1, f, t1), t1 < t2,$$
$$not\ Clipped(f, t1, t2)$$
$$Clipped(f, t1, t2) \leftarrow Happens(e, t), Terminates(e, f, t), t1 \leq t < t2$$

The event calculus was intended for reasoning about the consequences of given events, catering for the possibility that several events can occur simultaneously. It was also intended to alleviate the computational inefficiency of using the situation calculus frame axiom to determine whether a fact f holds at a time $t2$ by considering *all* of the actions and other events e taking place between $t2$ and the time $t1$ when f was initiated. Using the event calculus, it suffices to consider only those (considerably fewer) events e between $t1$ and $t2$ that can terminate f.

The event calculus was not originally intended for planning applications. However Eshghi [1988] showed that planning problems can be solved by using abductive reasoning, where:

> P is a logic program consisting of the event calculus together with clauses defining the predicates $Initiates$, $Terminates$, $Poss$, $\leq$, $<$, and assertions of the form $Happens(E_0, 0)$, describing events that generate an initial state.
> A is a set of ground atoms of the form $Happens(E, T)$, describing candidate actions.
> I is a set of integrity constraints of the form $Poss(a, t) \leftarrow Happens(a, t)$.
> G is a goal clause of the form $\leftarrow Holds(F_1, t), \ldots, Holds(F_n, t)$, where t is the time of the goal state.

To solve the planning problem, it suffices to find a set $\Delta \subseteq A$ such that $P \cup \Delta$ solves G and satisfies I.

Abductive Logic Programming

The application of abduction to planning with the event calculus is a special case of abductive logic programming (ALP) [Kakas et al, 1992; Denecker and Kakas, 2002], which is closely related to answer set programming (ASP). The map colouring program, commonly used to illustrate ASP, can also be used to illustrate

ALP. Given a map with countries x represented by the predicate $Country(x)$ and adjacent countries x and y represented by $Adjacent(x, y)$, the problem is to find a colouring *Red*, *Yellow* or *Green* for each country, such that no two adjacent countries have the same colour. Here is a simple representation in ALP:

P defines the predicates *Country*, *Adjacent* and $=$, defined by $x = x$.
A is a set of ground atoms of the form $Colour(X, C)$.
$I = \{ Colour(x, Red), Colour(x, Yellow), Colour(x, Green) \leftarrow Country(x),$
$c = d \leftarrow Colour(x, c), Colour(x, d),$
$\leftarrow Colour(x, c), Colour(y, c), Adjacent(x, y)\}$

To solve the map colouring problem, it suffices to find a set $\Delta \subseteq A$, such that $P \cup \Delta$ satisfies I.

In general, an *abductive framework* is a triple $\langle P, I, A\rangle$, where P is a logic program, I is a set of integrity constraints, and A is a set of ground atoms (whose predicates are called *abducible*). Given $\langle P, I, A\rangle$ and a goal clause G, an *abductive solution* (or *explanation*) of G is a subset Δ of A, such that

$P \cup \Delta$ solves G and
$P \cup \Delta$ satisfies I.

It is conventional in introductions to ALP, to state the solution and satisfaction requirements in such vague (or abstract) terms. This is partly because many concrete instances of these notions are possible, and partly because it has been difficult to decide which instances are best. The biggest difficulty has been how to understand integrity constraint satisfaction.

Having been involved in the early debates about the semantics of integrity constraints in the 1980s, and having contributed to proof procedures for both integrity checking [Sadri and Kowalski, 1988] and ALP [Fung and Kowalski, 1997], I am now convinced that solving G and satisfying I should both be understood in the same model-theoretic terms. Moreover, there is no need to distinguish between goals and integrity constraints, because goal clauses can be regarded as just a special kind of integrity constraint. In the simple case where P is a Horn clause program:

Given $\langle P, I, A\rangle$, an *abductive solution* is a subset Δ of A, such that I is true in the minimal model of $P \cup \Delta$.

With this semantics, finding an abductive solution in ALP is a model-generation problem, similar to ASP. Moreover, it has the added attraction that, because the definition of truth applies to any sentence of first-order logic (FOL), I can be an arbitrary set of sentences of FOL (as in the ID-Logic of [Denecker, 2000]). It also extends to the case where P is a more general logic program whose semantics is defined in terms of a unique canonical model, as in the case of the perfect model of a locally stratified program [Przymusinski, 1988], or the well-founded semantics of an arbitrary logic program [Van Gelder et al, 1991].

From EC and ALP to LPS

Both the event calculus (EC) and ALP have been relatively well received. Moreover, they have both contributed to my recent work with Fariba Sadri on developing a Logic-based Production System language (LPS) [Kowalski and Sadri; 2009, 2011, 2012, 2014a, 2014b]. The framework was originally intended to give a logical semantics to condition-action rules in production systems. But its original goal has been greatly extended to incorporate the functionalities of many other frameworks, including BDI agents, Golog, active databases, Transaction Logic, and abstract state machines.

In LPS computation is viewed as performing actions and generating state transitions in order to make rules of the form:

$$\forall x[antecedent \rightarrow \exists y[consequent_1 \vee \ldots \vee consequent_n]]$$

true. Here *antecedent* and the disjuncts $consequent_i$ are temporally constrained conjunctions of event atoms and FOL formulae referring to individual states. The *antecedent* of a rule represents a complex (or composite) event, and each $consequent_i$ represents a conditional plan of actions (or a transaction).

The framework combines such rules both with logic programs that define intensional predicates, complex event predicates and complex transaction predicates, and with an event theory that specifies the postconditions and preconditions of events. In the model-theoretic semantics, time is represented explicitly, as in the event calculus. However, in the computation of the model, states are updated destructively, as in most practical programming and database languages.

CHAPTER 7

Before rereading this chapter to write this commentary, I remembered it as a standard introduction to resolution. In fact, it is an introduction to resolution with a focus on how non-Horn clauses extend Horn clauses. Back in those days, I thought that it would be useful to extend logic programs from Horn clauses to non-Horn clauses. In particular, that was the point of exercise 6. But in practice other extensions of Horn clause programs have proved to be more useful - the most important one being the extension to normal logic programs, with negative conditions. This extension from Horn clauses to normal logic programs, and the relationship with non-Horn clauses is one of the main topics of Chapter 11.

Normal Logic Programs with Negative Conditions

Chapter 11 contains several examples showing how (meta-level) reasoning with normal logic programs, using negation as failure, can sometimes simulate (object-level) reasoning with non-Horn clauses expressing the only-if halves of definitions.

Here is another example, showing how the non-Horn clauses T1-14 in this chapter can be represented by a normal logic program:

T1 $True(x \ \& \ y) \leftarrow True(x), True(y)$
T4 $True(x \vee y) \leftarrow True(x)$
T5 $True(x \vee y) \leftarrow True(y)$
T7' $True(x \supset y) \leftarrow not\ True(x)$
T8 $True(x \supset y) \leftarrow True(y)$
T10 $True(x \leftrightarrow y) \leftarrow True(x \supset y), True(y \supset x)$
T13' $True(\neg x) \leftarrow not\ True(x)$

The non-Horn clauses T7 and T13 are turned into the logic programming clauses T7' and T13' by making one of the alternative conclusions a negative condition. The missing clauses T2 and T3 are the only-if halves of T1; clause T6 is the only-if half of T4 and T5; clause T9 is the only-if half of T7' and T8; clauses T11 and T12 are the only-if half of T10; and T14 is the only-if half of T13'.

The logic program above defines the predicate *True* for compound sentences of propositional logic, given definitions of *True* for atomic sentences. The missing only-if halves of the program are not needed to compute the predicate *True*, but are needed to prove properties of the program. In Chapter 11, I argued that using the non-Horn clauses expressing the only-if halves of the definition (in the object language) is one way to prove such properties, but reasoning about the logic program in the meta-language is another way.

Here is an informal sketch of such a meta-language argument for showing that, for all sentences p of propositional logic, $True(p \supset p)$ is a property of the logic program above:

> There are only two ways of using the program to show $True(p \supset p)$, either by using T7' and showing $not\ True(p)$ or by using T8 and showing $True(p)$. Interpreting negation as failure, the only way to show $not\ True(p)$ is by trying to show $True(p)$ and failing. But trying to show $True(p)$ either succeeds or fails. If it succeeds, then using T8 shows $True(p \supset p)$. If it fails, then using T7' shows $True(p \supset p)$. So either way, $True(p \supset p)$ can be shown.

The Value of Resolution

I do not mean to give the impression that reasoning with non-Horn clauses is unnecessary in general, but rather that there are other possibilities in the case of reasoning about properties of logic programs. Nor do I mean to suggest that all knowledge representation problems can be reduced to representation by means of logic programs. On the contrary, non-Horn clauses can be used to represent integrity constraints, including the rules I mention at the end of my commentary on Chapter 6, and resolution is useful for reasoning with such clauses.

Resolution and its refinements played an important role in the development of logic programming [Kowalski, 2014]. One of the main reasons for this is its use of unification (called "matching" in the book) to avoid the potentially infinite number

of applications of the rule of universal instantiation (from $\forall x\ p(x)$ derive $p(t)$ for any term t). Another is its avoidance of the thinning rule (from p derive $p \vee q$ for any sentence q). The thinning rule is responsible for the fact that in classical logic it is possible to derive any conclusion from an inconsistent set of sentences (because the thinning rule can be used to derive any sentence q, equivalent to $false \vee q$ from $false$).

Resolution is inconsistency tolerant. For example, given the inconsistent set of sentences $\{p,\ \leftarrow p\}$, resolution can derive only the empty clause $false$. However, the refutation completeness of resolution means that $false$ can also be derived from the set of clauses $\{p,\ \leftarrow p,\ \leftarrow q\}$, which shows that $\{p,\ \leftarrow p\}$ logically implies q. But from a pragmatic point-of-view, inspecting the resolution refutation, it is obvious that q does not play any role in the refutation, and therefore is not relevant to the proof of inconsistency. Such analysis of proofs of inconsistency is an important part of knowledge assimilation, the topic of Chapter 13.

Linear resolution (or backward reasoning) is even better. Given the same set of clauses $\{p,\ \leftarrow p,\ \leftarrow q\}$, and treating the clause $\leftarrow q$ as a goal, linear resolution fails to generate a refutation, and therefore to solve the goal, even though linear resolution and its various refinements are refutation complete.

A Semantic Tree Proof of the Completeness of Resolution

Before completing the commentary on this chapter, let's see how easy it is to prove the refutation completeness of resolution, using the semantic tree method described in my commentary on Chapter 1:

> Assume that the set of clauses S is inconsistent, and therefore has no models. Let $A_1, \ldots A_n, \ldots$ be an enumeration of the Herbrand base of S. Then the generation of the corresponding semantic tree terminates in a finite number of steps with a finite subtree T all of whose leaf nodes are failure nodes. The proof that there exists a resolution refutation of S is by induction on the number of nodes m in T.
>
> If $m = 1$, then, because the only clause that is false at the root of T is the empty clause, S must contain the empty clause, and proof by resolution terminates in 0 steps.
>
> Otherwise, there must exist a node N in T with exactly two children, both of which are failure nodes. (If not, then T would contain an infinite branch.) Therefore there exist clauses C_1 and C_2 in S that are falsified at the two children of N. It is easy to see that the resolvent C of the two clauses is falsified at the node N, and possibly already falsified at some earlier node on the branch to N. Therefore, there exists some smaller subtree T' of T all of whose leaves falsify some clause in $S \cup C$. By induction hypothesis, there exists a resolution refutation of $S \cup C$. Adding to this refutation the derivation of the resolvent C of C_1 and C_2 produces a resolution refutation of S.

Combining the semantic tree procedure with the addition of judiciously chosen resolvents of the kind involved in the completeness proof is similar to adding "lemma clauses" to SAT solvers based on the DPLL method [Marques-Silva and Sakallah, 1999].

CHAPTER 8

Unfortunately, except for the Horn clause case [Smolka, 1982], none of the more general completeness proofs referred to at the end of the Chapter have held up. Moreover, Eisinger [1986] constructed an ingenious counterexample that shows that the proof procedure is incomplete for a natural notion of *fair* selection of links, which is intended to ensure that every link (or descendant of a link) will eventually be selected.

The history of the various attempts to prove completeness and the problems they have encountered is discussed by Siekmann and Wrightson [2002]. However, it is still not known whether the proof procedure is complete for some other notion of fairness, or some further restrictions on the deletion of clauses or the inheritance of links.

Connection Graphs and the Web of Belief

In my 2011 book, I argue that connection graphs can be viewed as a concrete realisation of the web of belief, proposed by the philosopher W. V. O. Quine [1953], as a metaphor for the way that individual statements are connected in scientific theories and human beliefs. Quine appealed to the web of belief to support his argument against the analytic-synthetic distinction. Analytic truths are supposed to be true simply by virtue of their meaning, independent of matters of fact. Synthetic truths are true because of their correspondence with reality.

Quine argued that, if a set of beliefs is contradicted by observations, then any belief in the web of beliefs can be revised to restore consistency. From this point of view, there is no distinction between analytic statements and synthetic ones. The argument not only attacks the analytic-synthetic distinction, but it also attacks the modal distinction between necessary and possible truths, which was the original inspiration for modal logic. Quine's argument had a big influence on my own views about modal logic. (Note that databases and ALP also make a similar distinction between data, which is possibly true, and integrity constraints, which are necessarily true. However, they make this distinction without the use of modal logic.)

Connection Graphs and Action Networks

Connection graphs are also related to the action networks of [Maes, 1990], in which nodes are actions, and one action node A_1 is positively connected to another action node A_2 if A_1 initiates one of the preconditions of A_2. Action nodes have levels of

activation, which are input both from the facts that are observed in the current situation and from the goals that are currently active. These activation levels are propagated through the network. When the activation level of an action reaches a given threshold, then it is executed if all its preconditions hold in the current situation. The resulting behaviour is both situation-sensitive, goal-oriented, and similar to that of neural networks.

Ignoring frame axioms and negative connections due to one action terminating a precondition of another action, Maes-like action networks can be represented by connection graphs containing clauses:

$$Holds(F, t+1) \leftarrow Happens(A, t), Holds(F_1, t), \ldots, Holds(F_n, t)$$

which represent that an action A initiates F if the preconditions $F_1, \ldots, F_n$ of A all hold. In addition, the graph contains clauses $Holds(F, T)$ representing the facts F that hold at the current time T, and goal clauses $\leftarrow Holds(F_1, t), \ldots, Holds(F_n, t)$ representing goals that are meant to hold at some future time t.

Representing action networks in such connection graph form suggests how activation levels can be used in connection graphs more generally: Initial activation levels can be associated with both current observations and current goals, can be propagated from one clause to another, and activation levels can be used to guide the selection of links connected to the most active clauses.

Moreover, links can be assigned weights (or utilities) that are learned by keeping track of how often they have contributed to the solution of goals in the past (as suggested in Chapter 13). The more often a link has contributed to a solution, the greater its weight. The propagation of activation levels from clause to clause can then be adjusted by the weights associated with the links connecting those clauses. I like to think that the resulting control strategy resembles the way that neurons are activated in connectionist models of the mind.

CHAPTER 9

To the best of my knowledge, there has been little further work on applying difference analysis to logic programs. Probably the most important work in this area has been the identification by Apt and Bezem [1991] of acyclic logic programs, which are guaranteed to reduce differences and terminate for a large class of goals.

The Semantics of Integrity Constraints

Much more work has taken place on goal transformation and on integrity constraints more generally. At the time I was writing the book, the notion of integrity constraint for logic programs (and databases) was only just emerging and was not well understood. In this chapter, I considered that a program or database P satisfies an integrity constraint or program property I if I is consistent with P and together with P implies no ground atoms not already implied by P.

It would have been simpler to say that I is a logical consequence of P. But this simpler alternative does not capture the intuition, for example, that transitivity is a property of the $\leq$ relation defined by:

$0 \leq x$
$s(x) \leq s(y) \leftarrow x \leq y$

It is not possible to prove transitivity of $\leq$ in first-order logic without the only-if half of the program and axioms of induction.

The notion of integrity constraint proposed in this chapter captures the intuition, but in a round-about way. However, it fails with the simple example:

$P = \{R(A)\}$
$I = \{R(x) \rightarrow S(x) \vee T(x)\}$

According to the proposal in this chapter, program P satisfies the integrity constraint I. But intuitively, this is not correct. For I to be a property of the program, P should include or imply an additional atom, either $S(A)$ or $T(A)$.

As I mention in the commentary to Chapter 6, I now believe that the correct definition of what it means for an integrity constraint to be satisfied by a program P is that I is true in the canonical model of P. If P is a set of Horn clauses, then the canonical model of P is the minimal model of P. In the commentary on Chapter 11, I discuss the notion of canonical model for programs with negative conditions.

Constraint Logic Programming

Integrity constraints have many applications, including their use for goal transformation as described in this chapter, as well as their use for constraining abductive solutions in ALP. They are also related to the constraint satisfaction methods used in constraint logic programming (CLP).

In both ALP and most versions of CLP, there are two kinds of predicates: predicates defined by a logic program P, and additional predicates that can occur in the bodies of clauses in P, but not in their heads. These additional predicates are abducible predicates in ALP, and constraint predicates in CLP. In both cases, given a goal G, the computational task is to generate a set (or conjunction) C of conditions (abducible or constraint atoms) such that:

C solves the initial goal G and
C is satisfiable.

Proof procedures for CLP are similar to those for ALP. In both cases, the proof procedure uses P to reason backwards from G, incrementally generating C and incrementally testing C for satisfiability. In ALP, satisfiability is with respect to integrity constraints. In CLP it is with respect to a predefined domain D, such as the domain of real numbers with addition, multiplication, equality and inequality.

Colmerauer [1982] introduced constraints and the domain of rational trees with equality and disequality into Prolog II, mainly to justify the lack of an "occur check" in Prolog. The occur check is needed in the unification algorithm to ensure that a term, such as $f(X)$, does not unify with a subterm, such as X. The occur check is hard to implement both correctly and efficiently, and Colmerauer showed that unification without the occur check could be viewed as computation in the domain of possibly infinite, rational trees. Jaffar and Lassez [1987] introduced the CLP Scheme, in which the domain D, which defines the semantics of the constraint predicates, can be an arbitrary model-theoretic structure, and the satisfiability check can be an arbitrary algorithm.

Around the same time that the CLP Scheme was introduced, the CHIP system was developed at ECRC, the European Computer Research Centre, set up by the main European computer manufacturers in response to the Japanese FGCS project. CHIP [Dincbas et al, 1988; van Hentenryck,1989] combined logic programming with constraint propagation techniques developed to solve constraint satisfaction problems (CSP) in AI.

The survey of CLP by Jaffar and Maher [1994] describes the domain constraints of CHIP in terms of CLP Schemes. Operationally, domain constraints in CHIP are predicates that restrict the values of variables, which are further restricted by constraint propagation. For example, given the program and goal:

$Likes(John, x) \leftarrow x \in \{Logic, Wine, Mary\}$
$Person(x) \leftarrow x \in \{Bob, John, Mary\}$
$\leftarrow Likes(John, x), Person(x)$

backward reasoning combined with constraint propagation derives the solution $x \in \{Mary\}$, i.e. $x = Mary$, without any search. In contrast, Prolog would represent the two clauses of the constraint logic program as six facts, and would use backtracking search, failing eight times and succeeding only on the ninth and final try.

Given the similarities between proof procedures for ALP and CLP, it is tempting to try to unify them. Such an attempt was made in [Kowalski et al, 1998], treating integrity constraints I as approximations to relations, whether they are defined by a logic program P or by a built-in domain D. In the model-theoretic semantics, the integrity constraints I must be true in the canonical model defined by $P \cup D$. (This characterisation exploits the fact that Herbrand interpretations lead a dual life: D is both a model-theoretic structure and a component of the logic program $P \cup D$.) In the proof procedure, integrity checking incrementally checks candidate solutions C for satisfiability.

The unified framework includes not only ALP and CLP, but also goal transformation, as discussed in this chapter. It was another strand of work that also contributed to my more recent work with Fariba Sadri on LPS. In [Kowalski, 2006], I also argued that using goal transformation with integrity constraints is similar to programming with aspects in aspect-oriented programming [Kiczales et al, 1997].

Constraint Handling Rules

The unified framework mentioned above is similar to the general-purpose language CHR [Frühwirth; 1998, 2009], in which a constraint solver is programmed explicitly by means of constraint handling rules. There are two kinds of such rules: equivalences and propagation rules. Equivalences can be used like logic programs written in if-and-only-if form, but they can also be used to simplify conjunctions of constraints. Propagation rules are logical implications that are used to reason forward, like production rules - or like integrity constraints that are used to check updates for consistency.

Here is the classic introductory CHR example, implementing a constraint solver for partial ordering, written in the syntax of the book:

$x \leq x \leftrightarrow true$
$x \leq y,\ y \leq x \leftrightarrow x = y$
$x \leq y,\ y \leq z \rightarrow x \leq z$
$x \leq y \setminus x \leq y \leftrightarrow true$

The first two rules are equivalences, which implement reflexivity and antisymmetry, and are used as simplification rules. The third rule is a propagation rule, which implements transitivity, adding the consequent of the rule when the antecedent is satisfied. The fourth rule, which implements idempotency, is logically an equivalence (in which $\setminus$ is read as *and*), but operationally it combines simplification and propagation, by removing a duplicate constraint (before the $\setminus$), retaining only one copy. Given the goal $A \leq B, C \leq A, B \leq C$, CHR computes the solution $A = B, B = C$.

The semantics of CHR is defined in terms of logical implication: Given a CHR program P and a goal clause G, a solution of G is a conjunction of constraints C to which no inferences can be applied, such that

P logically implies $\forall[G \leftrightarrow C]$, where
$\forall$ represents the universal quantification of all the variables in G.

[Abdennadher and Schütz, 1998] developed an extension $CHR^{\vee}$ of CHR to include disjunctions on the right hand side of rules. The resulting variant of CHR can implement both top-down and bottom-up reasoning with Horn clauses. It can also implement abduction with disjunctive integrity constraints, as the following implementation of the map colouring problem shows:

$Country(x) \rightarrow Colour(x, Red); Colour(x, Yellow); Colour(x, Green)$
$Colour(x, c), Colour(y, c), Adjacent(x, y) \leftrightarrow\ false$
The goal clause is a conjunction of all the facts concerning the predicates *Country* and *Adjacent*.

Here ; represents disjunction.

Like production systems and concurrent logic programming languages, CHR is a committed choice language, in which all non-determinism is of the "don't care"

variety. $CHR^{\vee}$ retains this feature of CHR. However, it achieves the effect of "don't know" non-determinism by representing disjunctions explicitly.

CHAPTER 10

Most of this chapter is a fairly orthodox treatment of the relationship between the standard form of logic and clausal form. The last few pages touch upon the relationship between standard form, clausal form and logic programming form. As in Chapter 7, I discuss the representation of the only-if halves of logic programs.

As mentioned in earlier commentaries, and as I will discuss again in greater detail in the commentary on Chapter 11, other views of logic programming and other extensions of Horn clauses have become established in the last forty years. The most important extension is the extension to normal logic programs, which can have negative conditions. Closely related to this, is the extension to *generalised logic programs*, which can have conditions that are arbitrary formulas of FOL.

In fact, all of the examples on page 203 are examples of this generalised logic programming form. On page 219 in Chapter 11, I show how to transform the definition of subset on page 204 into a normal logic program, using an auxiliary predicate. The notion of generalised logic programs and their transformation into normal logic programs using auxiliary predicates were defined more generally and more formally by Lloyd and Topor [1984].

The Syntactic Form of Reactive Rules

In this chapter, I give examples where the standard form of logic is more natural than clausal form. But I argue that "what is needed is not full unrestricted standard form but a limited extension of clausal form". Most of the examples, like the definition of subset, show that in many cases what we need is an extension of Horn clause programs to generalised logic programs with conditions that are universally quantified implications. However, I did not appreciate then that we also need a different extension of clausal form to more naturally represent reactive rules and other kinds of integrity constraints. In [Kowalski and Sadri, 2014], we argue that such reactive rules are naturally represented as sentences of the form:

$$\forall x[antecedent \rightarrow \exists y[consequent_1 \vee \ldots \vee consequent_n]]$$

where *antecedent* and the disjuncts $consequent_i$ are conjunctions.

Here is a typical example in semi-formal notation (from [Kowalski and Sadri, 2012]):

$$[heat\ sensor\ detects\ high\ temperature\ at\ time\ t1\ \wedge$$
$$smoke\ detector\ detects\ smoke\ at\ time\ t2\ \wedge$$
$$|t1 - t2| \leq 60\ sec\ \wedge\ max(t1, t2, t)] \rightarrow$$
$$[activate\ sprinkler\ at\ time\ \ t3\ \wedge\ \ t < t3\ \leq t + \ 10\ sec\ \wedge$$

$$[\textit{send security guard at time } t4 \ \wedge \ t3 < t4 \ \leq t3 + 30 \ \textit{sec}] \vee$$
$$[\textit{call fire department at time } t5 \wedge \ \ t < t5 \ \leq t + \ 120 \ \textit{sec}]$$

The antecedent of the rule recognises the likelihood of a fire when a heat sensor detects high temperature and a smoke detector detects smoke within 60 seconds of one another. The consequent of the rule provides two alternative ways of dealing with the possible fire. The first is to activate the sprinkler and then send a security guard to investigate, within 10 + 30 = 40 seconds of recognising the possible fire. The second is to call the fire department within 120 seconds. The reactive rule can be made true, when the antecedent is made true, in either of the two alternative ways. Because of the timing constraints, the first alternative can be tried first, leaving enough time to try the second alternative if the first alternative fails.

As in the case of reactive rules in general, because all variables in the antecedent are universally quantified with scope the entire rule, and all variables in the consequent are existentially quantified with scope the consequent, it is convenient to omit the quantifiers and to let the quantification be implicit.

CHAPTER 11

At the time I was writing the book, I was familiar with the notion of explicit definition in mathematical logic, but I did not regard it as relevant. According to this notion, given a set S of sentences in FOL and a predicate symbol P with n arguments, a formula F not containing the symbol P *defines* P if and only if:

$$S \vdash_{FOL} \forall x_1, \ldots, x_n [P(x_1, \ldots, x_n) \leftrightarrow F(x_1, \ldots, x_n)]$$

Here $\vdash_{FOL}$ can represent either provability or logical implication, which are equivalent in FOL.

Logic Programs as Definitions

The notion of explicit definition is not relevant for our purposes, because it excludes recursive definitions, in which the formula F contains P. However, I did not know then that there is an alternative notion of definition in mathematical logic that applies to recursive relations. According to that notion, the recursive relation that is the transitive closure of a binary relation cannot be defined in FOL. In particular, the Horn clauses that "define" the *Ancestor* relation are not a definition in FOL, because there exist (non-minimal) models in which the *Ancestor* relation is not the transitive closure of the *Parent* relation.

In the book, I used the term *definition* informally, having in mind the notion in [van Emden and Kowalski, 1976], according to which, given a set S of sentences in FOL and a predicate symbol P in S, the relation corresponding to P "defined" by S is represented by the set of all ground atoms $P(t_1, \ldots, t_n)$ such that:

$$S \vdash_{FOL} P(t_1, \ldots, t_n).$$

For definite clause programs S, this is equivalent to $S \vdash_{Horn} P(t_1, \ldots, t_n)$, where $S \vdash_{Horn} A$ means that A is derivable from S using only the one rule of inference:

> From C in S and ground atoms $A_1, ..., A_n$, derive A_0, where $A_0 \leftarrow A_1, ..., A_n$ is a ground instance of C.

The more general notion of definition, $S \vdash_{FOL} P(t_1, \ldots, t_n)$, works perfectly when S is a definite clause program, but not when S is a normal logic program with negation. For example, for $S = \{P(A) \leftarrow \mathit{not}\ P(B)\}$, provability understood as $\vdash_{FOL}$ gives the empty relation, rather than the intuitively correct relation represented by $\{P(A)\}$.

To obtain the intuitive result for normal logic programs, we need to augment S with the only-if halves of definitions. In this example, if we interpret *only-if* and *not* in the object language, we obtain the augmented set of sentences $S^* = \{P(x) \leftrightarrow x = A \ \wedge \ \mathit{not}\ P(B)\} \ \cup \ CET$, where CET is the "Clark Equality Theory", containing such axioms of equality as $x = x$ and $\mathit{not}\ B = A$. The desired relation $\{P(A)\}$ is the set of all the ground atoms $P(t)$ such that $S^* \vdash_{FOL} P(t)$.

Stable Model Semantics

We can also obtain the same intuitively correct result, if we interpret a negative ground literal $\mathit{not}\ Q$ as an assertion in the meta-language stating that Q cannot be shown using $\vdash_{Horn}$. For this purpose, we need to augment S with a set of correct and complete "auto-epistemic" assertions Δ about what cannot be shown, treating Δ as a set of ground atoms, and therefore treating $S \cup \Delta$ as a definite clause program. With this interpretation, the relation corresponding to a predicate symbol P defined by S is represented by the set of all ground atoms $P(t_1, \ldots, t_n)$ such that:

$$S \cup \Delta \vdash_{Horn} P(t_1, \ldots, t_n) \text{ where } \Delta = \{\mathit{not}\ Q \mid S \cup \Delta \nvdash_{Horn} Q\}.$$

It is easy to see that this meta-level interpretation of the relations defined by a normal logic program is equivalent to the stable model semantics of Gelfond and Lifschitz [1987]. It is also equivalent to the interpretation of negative ground literals as abducible predicates in ALP [Eshghi and Kowalski, 1987].

However, not every normal logic program can be understood naturally as a definition. For example, both the object-level and meta-level interpretations of *not* apply to the program:

$P \leftarrow \mathit{not}\ Q$
$Q \leftarrow \mathit{not}\ P$

But they can hardly be said to interpret the program as a definition of P and Q.

Stratified Logic Programs and the Perfect Model Semantics

The notion of "definition" in the book was applied mainly to Horn clause programs. However, I also gave a non-Horn logic program $Clear(y) \leftarrow not\ \exists x\ On(x, y)$ defining the predicate *Clear* in terms of the predicate *On*. The definition can be understood as a simple example of a stratified logic program [van Gelder, 1989; Apt et al, 1988]. Here is another example, consisting of a lower stratum E defining a network of nodes, some of whose links may be broken, and a higher stratum I defining when two nodes are connected:

E: $Link(A, B)$ $Link(A, C)$ $Link(B, C)$ $Broken(A, C)$

I: $Connected(x, y) \leftarrow Link(x, y),\ not\ Broken(x, y)$
$Connected(x, y) \leftarrow Connected(x, z),\ Connected(z, y)$

The Horn clauses in E *extensionally* define the predicates in E by means of a definite clause program consisting of ground atoms. The minimal model of E is trivially also E. The clauses in I *intensionally* define the predicate *Connected* in terms of the extensional predicates in E. In particular, the conditions of the first clause in I are all defined in the lower stratum E, and they can be evaluated in the minimal model E. This results in a set $E \cup I'$ of Horn clauses, which intuitively define the same predicates as the original non-Horn clause program $E \cup I$:

I': $Connected(A, B)$ $Connected(B, C)$
$Connected(x, y) \leftarrow Connected(x, z),\ Connected(z, y)$

The minimal model M of this set $E \cup I'$ of Horn clauses is called the *perfect model* of $E \cup I$, and is intuitively the natural, intended model of $E \cup I$:

M: $Link(A, B)$ $Link(A, C)$ $Link(B, C)$ $Broken(A, C)$
$Connected(A, B)$ $Connected(B, C)$ $Connected(A, C)$

(Notice that A is connected to C, even though the direct link from A to C is broken.)

This construction of the perfect model can be generalised to the construction of the *perfect model* $M = M_0 \cup \ldots \cup M_n$ of a stratified logic program $S = S_0 \cup \ldots \cup S_n$, where the negative literals in clauses belonging to a higher stratum S_i are all defined in lower strata S_j where $j < i$:

> M_0 is the minimal model of the Horn clause program S_0.
> M_{i+1} is the minimal model of the Horn clause program $S'_{i+1} \cup M_i$,
> where S'_{i+1} is obtained by using M_i to evaluate the conditions in S_{i+1}
> that are already defined in $S_0 \cup \ldots \cup S_i$.

Przymusinski [1988] extended the perfect model construction from stratified programs to *locally stratified* programs S, by treating distinct atoms in the program $ground(S)$ as though they were distinct predicates. Here is a simple example of a locally stratified program E defining the predicate *Even*:

E: $\quad Even(0) \qquad Even(s(x)) \leftarrow not\ Even(x)$

The program $ground(E)$ can be partitioned into a countably infinite number of strata $ground(E) = \cup_{i<\omega}\ E_i$ where:

E_0: $\quad Even(0)$

E_{i+1}: $\quad Even(s^{i+1}(0)) \leftarrow not\ Even(s^i(0))$

The perfect model of E is the limit $\cup_{i<\omega} M_i = \{Even(0), Even(s(s(0))), \ldots\}$ where:

> M_0 is the minimal model of the Horn clause program E_0.
> M_{i+1} is the minimal model of the Horn clause program $E'_{i+1} \cup M_i$, where E'_{i+1} is obtained from E_{i+1} by using M_i to evaluate the negative condition $not\ Even(s^i(0))$.

The Well-founded Semantics

The perfect model semantics formalises the intuitive notion of inductive (or recursive) definition in many cases. However, [Denecker *et al.*, 2001] argue that it was only a step in the development of the well-founded semantics of [Van Gelder *et al.*, 1991], which "provides a more general and more robust formalization of the principle of iterated inductive definition that applies beyond the stratified case."

The well-founded semantics can be viewed as a generalisation of the perfect model semantics in which the stratification is determined dynamically, and the submodels M_i are three-valued models represented by sets of ground literals. A ground atom A is true in M_i if $A \in M_i$, false if $not\ A \in M_i$, and undefined if neither $A \in M_i$ nor $not\ A \in M_i$. Every normal logic program, whether it is stratified or not, and whether it has a natural interpretation as a definition or not, has a unique well-founded model.

For example, the program $\{P \leftarrow not\ Q, \quad Q \leftarrow not\ P\}$, which has two stable models $\{P\}$ and $\{Q\}$ and no perfect models, has the unique well-founded model $M = \emptyset$, in which P and Q are both undefined.

For another example, consider the following program Ev, which intuitively defines the same predicate $Even$ as the program E above:

Ev: $\quad Even(0) \quad Even(y) \leftarrow Successor(x, y), not\ Even(x) \quad Successor(x, s(x))$

This program does not have a perfect model, because *ground(Ev)* contains clauses, such as $Even(0) \leftarrow Successor(0, 0),\ not\ Even(0)$, that cannot be stratified. However, it does have a well-founded model $M = \cup_{i<\omega} M_i$, which is two-valued:

$M_0 = \emptyset$
$M_1 = \{Even(0), Successor(0, s(0)), \ldots, Successor(s^{i-1}(0), s^i(0)), \ldots\} \cup$
$\quad \{not\ Successor(s^j(0), s^i(0)) \mid j \neq i - 1\}$
$M_{i+1} = M_i \cup \{Even(s^i(0))\}$ if $i > 0$ is even.
$M_{i+1} = M_i \cup \{not\ Even(s^i(0))\}$ if $i > 0$ is odd.

As in the case of a stratified or locally stratified program, a positive atom A is in M_{i+1} if and only if it can be derived from the Horn clauses obtained by evaluating the conditions in *ground(Ev)* that are already true in M_i. However, a negative ground literal *not* A is in M_{i+1} if and only if every clause in *ground(Ev)* of the form $A \leftarrow$ *body* contains in its *body* a condition that is false in M_i.

In general, the *well-founded model* of a normal logic program S is the smallest three-valued interpretation M such that:

> $A \in M$ if A can be derived from the Horn clauses obtained by evaluating the conditions of clauses in *ground(S)* that are true in M.
>
> *not* $A \in M$ if every clause in *ground(S)* of the form $A \leftarrow$ *body* contains in its *body* a condition that is false in M.

The well-founded model can be generated bottom-up, starting with the empty model $\emptyset$ and repeatedly adding positive and negative atoms until the above condition holds. It can also be queried top-down using tabling [Chen and Warren, 1996].

As we have seen in this commentary, the object-level interpretation of "only-if" and the corresponding interpretation of logic programs as standing for object-level definitions in "if-and-only-if" form has largely been superseded by model-theoretic interpretations of logic programs. Moreover, the positive atoms in these models are all generated by Horn clause derivations. These Horn clause derivations give the resulting models a syntactic core, which makes these model-theoretic semantics meta-level in nature.

I have speculated elsewhere [Kowalski, 2014] on the relationship between the object-level and meta-level/model-theoretic interpretations and suggested that their relationship is similar that that between first-order axioms of arithmetic and the standard model of arithmetic.

CHAPTER 12

When I was writing this chapter, I was aware of a much simpler Horn clause definition of Horn clause provability, which I did not mention because I did not understand its logic.

Ambivalent Combination of Object Language and Meta-language

The definition:

$$Demo(p \;\&\; q) \leftarrow Demo(p), Demo(q)$$
$$Demo(p) \;\leftarrow\; Demo(p \leftarrow q), Demo(q)$$

is a notational variant of clauses $T1$ and $T9$ in Chapter 7, defining the truth predicate for propositional logic. Here the infix function symbol & names conjunction

and the infix function symbol $\leftarrow$ "ambivalently" names the implication symbol $\leftarrow$. (The symbol $\leftarrow$ is not ambiguous, because the context makes its intended interpretation clear.)

With this representation of the proof predicate, it suffices to represent the problem of the fallible Greek simply by the additional clauses:

$\leftarrow Demo(Fallible(x)\ \&\ Greek(x))$
$Demo(Fallible(x) \leftarrow Human(x))$
$Demo(Human(Turing))$
$Demo(Human(Socrates))$
$Demo(Greek(Socrates))$

Notice that, in first-order logic, $Fallible$, $Greek$ and $Human$ are function symbols, and not predicate symbols. However, it is possible to combine these clauses with additional clauses, in which the same symbols are used ambivalently as predicate symbols - for example, by adding such clauses as:

$Fallible(x) \leftarrow Human(x)$
$Human(Turing)$
$Human(Socrates)$
$Greek(Socrates)$
$Greek(Aristotle)$

The resulting ambivalent amalgamation of object-language and meta-language resembles a modal logic, in which the $Demo$ predicate behaves like a modal operator representing belief. In this example, the combined set of clauses represents a situation in which $Aristotle$ is $Greek$, but is not known or believed to be $Greek$.

I have investigated different representations of the wise man puzzle, using such an amalgamation of object language and meta-language, in which the $Demo$ predicate represents an agent's beliefs. The representation in [Kowalski and Kim, 1991] uses an explicit naming convention like the one presented in this chapter, whereas the representation in [Kowalski, 2011] uses the kind of ambivalent syntax, which resembles modal logic, illustrated in the example above. The ambivalent syntax is much simpler and more natural.

The ambivalent syntax has also been used for practical applications, since the earliest days of Prolog. However, a common problem with the logic of many of these applications is that they often use a meta-predicate $Var(t)$, to test whether a term t is a variable. This predicate holds, as intended, when t truly is a variable. Logically, like any other predicate containing variables, it should continue to hold if the variable t is later instantiated to a non-variable. But this is not the correct, intended interpretation of the meta-predicate $Var(t)$.

The problem is due to the ambiguity of the implicit quantification of variables in clauses such as $Demo(Fallible(x) \leftarrow Human(x))$. It is tempting to interpret the clause as standing for $Demo(\text{“}\forall x(Fallible(x) \leftarrow Human(x))\text{”})$. But the status of x in this interpretation is problematic. Is it an object-level variable, a meta-variable that ranges over object-level variables, or a meta-constant that names

an object-level variable? It depends in part on how the quotation marks are implemented.

For Horn clause provability, the problem goes away if clauses with variables are understood as standing for all their variable-free instances. In that case, x can be understood as a meta-variable ranging over object-level terms t, and the clause can be interpreted as standing for the sentence $\forall x\ Demo(Fallible(x) \leftarrow Human(x))$. In this interpretation, the single meta-level sentence states there are possibly infinitely many object level sentences of the form $Fallible(t) \leftarrow Human(t)$ for all ground terms t. To be sensible, of course, the instantiation of the variable x by terms t needs to be restricted to terms of the appropriate sort, as in many-sorted logic.

The problems of the semantics of the ambivalent combination of object language and meta-language has a complex history, which is recounted in [Costantini, 2002]. It is still not obvious to me whether all of the problems have been fully resolved.

CHAPTER 13

This chapter was my first step in the attempt to show how an intelligent agent can use logic to guide its interactions with the world.

From Information Systems to Intelligent Agents

At the time I was working on the book, I was not fully aware of the importance of actions in helping an agent to achieve its goals. I addressed this broader use of logic in [Kowalski, 1988], which was a response to Carl Hewitt's [1985] advocacy of open systems as a challenge to logic programming. In this broader use of logic as an agent's language of thought, logic programs represent the agent's beliefs and integrity constraints represent the agent's goals.

As I put it more generally in [Kowalski, 2011]:

> In Artificial Intelligence, an agent is any entity, embedded in a real or artificial world, that can observe the changing world and perform actions on the world to maintain itself in a harmonious relationship with the world. Computational logic, as used in artificial intelligence, is the agent's language of thought. Sentences expressed in this language represent the agent's beliefs about the world as it is and its goals for the way it would like it to be. The agent uses its goals and beliefs to control its behaviour.

In recent years, the notion of intelligent agent has become a unifying theme of much research in AI, and showing how computational logic can be used to model the thinking of an intelligent agent has become the driving force of most of my own research.

One of the main challenges of AI has been to reconcile the notion of a deliberative agent, which uses its beliefs to achieve its goals, with the contrary notion

of a reactive agent, which acts "instinctively" in response to situations that arise in its environment. In [Kowalski, 1995], I argued that computational logic can reconcile and combine the characteristics of both deliberative and reactive agents. In [Kowalski and Sadri; 1996, 1999, 2009], we used abductive logic programming to develop in greater detail an agent model in which beliefs are represented by logic programs, goals are represented by integrity constraints, and actions are represented by abducible predicates. We argued that this combination of logic programs and integrity constraints can model both deliberative and reactive behaviour. More recently, we developed LPS [Kowalski and Sadri; 2011, 2012, 2014a, 2014b], as a scaled down version of the agent model, using destructive change of state for practical database and programming applications.

Abduction

In this chapter, I also explored the use of abduction to generate hypotheses, and to reason by means of defaults. The use of abduction for default reasoning was developed in greater detail in Theorist [Poole et al, 1987; Poole, 1988]. In Theorist, abduction extends a first-order clausal theory T with assumptions Δ from a set of candidate hypotheses A, restricted by a set of first-order constraints I. The implementation of Theorist used a combination of backward reasoning from G to generate candidate Δ, and a refutation procedure to show that $T \cup \Delta \cup I$ is not inconsistent (similar to negation as failure). Theorist was one of the main inspirations of abductive logic programming [Eshghi and Kowalski, 1989].

Argumentation

Building on Phan Minh Dung [1991], [Kakas et al, 1992] showed that the abductive proof procedure in [Eshghi and Kowalski, 1989] could be given an argumentation interpretation, in which, given a logic program P and set of negative literals Δ, a derivation $P \cup \Delta \vdash_{Horn} G$ is interpreted as an *argument* for G supported by the assumptions Δ. With this interpretation, $argument_1$ for G attacks $argument_2$ if *not* G is in the set of assumptions Δ supporting $argument_2$. A set of arguments $Args$ is *admissible*, if for every $argument_1$ that attacks an $argument_2 \in Args$ there is an $argument_3 \in Args$ that attacks $argument_1$. In other words, a set of arguments $Args$ is *admissible* if it can counter-attack every attack.

Dung [1993, 1995] generalized this interpretation and developed an abstract argumentation theory, which has had a wide-ranging impact. [Bondarenko et al, 1997] developed a more concrete assumption-based version of Dung's abstract theory, and showed that it could be used to reconstruct not only most logic programming semantics, but also most semantics developed for non-monotonic reasoning.

Inductive Logic Programming

Another important development, related to the topic of this chapter, is inductive logic programming (ILP), which is concerned with generating inductive generali-

sations of examples. ILP was introduced by Muggleton [1991] building upon the inverse unification of [Plotkin, 1970]. Given a background theory B represented as a logic program, a set E^+ of positive examples, and a set E^- of negative examples, the inductive task in ILP is to generate a set of hypotheses H represented by a logic program such that:

$H \cup B$ covers E^+ and
$H \cup B$ does not cover any of the negative examples in E^-.

Informally speaking, H solves the inductive task if H explains all the positive examples, without "implying" any of the negative examples.

There is an obvious parallel here with the definition of abduction, with H being the parallel of abductive assumptions, E^+ the parallel of the goal to be solved, and E^- the parallel of integrity constraints to be satisfied. A number of frameworks have been developed (e.g. [Ade and Denecker, 1995; Flach and Kakas, 2000; Lamma et al, 1999]), combining ILP and ALP, and exploiting this parallelism.

As in the case of the vague (or abstract) specification of the abductive task, the abstract notion of "covering" also admits several different concrete instances.

In recent years, the field of ILP has been extended to incorporate the representation of probability. The recent survey of ILP [Muggleton et al, 2012] suggests that this extension of ILP can function as a foundation for much of AI, both helping to understand existing research and to serve as a vision for the future. To this extension of ILP with probability, I would add its embedding in the dynamic setting of an intelligent agent interacting with the world, observing changes and performing actions to satisfy its goals. I like to believe that the use of probabilistic ILP in such a setting, to generalise observations and to learn new beliefs together with their probabilities and their utilities, will play a central role in the future, both to develop more intelligent artificial agents and to build better cognitive models of human agents.

ACKNOWLEDGEMENTS

Many thanks to Maarten van Emden, Thom Frühwirth and Fariba Sadri for their helpful comments. Special thanks to Thom for inviting me to republish the book with this commentary, and for his efforts in converting the book into its present form.

NEW REFERENCES

Ade, H. and Denecker, M. (1995). AILP: Abductive Inductive Logic Programming. *IJCAI*, 1201-1209.

Abdennadher, S. and Schütz, H. (1998). $CHR^{\vee}$: A Flexible Query Language. In T. Andreasen, H. Christiansen, and H.L. Larsen, editors, *FQAS '98: Proc. 3rd*

Intl. Conf. on Flexible Query Answering Systems, Volume 1495 of Lecture Notes in Artificial Intelligence, 1-14, Springer-Verlag.

Apt, K. R, and Bezem, M. (1991). Acyclic Programs. *New Generation Computing*, 9(3-4), 335-363.

Apt, K. R, Blair, H. and Walker, A. (1988). Towards a Theory of Declarative Knowledge. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, 89-148. Morgan Kaufman, Los Altos, CA.

Baader, F. (Ed.). (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Baumgartner, P. (2000). FDPLL - a First-order Davis-Putnam-Logeman-Loveland Procedure. In *Automated Deduction, CADE-17* 200-219. Springer Berlin Heidelberg.

Biere, A., Heule, M., van Maaren, H. and Walsh, T. (2009). *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bondarenko, A., Dung, P. M., Kowalski, R., and Toni, F. (1997). An Abstract Argumentation-theoretic Approach to Default Reasoning. *Journal of Artificial Intelligence*, 93 (1-2), 63-101.

Bonner, A. J., and Kifer, M. (1993). Transaction Logic Programming. In *Proceedings of the International Conference on Logic Programming*, 257-279.

Bowen, K. A. and Kowalski, R. (1982). Amalgamating Language and Metalanguage. In *Logic Programming*. (Clark and Tarnlund, eds.) Academic Press, 153-172.

Brewka, G., Eiter, T., and Truszczyski, M. (2011). Answer Set Programming at a Glance. *Communications of the ACM*, 54(12), 92-103.

Brough, D. R. and Walker, A. (1984). Some Practical Properties of Logic Programming Interpreters. In *Proceedings of FGCS*, 149-156.

Burton A., Kowalski, R., Gacic-Dobo M., Karimov R. and Brown, D. (2012). A Formal Representation of the WHO and UNICEF Estimates of National Immunization Coverage: a Computational Logic Approach. In *PLOS ONE*, October 25, 2012, Online at http://dx.plos.org/10.1371/journal.pone.0047806

Ceri, S., Gottlob, G. and Tanca, L. (1989). What You Always Wanted to Know about Datalog (and Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 146-166.

Chen, W. and Warren, D. (2012) Tabled Evaluation with Delaying for General Logic Programs. *JACM* 43, 20-74.

Clark, K. L. and McCabe, F. G. (1979). The Control Facilities of IC-Prolog. In *Expert Systems in the Electronic Age* (Ed. D. Michie), Edinburgh University Press.

Clark, K. L. and Gregory, S. (1981). A Relational Language for Parallel Programming. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, 171-178. ACM.

Clark, K. L., McCabe, F. G., and Gregory, S. (1982). IC-Prolog Language Features. *Logic programming* (Eds. Clark and Tarnlund) Academic Press, 254-

266.

Clark, K. and Gregory, S. (1986). Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1), 1-49.

Colmerauer, A. (1982). Prolog II: Reference Manual and Theoretical Model. *Groupe D'intelligence Artificielle, Faculté Des Sciences De Luminy, Marseille.*

Costantini, S. (2002). Meta-reasoning: a Survey. In *Computational Logic: Logic Programming and Beyond*, 253-288. Springer Berlin Heidelberg.

Davis, M., Logemann, G., Loveland, D. (1962). A Machine Program for Theorem-Proving. *Communications of the ACM* 5 (7): 394397.

Denecker, M. (2000). Extending Classical Logic with Inductive Definitions. In *Computational Logic - CL 2000* 703-717. Springer Berlin Heidelberg.

Denecker, M. and Kakas, A. (2002). Abduction in Logic Programming. In *Computational Logic: Logic Programming and Beyond*, 402-436. Springer Berlin Heidelberg.

Denecker, M., Bruynooghe, M. and V. Marek. (2001). Logic Programming Revisited: Logic Programs as Inductive Definitions. *ACM Transactions on Computational Logic*, 2(4), 623-654.

Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T. and Berthier, F. (1988). The Constraint Logic Programming Language CHIP. In *FGCS*, 693-702.

Dung P. M. (1991). Negation as Hypothesis: an Abductive Foundation for Logic Programming. In *Proc. 8th International Conference on Logic Programming*, MIT Press.

Dung P. M. (1993). On the Acceptability of Arguments and its Fundamental Roles in Nonmonotonic Reasoning and Logic Programming. In *Proceedings of IJCAI*, 852-857, Morgan Kaufmann.

Dung, P. M. (1995). On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and N-person Games. *Artificial intelligence*, 77(2), 321-357.

Eisinger, N. (1986). What You Always Wanted to Know about Clause Graph Resolution. In *8th International Conference on Automated Deduction*, 316-336. Springer Berlin Heidelberg.

van Emden, M. and Kowalski, R. (1976). The Semantics of Predicate Logic as a Programming Language. *JACM*, 23 (4), 733-742. Earlier version DCL Memo. School of Artificial Intelligence, University of Edinburgh, 1974.

Eshghi, K. (1988). Abductive Planning with Event Calculus. In *ICLP/SLP*, 562-579.

Fung, T. H. and Kowalski, R. (1997). The IFF Proof Procedure for Abductive Logic Programming. *Journal of Logic Programming.*

Eshghi, K. and Kowalski, R. (1989) Abduction Compared with Negation by Failure, In *Sixth International Conference on Logic Programming*, (eds. G. Levi and M. Martelli) MIT Press, 234-254.

Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. and Welty, C. (2010). Building Watson: An Overview of the DeepQA Project. *AI*

magazine, 31(3), 59-79.

Flach, P. A., and Kakas, A. C. (2000). Abductive and Inductive Reasoning: Background and Issues. In *Abduction and Induction*, 1-27. Springer.

Frühwirth, T. (1998). Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, 37(1-3), 95-138.

Frühwirth, T. (2009. *Constraint Handling Rules*. Cambridge University Press.

Gelfond, M. and Lifschitz, V. (1988). The Stable Model Semantics for Logic Programming. In *ICLP/SLP*, 1070-1080.

Green, T. J., Huang, S. S., Loo, B. T. and Zhou, W. (2013). Datalog and Recursive Query Processing. *Foundations and Trends in Databases*, 5(2), 105-195.

Harel, D. (1980). Review on Logic and Data Bases, *Computing Reviews*, 367-369.

van Hentenryck, P. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge: MIT press.

Hewitt, C. (1985). The Challenge of Open Systems: Current Logic Programming Methods may be Insufficient for Developing the Intelligent Systems of the Future. *Byte*, 10(4), 223-242.

Hill, P. and Lloyd, J. (1994). *The Gödel Programming Language*. MIT press.

Horrocks, I., Patel-Schneider, P. F. and Van Harmelen, F. (2003). From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Web semantics: science, services and agents on the World Wide Web*, 1(1), 7-26.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B. and Dean, M. (2004). SWRL: A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member submission*, 21, 79.

Jaffar, J. and Lassez, J. L. (1987). Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 111-119. ACM.

Jaffar, J. and Maher, M. J. (1994). Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19, 503-581.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M. and Irwin, J. (1997). *Aspect-oriented Programming*, Springer, Berlin Heidelberg.

Kakas, A., Kowalski, R. and Toni, F. (1992). Abductive Logic Programming. *Journal of Logic and Computation*, 2(6), 719-770.

Kowalski, R. (1988). Logic-based Open Systems. In *Representation and Reasoning*, Jakob ph. Hoepelman (Hg.) Max Niemeyer Verlag, Tbingen, 125-134. Also Department of Computing, Imperial College, 1985.

Kowalski, R. (1995). Logic without Model Theory. *In What is a logical system?*, (ed. D. Gabbay), Oxford University Press.

Kowalski, R. (1995). Using Metalogic to Reconcile Reactive with Rational Agents. In *Meta-Logics and Logic Programming*, (K. Apt and F. Turini, eds.), MIT Press.

Kowalski, R. (2006). Computational Logic in an Object-oriented World. In *Reasoning, Action and Interaction in AI Theories and Systems*, 59-82. Springer

Berlin Heidelberg.

Kowalski, R. (2011). *Computational Logic and Human Thinking: How to be Artificially Intelligent*, Cambridge University Press.

Kowalski, R. (2014). Logic Programming. To appear in *Volume 9, Computational Logic*, (Joerg Siekmann, editor). In the *History of Logic* series, edited by Dov Gabbay and John Woods, Elsevier.

Kowalski, R. and Burton A. (2012). WUENIC A Case Study in Rule-based Knowledge Representation and Reasoning. In *Post-proceedings International Workshop on Juris-informatics*, Springer-Verlag.

Kowalski, R. and Kim, J. S. (1991). A Metalogic Programming Approach to Multi-agent Knowledge and Belief. *Artificial intelligence and Mathematical Theory of Computation*, 231-246.

Kowalski, R. and Sadri, F. (1996). Towards a Unified Agent Architecture that Combines Rationality with Reactivity, In *Proc. International Workshop on Logic in Databases*, San Miniato, Italy, Springer- Verlag, LNCS 1154.

Kowalski, R. and Sadri, F. (1999). From Logic Programming towards Multi-agent Systems, *Annals of Mathematics and Artificial Intelligence*, pages 391-419.

Kowalski, R. and Sadri, F. (2009). Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents, In *Web Reasoning and Rule Systems* (eds. A. Polleres and T. Swift), Springer, LNCS 5837.

Kowalski, R. and Sadri, F. (2011). Abductive Logic Programming Agents with Destructive Databases, *Annals of Mathematics and Artificial Intelligence* 62(1-2), 129-158.

Kowalski, R. and Sadri, F. (2012). A Logic-Based Framework for Reactive Systems. *RuleML 2012*, A. Bikakis and A. Giurca (Eds.), LNCS 7438, 1-15. Springer, Heidelberg.

Kowalski, R. and Sadri, F. (2014a). A Logical Characterization of a Reactive System Language. In *Rules on the Web: From Theory to Applications*, 22-36), Springer International Publishing.

Kowalski, R. and Sadri, F. (2014b) Reactive Computing as Model Generation. to appear in *New Generation Computing*.

Kowalski, R. and Sergot, M. (1986). A Logic-based Calculus of Events. *New Generation Computing*, 4(1), 67-95.

Kowalski, R., Toni, F. and Wetzel, G. (1998). Executing Suspended Logic Programs. *Fundamenta Informatica*, 34(3), 203-224.

Lally, A., Prager, J. M., McCord, M. C., Boguraev, B. K., Patwardhan, S., Fan, J. and Chu-Carroll, J. (2012). Question analysis: How Watson reads a clue. *IBM Journal of Research and Development*, 56(3.4), 2-1.

Lamma, E., Mello, P., Milano, M., and Riguzzi, F. (1999). Integrating Induction and Abduction in Logic Programming. *Information Sciences*, 116(1), 25-54.

Lloyd, J. W. and Topor, R. W. (1984). Making Prolog More Expressive. *The Journal of Logic Programming*, 1(3), 225-240.

Lusk, E., Butler, R., Disz, T., Olson, R., Overbeek, R., Stevens, R., Warren, D., Calderwood, A., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A. and

Hausman, B. (1990). The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(23), 243271.

Maes, P. (1990). Situated Agents Can Have Goals. *Robot. Autonomous Syst.* 6 (1-2), 49-70.

Manthey, R. and Bry, F. (1988). SATCHMO: a Theorem Prover Implemented in Prolog. In *9th International Conference on Automated Deduction*, 415-434. Springer Berlin Heidelberg.

Quine, W. V. O. (1963). Two Dogmas of Empiricism. In *From a Logical Point of View*, Harper and Row, 20-46.

Marques-Silva, J. and Sakallah, K. A. (1999). GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.*, 48(5):506521.

Minto, B. (2010). *The Pyramid Principle: Logic in Writing and Thinking.* Pearson Education. (An earlier version was published in 1987.)

Moto-Oka, T. (Ed.). (1982). *Fifth Generation Computer Systems.* Elsevier.

Muggleton, S. (1991). "Inductive Logic Programming". *New Generation Computing*, 8 (4): 295318.

Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P., Inoue, K. and Srinivasan, A. (2012). ILP Turns 20. *Machine Learning*, 86(1), 3-23.

Plotkin, G. D. (1970). "A Note on Inductive Generalization". In Meltzer, B.; Michie, D. *Machine Intelligence 5*, Edinburgh University Press, 153163.

Poole, D., Goebel, R. and Aleliunas, R. (1987). Theorist: A Logical Reasoning System for Defaults and Diagnosis. In N. Cercone and G. McCalla (Eds.) *The Knowledge Frontier: Essays in the Representation of Knowledge*, Springer Verlag, New York, 331-352.

Poole, D. (1988). A Logical Framework for Default Reasoning. *Artificial intelligence*, 36(1), 27-47.

Przymusinski, T. C. (1988). On the Declarative Semantics of Deductive Databases and Logic Programs, In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, Los Altos, CA, 193-216.

Robinson, J. A. (1968). The Generalized Resolution Principle. *Machine intelligence 3*, 77-93.

Sadri, F. and Kowalski, R. (1988). A Theorem-Proving Approach to Database Integrity. In Minker, J. [ed.], *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, 313-362.

Sergot, M., Sadri, F., Kowalski, R., Kriwaczek, F., Hammond, P., and Cory, T. (1986). The British Nationality Act as a Logic Program, *CACM*, 29(5), 370-386.

Shapiro, E. Y. (1987). *Concurrent Prolog: Collected Papers.* MIT press.

Siekmann, J. and Wrightson, G. (2002). An Open Research Problem: Strong Completeness of R. Kowalski's Connection Graph Proof Procedure. In *Computational Logic: Logic Programming and Beyond*, 231-252. Springer Berlin Heidelberg.

Smolka, G. (1982). Completeness of the Connection Graph Proof Procedure for Unit-Refutable Clause Sets. *GWAI-82*, 191-204. Springer Berlin Heidelberg.

Swift, T., and Warren, D. S. (2012). XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1-2), 157-187.

Ueda, K. (1986). Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. *ICOT Technical Report TR-208*, Institute for New Generation Computer Technology (ICOT), Tokyo. Revised version in *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, 441-456, 1988.

Ueda, K. (1999). Concurrent Logic/Constraint Programming: The next 10 years. In *The Logic Programming Paradigm*, 53-71. Springer Berlin Heidelberg.

Van Gelder, A. (1989). Negation as Failure Using Tight Derivations for General Logic Programs, *The Journal of Logic Programming*, 6 (1-2), 109-133.

Van Gelder, A, Ross, K. A. and Schlipf, J. (1991). The Well-Founded Semantics for General Logic Programs. *JACM* 38(3), 620-650.