HANDBOOK OF PHILOSOPHICAL LOGIC
2ND EDITION

VOLUME 12

HANDBOOK OF PHILOSOPHICAL LOGIC

2nd Edition

Volume 12

edited by D.M. Gabbay and F. Guenthner

Volume 1 – ISBN 0-7923-7018-X
Volume 2 – ISBN 0-7923-7126-7
Volume 3 – ISBN 0-7923-7160-7
Volume 4 – ISBN 1-4020-0139-8
Volume 5 – ISBN 1-4020-0235-1
Volume 6 – ISBN 1-4020-0583-0
Volume 7 – ISBN 1-4020-0599-7
Volume 8 – ISBN 1-4020-0665-9
Volume 9 – ISBN 1-4020-0699-3
Volume 10 – ISBN 1-4020-1644-1
Volume 11 – ISBN 1-4020-1966-1

HANDBOOK OF PHILOSOPHICAL LOGIC

2nd EDITION

VOLUME 12

Edited by

D.M. GABBAY
King's College, London, U.K.

and

F. GUENTHNER
Centrum für Informations- und Sprachverarbeitung,
Ludwig-Maximilians-Universität München, Germany

Springer

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN-10 1-4020-3091-6 (HB)
ISBN-13 978-1-4020-3091-8 (HB)
ISBN-10 1-4020-3092-4 (e-book)
ISBN-13 978-1-4020-3092-5 (e-book)

Published by Springer,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

www.springeronline.com

Printed on acid-free paper

Printed in the Netherlands.

CONTENTS

PREFACE TO THE SECOND EDITION

It is with great pleasure that we are presenting to the community the second edition of this extraordinary handbook. It has been over 15 years since the publication of the first edition and there have been great changes in the landscape of philosophical logic since then.

The first edition has proved invaluable to generations of students and researchers in formal philosophy and language, as well as to consumers of logic in many applied areas. The main logic article in the Encyclopaedia Britannica 1999 has described the first edition as 'the best starting point for exploring any of the topics in logic'. We are confident that the second edition will prove to be just as good!

The first edition was the second handbook published for the logic community. It followed the North Holland one volume *Handbook of Mathematical Logic*, published in 1977, edited by the late Jon Barwise. The four volume *Handbook of Philosophical Logic*, published 1983–1989 came at a fortunate temporal junction at the evolution of logic. This was the time when logic was gaining ground in computer science and artificial intelligence circles.

These areas were under increasing commercial pressure to provide devices which help and/or replace the human in his daily activity. This pressure required the use of logic in the modelling of human activity and organisation on the one hand and to provide the theoretical basis for the computer program constructs on the other. The result was that the *Handbook of Philosophical Logic*, which covered most of the areas needed from logic for these active communities, became their bible.

The increased demand for philosophical logic from computer science and artificial intelligence and computational linguistics accelerated the development of the subject directly and indirectly. It directly pushed research forward, stimulated by the needs of applications. New logic areas became established and old areas were enriched and expanded. At the same time, it socially provided employment for generations of logicians residing in computer science, linguistics and electrical engineering departments which of course helped keep the logic community thriving. In addition to that, it so happens (perhaps not by accident) that many of the Handbook contributors became active in these application areas and took their place as time passed on, among the most famous leading figures of applied philosophical logic of our times. Today we have a handbook with a most extraordinary collection of famous people as authors!

The table below will give our readers an idea of the landscape of logic and its relation to computer science and formal language and artificial intelligence. It shows that the first edition is very close to the mark of what was needed. Two topics were not included in the first edition, even though

D. Gabbay and F. Guenthner (eds.),
Handbook of Philosophical Logic, Volume 12, vii–xiii.

they were extensively discussed by all authors in a 3-day Handbook meeting. These are:

- a chapter on non-monotonic logic
- a chapter on combinatory logic and λ-calculus

We felt at the time (1979) that non-monotonic logic was not ready for a chapter yet and that combinatory logic and λ-calculus was too far removed.[1] Non-monotonic logic is now a very major area of philosophical logic, alongside default logics, labelled deductive systems, fibring logics, multi-dimensional, multimodal and substructural logics. Intensive re-examinations of fragments of classical logic have produced fresh insights, including at time decision procedures and equivalence with non-classical systems.

Perhaps the most impressive achievement of philosophical logic as arising in the past decade has been the effective negotiation of research partnerships with fallacy theory, informal logic and argumentation theory, attested to by the Amsterdam Conference in Logic and Argumentation in 1995, and the two Bonn Conferences in Practical Reasoning in 1996 and 1997.

These subjects are becoming more and more useful in agent theory and intelligent and reactive databases.

Finally, fifteen years after the start of the Handbook project, I would like to take this opportunity to put forward my current views about logic in computer science, computational linguistics and artificial intelligence. In the early 1980s the perception of the role of logic in computer science was that of a specification and reasoning tool and that of a basis for possibly neat computer languages. The computer scientist was manipulating data structures and the use of logic was one of his options.

My own view at the time was that there was an opportunity for logic to play a key role in computer science and to exchange benefits with this rich and important application area and thus enhance its own evolution. The relationship between logic and computer science was perceived as very much like the relationship of applied mathematics to physics and engineering. Applied mathematics evolves through its use as an essential tool, and so we hoped for logic. Today my view has changed. As computer science and artificial intelligence deal more and more with distributed and interactive systems, processes, concurrency, agents, causes, transitions, communication and control (to name a few), the researcher in this area is having more and more in common with the traditional philosopher who has been analysing

[1] I am really sorry, in hindsight, about the omission of the non-monotonic logic chapter. I wonder how the subject would have developed, if the AI research community had had a theoretical model, in the form of a chapter, to look at. Perhaps the area would have developed in a more streamlined way!

such questions for centuries (unrestricted by the capabilities of any hardware).

The principles governing the interaction of several processes, for example, are abstract an similar to principles governing the cooperation of two large organisation. A detailed rule based effective but rigid bureaucracy is very much similar to a complex computer program handling and manipulating data. My guess is that the principles underlying one are very much the same as those underlying the other.

I believe the day is not far away in the future when the computer scientist will wake up one morning with the realisation that he is actually a kind of formal philosopher!

The projected number of volumes for this Handbook is about 18. The subject has evolved and its areas have become interrelated to such an extent that it no longer makes sense to dedicate volumes to topics. However, the volumes do follow some natural groupings of chapters.

I would like to thank our authors and readers for their contributions and their commitment in making this Handbook a success. Thanks also to our publication administrator Mrs J. Spurr for her usual dedication and excellence and to Kluwer Academic Publishers (now Springer) for their continuing support for the Handbook.

Dov Gabbay
King's College London

Logic	**IT**			
	Natural language processing	**Program control specification, verification, concurrency**	**Artificial intelligence**	**Logic programming**
Temporal logic	Expressive power of tense operators. Temporal indices. Separation of past from future	Expressive power for recurrent events. Specification of temporal control. Decision problems. Model checking.	Planning. Time dependent data. Event calculus. Persistence through time—the Frame Problem. Temporal query language. temporal transactions.	Extension of Horn clause with time capability. Event calculus. Temporal logic programming.
Modal logic. Multi-modal logics	generalised quantifiers	Action logic	Belief revision. Inferential databases	Negation by failure and modality
Algorithmic proof	Discourse representation. Direct computation on linguistic input	New logics. Generic theorem provers	General theory of reasoning. Non-monotonic systems	Procedural approach to logic
Non-monotonic reasoning	Resolving ambiguities. Machine translation. Document classification. Relevance theory	Loop checking. Non-monotonic decisions about loops. Faults in systems.	Intrinsic logical discipline for AI. Evolving and communicating databases	Negation by failure. Deductive databases
Probabilistic and fuzzy logic	logical analysis of language	Real time systems	Expert systems. Machine learning	Semantics for logic programs
Intuitionistic logic	Quantifiers in logic	Constructive reasoning and proof theory about specification design	Intuitionistic logic is a better logical basis than classical logic	Horn clause logic is really intuitionistic. Extension of logic programming languages
Set theory, higher-order logic, λ-calculus, types	Montague semantics. Situation semantics	Non-well-founded sets	Hereditary finite predicates	λ-calculus extension to logic programs

Imperative vs. declarative languages	**Database theory**	**Complexity theory**	**Agent theory**	**Special comments: A look to the future**
Temporal logic as a declarative programming language. The changing past in databases. The imperative future	Temporal databases and temporal transactions	Complexity questions of decision procedures of the logics involved	An essential component	Temporal systems are becoming more and more sophisticated and extensively applied
Dynamic logic	Database updates and action logic	Ditto	Possible actions	Multimodal logics are on the rise. Quantification and context becoming very active
Types. Term rewrite systems. Abstract interpretation	Abduction, relevance	Ditto	Agent's implementation rely on proof theory.	
	Inferential databases. Non-monotonic coding of databases	Ditto	Agent's reasoning is non-monotonic	A major area now. Important for formalising practical reasoning
	Fuzzy and probabilistic data	Ditto	Connection with decision theory	Major area now
Semantics for programming languages. Martin-Löf theories	Database transactions. Inductive learning	Ditto	Agents constructive reasoning	Still a major central alternative to classical logic
Semantics for programming languages. Abstract interpretation. Domain recursion theory.		Ditto		More central than ever!

Classical logic. Classical fragments	Basic background language	Program synthesis	A basic tool	
Labelled deductive systems	Extremely useful in modelling		A unifying framework. Context theory.	Annotated logic programs
Resource and substructural logics	Lambek calculus		Truth maintenance systems	
Fibring and combining logics	Dynamic syntax	Modules. Combining languages	Logics of space and time	Combining features
Fallacy theory				
Logical Dynamics	Widely applied here			
Argumentation theory games		Game semantics gaining ground		
Object level/ metalevel			Extensively used in AI	
Mechanisms: Abduction, default relevance			ditto	
Connection with neural nets				
Time-action-revision models			ditto	

	Relational databases	Logical complexity classes	The workhorse of logic	The study of fragments is very active and promising.
	Labelling allows for context and control.		Essential tool.	The new unifying framework for logics
Linear logic			Agents have limited resources	
	Linked databases. Reactive databases		Agents are built up of various fibred mechanisms	The notion of self-fibring allows for self-reference
				Fallacies are really valid modes of reasoning in the right context.
			Potentially applicable	A dynamic view of logic
				On the rise in all areas of applied logic. Promises a great future
			Important feature of agents	Always central in all areas
			Very important for agents	Becoming part of the notion of a logic
				Of great importance to the future. Just starting
			A new theory of logical agent	A new kind of model

GERHARD BREWKA AND JÜRGEN DIX

KNOWLEDGE REPRESENTATION WITH LOGIC PROGRAMS

In this overview article we show how knowledge representation (KR) can be done with the help of *generalized* logic programs. We start by introducing the core of PROLOG, which is based on *definite* logic programs. Although this class is very restricted (and will be enriched by various additional features in the rest of the paper), it has a very nice property for KR-tasks: there exist efficient *query-answering procedures*—a *top-down* approach and a *bottom-up* evaluation. In addition we can not only handle ground queries but also queries with variables and compute *answer-substitutions*.

It turns out that more advanced KR-tasks can not be properly handled with definite programs. Therefore we extend this basic class of programs by additional features like *negation-as-finite-failure*, *default-negation*, *explicit negation*, *preferences*, and *disjunction*. The need for these extensions is motivated by suitable examples and the corresponding semantics are discussed in detail.

Clearly, the more expressive the respective class of programs under a certain semantics is, the less efficient are potential query-answering methods. This point will be illustrated and discussed for every extension. By well-known recursion-theoretic results, it is obvious that there do not exist complete query-answering procedures for the general case where variables and function symbols are allowed. Nevertheless we consider it an important topic of further research to extract *feasible* classes of programs where answer-substitutions can be computed.

1 INTRODUCTION

One of the major reasons for the success story (if one is really willing to call it a success story) of human beings on this planet is our ability to invent tools that help us improve our—otherwise often quite limited—capabilities. The invention of machines that are able to do interesting things, like transporting people from one place to the other (even through the air), sending moving pictures and sounds around the globe, bringing our email to the right person, and the like, is one of the cornerstones of our culture and determines to a great degree our everyday life.

Among the most challenging tools one can think of are machines that are able to handle knowledge adequately. Wouldn't it be great if, instead of the stupid delivery robot which brings coffee from the kitchen to your office every day at 9 am, and which needs complete reengineering whenever your

D. Gabbay and F. Guenthner (eds.),
Handbook of Philosophical Logic, Volume 12, 1–85.

coffee preferences change, you could (for the same price, admitted) get a smart device which simply can be told that you want your coffee black this morning, and that you need an extra Aspirin since it was your colleague's birthday yesterday? To react in the right way to your needs such a robot would have to know a lot, for instance that Aspirin should come with a glass of water, or that people in certain situations need their coffee extra strong.

Building smart machines of this kind is at the heart of Artificial Intelligence (AI). Since such machines will need tremendous amounts of knowledge to work properly, even in very limited environments, the investigation of techniques for representing knowledge and reasoning is highly important.

In the early days of AI it was still believed that modeling general purpose problem solving capabilites, as in Newell and Simon's famous GPS (General Problem Solver) program, would be sufficient to generate intelligent behaviour. This hypothesis, however, turned out to be overly optimistic. At the end of the sixties people realized that an approach using available knowledge about narrow domains was much more fruitful. This led to the expert systems boom which produced many useful application systems, expert system building tools, and expert system companies. Many of the systems are still in use and save companies millions of dollars per year.[1]

Nevertheless, the simple knowledge representation and reasoning methods underlying the early expert systems soon turned out to be insufficient. Most of the systems were built based on simple rule languages, often enhanced with ad hoc approaches to model uncertainty. It became apparent that more advanced methods to handle incompleteness, defeasible reasoning, uncertainty, causality and the like were needed.

This insight led to a tremendous increase of research on the foundations of knowledge representation and reasoning. Theoretical research in this area has blossomed in recent years. Many advances have been made and important results were obtained. The technical quality of this work is often impressive.

On the other hand, most of these advanced techniques have had surprisingly little influence on practical applications so far. To a certain degree this is understandable since theoretical foundations had to be laid first and pioneering work was needed. However, if we do not want research in knowledge representation to remain a theoreticians' game more emphasis on computability and applicability seems to be needed. We strongly believe that the kind of research presented in this tutorial, that is research aiming at interesting combinations of ideas from logic programming and nonmonotonic reasoning, provides an important step into this direction.

[1] We refer the interested reader to [Russel and Norvig, 1995] which gives a very detailed and nice exposition of what has been done in AI since its very beginning until today.

1.1 Some History

Historically, logic programs have been considered in the logic programming community for about 30 years. It began with [Colmerauer *et al.*, 1973; Kowalski, 1974; van Emden and Kowalski, 1976] and led to the definition and implementation of *PROLOG*, a by now theoretically well-understood programming language (at least the declarative part consisting of Horn-clauses: *pure PROLOG*). Extensions of PROLOG allowing negative literals have been also considered in this area: they rely on the idea of *negation-as-finite-failure*, we call them *logic-programming-semantics* (or shortly *LP-semantics*).

In parallel, starting at about 1980, *Nonmonotonic Reasoning* entered into computer science and began to constitute a new field of active research. It was originally initiated because *Knowledge Representation* and *Common-Sense Reasoning* using classical logic came to its limits. Formalisms like classical logic are inherently monotonic and they seem to be too weak and therefore inadequate for such reasoning problems.

In recent years, independently of the research in logic programming, people interested in knowledge representation and nonmonotonic reasoning also tried to define declarative semantics for programs containing *default* or *explicit* negation and even *disjunctions*. They defined various semantics by appealing to (different) intuitions they had about programs.

This second line of research started in 1986 with the *Workshop on the Foundations of Deductive Databases and Logic Programming* organized by Jack Minker: the revised papers of the proceedings were published in [Minker, 1988]. The *stratified* (or the similar *perfect*) semantics presented there can be seen as a splitting-point: it is still of interest for the logic programming community (see [Cavedon and Lloyd, 1989]) but its underlying intuitions were inspired by nonmonotonic reasoning and therefore much more suitable for knowledge representation tasks. Semantics of this kind leave the philosophy underlying classical logic programming in that their primary aim is not to model *negation-as-finite-failure*, but to construct new, more powerful semantics suitable for applications in knowledge representation. Let us call such semantics *NMR-semantics*.

Nowadays, due to the work of Apt, Blair and Walker, Fitting, Gelfond, Lifschitz, Przymusinski and others, very close relationships between these two independent research lines became evident. Methods from logic programming, e.g. least fixpoints of certain operators, can be used successfully to define NMR-semantics.

The NMR-semantics also shed new light on the understanding of the classical nonmonotonic logics such as *Default Logic*, *Autoepistemic Logic* and the various versions of *Circumscription*. In addition, the investigation of possible semantics for logic programs seems to be useful because

1. parts of nonmonotonic systems (which are usually defined for full predicate logic, or even contain additional (modal)-operators) may be "implemented" with the help of such programs,

2. nonmonotonicity in these logics may be described with an appropriate treatment of negation in logic programs.

1.2 Non-Monotonic Formalisms in KR

As already mentioned above, research in nonmonotonic reasoning has begun at the end of the seventies. One of the major motivations came from reasoning about actions and events. John McCarthy and Patrick Hayes had proposed their situation calculus as a means of representing changing environments in logic. The basic idea is to use an extra situation argument for each fact which describes the situation in which the fact holds. Situations, basically, are the results of performing sequences of actions. It soon turned out that the problem was not so much to represent what changes but *to represent what does not change* when an event occurs. This is the so-called *frame problem*. The idea was to handle the frame problem by using a default rule of the form

> *If a property P holds in situation S then P typically also holds in the situation obtained by performing action A in S.*

Given such a rule it is only necessary to explicitly describe the changes induced by a particular action. All non-changes, for instance that the *real colour* of the kitchen wall does not change when the light is turned on, are handled implicitly. Although it turned out that a straightforward formulation of this rule in some of the most popular nonmonotonic formalisms may lead to unintended results the frame problem was certainly the challenge motivating many people to join the field.

In the meantime a large number of different nonmonotonic logics have been proposed. We can distinguish four major types of such logics:

1. Logics using nonstandard inference rules with an additional consistency check to represent default rules. Reiter's default logic (see Appendix A.3) and its variants are of this type.

2. Nonmonotonic modal logics using a modal operator to represent consistency or (dis-) belief. These logics are nonmonotonic since conclusions may depend on disbelief. The most prominent example is Moore's autoepistemic logic.

3. Circumscription (see Appendix A.4) and its variants. These approaches are based on a preference relation on models. A formula is a consequence iff it is true in all most preferred models of the premises. Syn-

tactically, a second order formula is used to eliminate all non-preferred models.

4. Conditional approaches which use a non truth-functional connective $\mid\!\sim$ to represent defaults. A particularly interesting way of using such conditionals was proposed by Kraus, Lehmann and Magidor. They consider p as a default consequence of q iff the conditional $q \mid\!\sim p$ is in the closure of a given conditional knowledge base under a collection of rules. Each of the rules directly corresponds to a desirable property of a nonmonotonic inference relation.

The various logics are intended to handle different intuitions about nonmonotonic reasoning in a most general way. On the other hand, the generality leads to problems, at least from the point of view of implementations and applications. In the first order case the approaches are not even semidecidable since an implicit consistency check is needed. In the propositional case we still have tremendous complexity problems. For instance, the complexity of determining whether a formula is contained in all extensions of a propositional default theory is on the second level of the polynomial hierarchy. As mentioned earlier we believe that logic programming techniques can help to overcome these difficulties.

Originally, nonmonotonic reasoning was intended to provide us with a *fast* but *unsound* approximation of classical reasoning in the presence of incomplete knowledge. Therefore one might ask whether the higher complexity of NMR-formalisms (compared to classical logic) is not a real drawback of this aim? The answer is that NMR-systems allow us to formulate a problem in a very *compact* way as a theory T. It turns out that for some problems any equivalent formulation in classical logic (if possible at all) as a theory T' is much larger: the size of T' is exponential in the size of T! We refer to [Gogic *et al.*, 1995] and [Cadoli *et al.*, 1996; Cadoli *et al.*, 1997; Cadoli *et al.*, 1995] where such problems are investigated.

1.3 How This Chapter is Organized

In this overview paper we show how Knowledge Representation can be done with the help of *generalized* logic programs. We start by introducing the core of PROLOG, which is based on *definite* logic programs. Although this class is very restricted (and will be enriched by various additional features in the rest of the paper), it has a very nice property for KR-tasks: there exist efficient *query-answering procedures*— a *top-down* approach and a *bottom-up* evaluation. In addition we can not only handle ground queries but also queries with variables and compute *answer-substitutions*.

It turns out that more advanced KR-tasks can not be properly handled with definite programs. Therefore we extend this basic class of programs by additional features like *negation-as-finite-failure*, *default-negation*, *explicit negation*, *preferences*, and *disjunction*. The need for these extensions is

motivated by suitable examples and the corresponding semantics are also discussed.

Clearly, the more expressive the respective class of programs under a certain semantics is, the less efficient are potential query-answering methods. This point will be illustrated and discussed for every extension. By well-known recursion-theoretic results, it is obvious that there do not exist complete query-answering procedures for the general case where variables and function symbols are allowed. Nevertheless we consider it an important topic of further research to extract *feasible* classes of programs where answer-substitutions can be computed.

2 DEFINITE LOGIC PROGRAMS

In this section we consider the most restricted class of programs: *definite* logic programs, programs without any negation at all. All the extensions of this basic class we will introduce later contain at least some kind of negation (and perhaps additional features). But here we also allow the ocurrence of free variables as well as function symbols.

In Section 2.1 we introduce as a representative for the *top-down* approach the SLD-resolution. Section 2.2 presents the main competing approach of SLD: *bottom-up evaluation*. This approach is used in the database community and it is efficient when additional assumptions are made (*finiteness-assumptions*, *no function symbols*). In Section 2.3 we consider the influence and appropriateness of Herbrand models and their underlying intuition. Finally in Section 2.4 we present and discuss two important examples in KR: *reasoning in inheritance hierarchies* and *reasoning about actions*. Both examples clearly motivate the need of extending definite programs by a kind of *default-negation* "not".

First some notation used throughout this paper. A language $\mathcal{L}$ consists of a set of relation symbols and a set of function symbols (each symbol has an associated arity). Nullary functions are called constants. Terms and atoms are built from $\mathcal{L}$ in the usual way starting with variables, applying function symbols and relation-symbols.

Instead of considering arbitrary $\mathcal{L}$-formulae, our main object of interest is a program:

DEFINITION 1 (Definite Logic Program).
A *definite* logic program consists of a finite number of *rules* of the form

$$A \leftarrow B_1, \ldots, B_m,$$

where $A, B_1, \ldots, B_m$ are positive atoms (containing possibly free variables). We call A the *head* of the rule and $B_1, \ldots, B_m$ its *body*. The comma represents conjunction $\wedge$.

We can think of a program as formalizing our knowledge about the world and how the world behaves. Of course, we also want to derive new information, i.e. we want to ask queries:

DEFINITION 2 (Query).
Given a definite program we usually have a definite query in mind that we want to be solved. A definite query Q is a conjunction of positive atoms $C_1 \wedge \ldots \wedge C_l$ which we denote by

$$?\text{-}\ \ C_1, \ldots, C_l.$$

These C_i may also contain variables. Asking a query Q to a program P means asking for all possible substitutions Θ of the variables in Q such that $Q\Theta$ follows from P. Often, Θ is also called an answer to Q. Note that $Q\Theta$ may still contain free variables.

Note that if a program P is given, we usually assume that it also determines the underlying language $\mathcal{L}$, denoted by $\mathcal{L}_P$, which is generated by exactly the symbols ocurring in P. The set of all these atoms is called the *Herbrand base* and denoted by $B_{\mathcal{L}_P}$ or simply B_P. The corresponding set of all ground terms is the *Herbrand universe*. Another important notion that we are not explaining in detail here is that of *unification*. Given two atoms A and B with free variables we can ask if we can compute two substitutions Θ_1, Θ_2 for the variables such that

$$A\Theta_1 \ \text{ is identical to } \ B\Theta_2,$$

or if we can decide that this is not possible at all. In fact, if the two atoms are *unifiable* we can indeed compute a *most general unifier*, called mgU (see [Lloyd, 1987]). The mgU Θ is a substitution defined on the set of variables occurring in both A and B such that $A\Theta$ is identical to $B\Theta$.

This will be important in our framework because if an atom appears as a subgoal in a query, we may want to determine if there are rules in the program whose heads unify with this atom.

How are our programs related to classical predicate logic? Of course, we can map a program-rule into classical logic by interpreting "$\leftarrow$" as material implication "$\supset$" and universally quantifying. This means we view such a rule as the following universally quantified formula

$$B_1 \wedge \ldots \wedge B_m \ \supset \ A.$$

However, as we will see later, there is a great difference: a logic program-rule takes some orientation with it. This makes it possible to formulate the following principle as an underlying intuition of all semantics of logic programs:

Principle 1 (Orientation).
If a ground atom A does not unify with some head of a program rule of P,

then this atom is considered to be false. In this case we say that "not A" is derivable from P to distinguish it from classical $\neg A$.

The orientation principle is nothing but a weak form of *negation-by-failure*. Given an intermediate goal not A, we first try to prove A. But if A does not unify with any head, A fails and this is the reason to derive not A.

2.1 Top-Down

SLD-Resolution[2] is a special form of Robinson's general resolution rule. While Robinson's rule is complete for full first order logic, SLD is complete for definite logic programs (see Theorem 4 on the facing page). We do not give a complete definition of SLD-resolution (see [Lloyd, 1987]) but rather prefer to illustrate its behaviour on the following example.

EXAMPLE 3 (SLD-Resolution).
Let the program P_{SLD} consist of the following three clauses

(1) $p(x,z) \leftarrow q(x,y), p(y,z)$
(2) $p(x,x)$
(3) $q(a,b)$

The query Q we are interested in is given by $p(x,b)$. I.e. we are looking for all substitutions Θ for x such that $p(x,b)\Theta$ follows from P.

Figure 1 on the next page illustrates the behaviour of SLD-resolution. We start with our query in the form $\leftarrow Q$. Sometimes the notation $\Box \leftarrow Q$ is also used, where $\Box$ denotes the falsum. In any round the selected atom is underlined: numbers 1, 2 or 3 indicate the number of the clause which the selected atom is resolved against. Obviously, there are three different sorts of branches, namely

1. *infinite* branches,
2. branches that *end up with the empty clause*, and
3. branches that *end in a deadlock ("Failure")*: no applicable rule is left.

In this example we always resolve with the *last* atom in the goal under consideration. If we choose always the *first* atom in the goal, we will obtain, at least in this example, a *finite* tree.

Definite programs have the nice feature that the intersection of all Herbrand-models exists and is again a Herbrand model of P. It is denoted by M_P and called the least Herbrand-model of P. Note that our original

[2] **SL**-resolution for **D**efinite clauses. **SL**-resolution stands for **L**inear resolution with **S**election function.

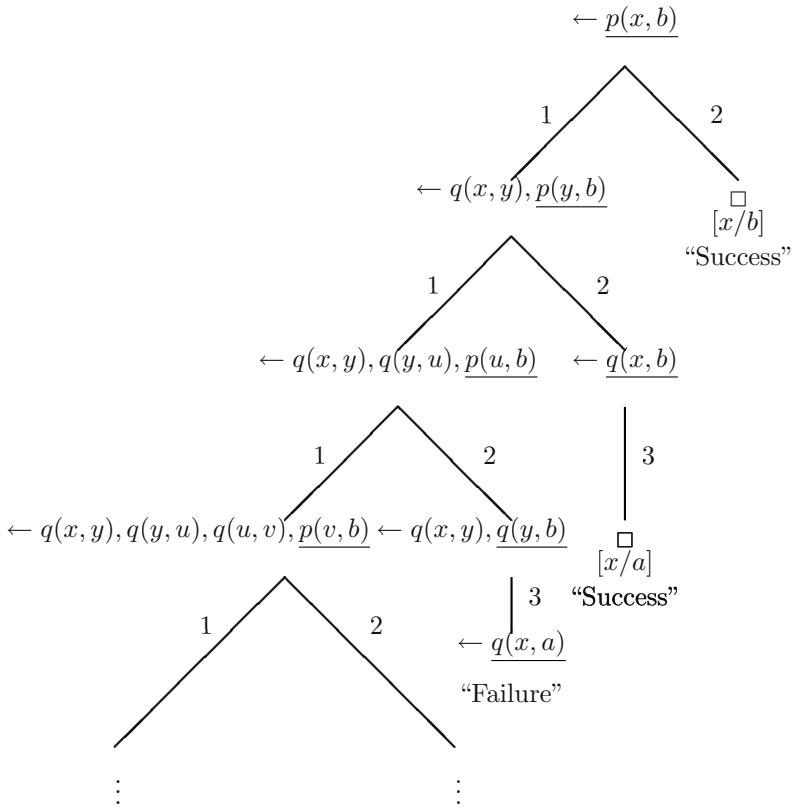

Figure 1. An infinite SLD-Tree

aim was to find substitutions Θ such that $Q\Theta$ is derivable from the program P. This task as well as M_P is closely related to SLD:

THEOREM 4 (Soundness and Completeness of SLD).
The following properties are equivalent:

- $P \models \forall\, Q\Theta$, *i.e.* $\forall\, Q\Theta$ *is true in all models of* P,
- $M_P \models \forall\, Q\Theta$,
- *SLD computes an answer* τ *that subsumes*[3] Θ *wrt* Q.

Note that not any correct answer is computed, only the most general one is (which of course subsumes all the correct ones).

The main feature of SLD-resolution is its *goal-orientedness*. SLD automatically ensures (because it starts with the Query) that we consider only

[3] i.e. $\exists \sigma : Q\tau\sigma = Q\Theta$.

those rules that are relevant for the query to be answered. Rules that are not at all related are simply not considered in the course of the proof.

2.2 Bottom-Up

We mentioned in the last section the least Herbrand model M_P. The bottom-up approach can be described as computing this least Herbrand model from below. We start first with rules with empty bodies (in our example these are all instantiations of rules (2) and (3)). We get as facts all atoms that are in the heads of rules with empty bodies (namely $p(a,a), p(b,b), q(a,b)$ in Example 3 on page 8). In the next round we use the facts that we computed before and try to let the rules "fire", i.e. when their bodies are true, we add their heads to the atoms we already have (this gives us $p(a,b)$).

To be more precise we introduce the immediate consequence operator T_P which associates to any Herbrand model another Herbrand model.

EXAMPLE 5 (T_P).
Given a definite program P let $T_P : \ 2^{B_P} \longmapsto 2^{B_P}$; $\mathcal{I} \longmapsto T_P(\mathcal{I})$

$$T_P(\mathcal{I}) := \{A \in B_P : \quad \begin{array}{l}\text{there is an instantiation of a rule in } P \\ \text{s.t. } A \text{ is the head of this rule and all} \\ \text{body-atoms are contained in } \mathcal{I} \ \}\end{array}$$

It turns out that T_P is monotone and continuous so that (by a general theorem of Knaster-Tarski) the least fixpoint is obtained after ω steps. Moreover we have

THEOREM 6 (T_P and M_P).
$M_P = T_P{\uparrow}^{\omega} = lfp(T_P)$.

This approach is especially important in database applications, where the underlying language does not contain function symbols (DATALOG) — this ensures the Herbrand universe to be finite. Under this condition the iteration stops after finitely many steps. In addition, rules of the form

$$p \leftarrow p$$

do not make any problems. They simply can not be applied or do not produce anything new. Note that in the top-down approach, such rules give rise to infinite branches! Later, elimination of such rules will turn out to be an interesting property. We therefore formulate it as a principle:

Principle 2 (Elimination of Tautologies).
Suppose a program P has a rule which contains the same atom in its body as well as in its head (i.e. the head consists of exactly this atom). Then we can eliminate this rule without changing the semantics.

Unfortunately, such a bottom-up approach has two serious shortcomings. First, the goal-orientedness of SLD-resolution is lost: we are always computing the whole M_P, even those facts that have nothing to do with the query. The reason is that in computing T_P we do not take into account the query we are really interested in. Second, in any step facts that are already computed before are recomputed again. It would be more efficient if only new facts were computed. Both problems can be (partially) solved by appropriate refinements of the naive approach (1) *Semi-naive* bottom-up evaluation ([Bry, 1990; Ullman, 1989a]), and (2) *Magic Sets* techniques ([Beeri and Ramakrishnan, 1991; Ullman, 1989b]).

2.3 Herbrand-Models and the Underlying Language

Usually when we represent some knowledge in first order logic or even in logic programs, it is understood that the underlying language is given exactly by the symbols that occur in the formal theory. Suppose we have represented some knowledge about the world as a theory T in a language $\mathcal{L}$. Classical predicate logic formalizes the notion of a formula ϕ *entailed* by the theory T. This means that ϕ is true in all models of T (we denote this set by $\mathrm{MOD}(T)$). Why are we considering all models? Doesn't it make sense to look only at Herbrand models, i.e. models generated by the underlying language? After all we are not interested in models that contain elements which are not representable as terms in our language. These requirements are usually called *unique names assumption* and *domain closure assumption*:

DEFINITION 7 (UNA and DCA).
Let a language $\mathcal{L}$ be given. We understand by the *unique names assumption* the restriction to those models $\mathcal{I}$, where syntactically different ground $\mathcal{L}$-terms t_1, t_2 are interpreted as nonidentical elements: $t_1^{\mathcal{I}}$ is not identical to $t_2^{\mathcal{I}}$.

By the *domain closure assumption* we mean the restriction to those models $\mathcal{I}$ where for any element a in $\mathcal{I}$ there is a $\mathcal{L}$-term t that represents this element: $a = t^{\mathcal{I}}$.

As an example, in Theorem 4 on page 9 of Section 2.1 we referred to M_P, the least Herbrand model of P. The reason that the first equivalence in this theorem holds is given by the fact that for universal theories T and existential formulae ϕ the following holds

$$\mathrm{MOD}(T) \models \phi \quad \text{iff} \quad \mathrm{Herb}_{\mathcal{L}}\text{-}\mathrm{MOD}(T) \models \phi.$$

In our particular case, where T is a definite program P, we can even replace $\mathrm{Herb}_{\mathcal{L}}$-$\mathrm{MOD}(T)$ in the above equation by the single model M_P.

This last result does not hold in general. But what happens if we nevertheless are interested in only the Herbrand-models of a theory T (and

therefore automatically[4] assume UNA and DCA)? At first sight one can argue that such an approach is much simpler: in contrast to *all* models we only need to take care about the very specific Herbrand models. But it turns out that determining the truth of a formula in all Herbrand models is a much more complex task (namely Π_1^1-complete) than to determine if it is true in all models. This latter task is also undecidable in general, but it is recursively enumerable, i.e. Π_1^0-complete. The fact that this task is recursively enumerable is the content of the famous completeness theorem of Gödel, where "truth of a formula in all models" is shown to be equivalent to deriving this formula in a particular axiomatization of the predicate calculus of first order. We refer to the appendix (Section A.1 and Section A.2) where the necessary notions are introduced.

But we have still a problem with Theorem 4 on page 9 in our restricted setting:

EXAMPLE 8 (Universal Query Problem).
Consider the program $P := p(a)$, the query $Q := p(x)$ and the empty substitution $\Theta := \epsilon$. We have

- $M_P \models \forall x p(x)$,
- but SLD only computes the answer x/a.

Przymusinski called this *the universal query problem.*

There are essentially two solutions to avoid this behaviour: to use a language which is *rich enough* (i.e. contains sufficiently many terms, not only those ocurring in the program P itself) or to consider arbitrary models, not only Herbrand models. Both approaches have been followed in the literature but they are beyond the scope of this paper.

2.4 Why Going Beyond Definite Programs?

So far we have a nice query-answering procedure, SLD-Resolution, which is goal-oriented as well as sound and complete with respect to general derivability. But note that up to now we are not able to derive any *negative* information. Not even our queries allow this. From a very pragmatic viewpoint, we can consider "not A" to be derivable if A is not. Of course, this is not sound with respect to classical logic but it is with respect to M_P.

In KR we do not only want to formulate negative queries, we also want to express *default-statements* of the form

Normally, unless something abnormal holds, then ψ implies ϕ.

[4] The only difference between Herbrand models and models satisfying UNA and DCA is that the interpretation of terms is uniquely determined in Herbrand models. It is required that a term "$f(t_1, \ldots, f_n)$" is interpreted in a Herbrand model $\mathcal{I}$ as "$f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$".

Such statements were the main motivation for nonmonotonic logics, like Default Logic or Circumscription (see Section A.3 and Section A.4 of the appendix). How can we formulate such a statement as a logic program? The most natural way is to use negation "not"

$$\phi \leftarrow \psi, \ \text{not}\, ab$$

where *ab* stands for *abnormality*. Obviously, this forces us to extend definite programs by *negative* atoms, we call them default atoms.

A typical example for such statements occurs in Inheritance Reasoning. We take the following example from [Baral and Gelfond, 1994]:

EXAMPLE 9 (Inheritance Hierachies).
Suppose we know that birds typically fly and penguins are non-flying birds. We also know that Tweety is a bird. Now an agent is hired to build a cage for Tweety. Should the agent put a roof on the cage? After all it could be still the case that Tweety is a penguin and therefore can not fly, in which case we would not like to pay for the unnecessary roof. But under normal conditions, it should be obvious that one should conclude that Tweety is flying.

A natural axiomatization is given as follows:

$$\begin{array}{lllll} P_{Inheritance}: & flies(x) & \leftarrow & bird(x), & \text{not}\, ab(r_1, x) \\ & bird(x) & \leftarrow & penguin(x) & \\ & ab(r_1, x) & \leftarrow & penguin(x) & \\ & make\_top(x) & \leftarrow & flies(x) & \end{array}$$

together with some particular facts, like e.g. $bird(Tweety)$ and $penguin(Sam)$. The first rule formalizes our default knowledge, while the third formalizes that the default rule should not be applied in abnormal or exceptional cases. In our example, it expresses the famous *specificity principle* which says that more specific knowledge should override more general one ([Touretzky, 1986; Touretzky *et al.*, 1988; Horty *et al.*, 1990]).

For the query "$make\_top(Tweety)$" we expect the answer "yes" while for "$make\_top(Sam)$" we expect the answer "no".

Another important KR task is to formalize knowledge for reasoning about action. We again consider a particular important instance of such a task, namely *temporal projection*. The overall framework consists in describing the initial state of the world as well as the effects of all actions that can be performed. What we want to derive is how the world looks like after a sequence of actions has been performed.

EXAMPLE 10 (Temporal Projection: Yale-Shooting Problem).
We distinguish between three sorts[5] of variables:

[5]To be formally correct we have to use many-sorted logic. But since all this could also

- situation variables: $S, S', \ldots$,
- fluent variables: $F, F', \ldots$,
- action variables: $A, A', \ldots$.

The initial situation is denoted by the constant s_0, and the binary function symbol $res(A, S)$ denotes the situation that is reached when in situation S the action A has been performed. The relation symbol $holds(F, S)$ formalizes that the fluent F is true in situation S.

For the YSP there are three actions (*wait*, *load* and *shoot*) and two fluents (*alive* and *loaded*). Initially a turkey called $Fred$ is alive. We then load a gun, wait and shoot. The effect should be that Fred is dead after this sequence of actions. The common-sense argument from which this should follow is the

Law of Inertia: Things normally tend to stay the same.

Using our intuition from the last example, a natural formalization is given as follows:

$$
\begin{array}{lllll}
P_{YSP}: & holds(F, res(A,S)) & \leftarrow & holds(F,S),\ \ \mathrm{not}\, ab(r_1, A, F, S) \\
 & holds(loaded, res(load, S)) & \leftarrow & \\
 & ab(r_1, shoot, alive, S) & \leftarrow & holds(loaded, S) \\
 & holds(alive, s_0) & \leftarrow &
\end{array}
$$

Such a straightforward formalization leads in most versions of classical nonmonotonic logic to the unexpected result, that Fred is not necessarily dead. But obviously we expect to derive $holds(alive, res(load, s_0))$ and

$$\mathrm{not}\, holds(alive\,,\ res(shoot, res(wait, res(load, s_0))))$$

Up to now we only have stated some very "natural" axiomatizations of given knowledge. We have motivated that something like default-negation "not" should be added to definite programs in order to do so and we have explicitly stated the answers to particular queries. What is still missing are solutions to the following very important problems

- *How should an appropriate query answering mechanism handling default-negation "not" look like?*
- *What is the formal semantics that such a procedural mechanism should be checked against?*

Such a semantics is certainly not classical predicate logic because of the default character of "not"— not is not classical $\neg$. Both problems will be considered in detail in Section 3.

be coded in predicate logic by using additional relation symbols, we do not emphasize this fact. We also understand that instantiations are done in such a way that the sorts are respected.

2.5 What is a Semantics?

In the last subsections we have introduced two principles (*Orientation* and *Elimination of Tautologies*) and used the term *semantics of a program* in a loose, imprecise way. We end this section with a precise notion of what we understand by a semantics.

As a first attempt, we can view a semantics as a mapping that associates to any program a set of positive atoms and a set of default atoms. In the case of SLD-Resolution the positive atoms are the ground instances of all derivable atoms. But sometimes we also want to derive default atoms (like in our two examples above). Our *Orientation*-Principle formalizes a minimal requirement for deriving such default-atoms.

Of course, we also want that a semantics SEM should *respect* the rules of P, i.e. whenever SEM makes the body of a rule true, then SEM should also make the head of the rule true. But it can (and will) happen that a semantics SEM does not always decide *all* atoms. Some atoms A are not derivable nor are their default-counterparts $\mathrm{not}\, A$. This means that a semantics SEM can view the body of a rule as being *undefined*.

This already happens in classical logic. Take the theory

$$T := \{(A \wedge B) \supset C,\ \neg A \supset B\}.$$

What are the atoms and negated atoms derivable from T, i.e. true in all models of T? No positive atom nor any negated atom is derivable! The classical semantics therefore makes the truthvalue of $A \wedge B$ undefined in a sense.

Suppose a semantics SEM treats the body of a program rule as undefined. What should we conclude about the head of this rule? We will only require that this head is not treated as false by SEM—it could be true or undefined as well. This means that we require a semantics to be compatible with the program *viewed as a 3-valued theory*—the three values being "*true*", "*false*" and "*undefined*". For the understanding it is not necessary to go deeper into 3-valued logic. We simply note that we interpret "$\leftarrow$" as the Kleene-connective which is *true* for "*undefined* $\leftarrow$ *undefined*" and *false* for "*false* $\leftarrow$ *undefined*".

Our discussion shows that we can view a semantics SEM as a 3-valued model of a program. In classical logic, there is a different viewpoint. For a given theory T we consider there the set of all classical models $\mathrm{MOD}(T)$ as the semantics. The intersection of all these models is of course a 3-valued model of T, but $\mathrm{MOD}(T)$ contains more information. In order to formalize the notion of semantics as general as possible we define

DEFINITION 11 (SEM).
A semantics SEM is a mapping from the class of all programs into the powerset of the set of all 3-valued structures. SEM assigns to every program

P a set of 3-valued models of P:

$$\mathrm{SEM}(P) \subseteq \mathrm{MOD}_{3-val}^{\mathcal{L}_P}(P).$$

This definition covers both the classical viewpoint (classical models are 2-valued and therefore special 3-valued models) as well as our first attempt in the beginning of this subsection. Later on, in most cases we will be really interested only in Herbrand models.

Formally, we can associate to any semantics SEM in the sense of Definition 11 on the page before two entailment relations

sceptical: $\mathrm{SEM}^{scept}(P)$ is the set of all atoms or default atoms that are true in *all models* of $\mathrm{SEM}(P)$.

credulous: $\mathrm{SEM}^{cred}(P)$ is the set of all atoms or default atoms that are true in *at least one model* of $\mathrm{SEM}(P)$.

In this article we only consider the sceptical viewpoint. Also, to facilitate notation, we will not formally distinguish between SEM and SEM^{scept}. In cases where by definition SEM can only contain a single model (like in the case of well-founded semantics) we will omit the outer brackets and write

$$\mathrm{SEM}(P) = M$$

instead of $\mathrm{SEM}(P) = \{M\}$. We will also slightly abuse notation and write $l \in \mathrm{SEM}(P)$ as an abbreviation for $l \in M$ for all $M \in \mathrm{SEM}(P)$.

3 ADDING DEFAULT-NEGATION

In the last section we have illustrated that logic programs with negation are very suitable for KR—they allow a natural and straightforward formalization of default-statements. The problem still remained to define an appropriate semantics for this class and, if possible, to find efficient query-answering methods. Both points are addressed in this section.

We can distinguish between two quite different approaches:

LP-Approach: This is the approach taken mainly in the logic programming community. There one tried to stick as close as possible to SLD-resolution and treat negation as "finite-failure". This resulted in an extension of SLD, called SLDNF-resolution, a procedural mechanism for query answering. For a nice overview, we refer to [Apt and Bol, 1994].

NML-Approach: This is the approach suggested by non-monotonic reasoning people. Here the main question is *"What is the right semantics?"* I.e. we are looking first for a semantics that correctly fits to

our intuitions and treats the various KR-Tasks in the right (or appropriate) way. It should allow us to jump to conclusions even when only little information is available. Here it is of secondary interest how such a semantics can be implemented with a procedural calculus. Interesting overviews are [Minker, 1993; Minker, 1996] and [Dix, 1995c; Dix *et al.*, 2001a].

The LP-approach is dealt with in Section 3.1. It is still very near to classical predicate logic—default negation is interpreted as *finite-failure*. To get a stronger semantics, we interpret "not" as *failure* in Section 3.2. The main difference is that the principle *Elimination of Tautologies* holds. We then introduce a principle GPPE which is related to partial evaluation. In KR one can see this principle as allowing for definitional extensions—names or abbreviations can be introduced without changing the semantics.

All these principles do not yet determine a unique semantics—there is still room for different semantics and a lot of them have been defined in the last years. We do not want to present the whole zoo of semantics nor to discuss their merits or shortcomings. We refer the reader to the overview articles [Apt and Bol, 1994] and [Dix, 1995c] and the references given therein. We focus on the two main competing approaches that still have survived. These are the wellfounded semantics WFS (Section 3.3) and the stable semantics STABLE (Section 3.4). Finally, in Section 3.5 we discuss complexity and expressibility results for the semantics presented so far.

3.1 Negation-as-Finite-Failure

The idea of negation treated as *finite-failure* can be best illustrated by still considering definite programs, but queries containing default-atoms. How should we handle such default-atoms by modifying our SLD-resolution? Let us try this:

- If we reach a default-atom "not A" as a subgoal of our original query, we keep the current SLD-tree in mind and start a new SLD-tree by trying to solve "A".
- If this succeeds, then we falsified "not A", the current branch is failing and we have to backtrack and consider a different subquery.
- But it can also happen that the SLD-tree for "A" is *finite with only failing branches*. Then we say that A *finitely fails*, we turn back to our original SLD-tree, consider the subgoal "not A" as successfully solved and go on with the next subgoal in the current list.

It is important to note that an SLD-tree for a positive atom can fail without *being finite*. The SLD-tree for the program consisting of the single rule $p \leftarrow p$ with respect to the query p is infinite but failing (it consists of one single

infinite branch). In Figure 1 on page 9 the leftmost branch is also failing but infinite.

Although this idea of *finite-failure* is very procedural in nature, there is a nice model theoretical counterpart—Clark's completion *comp*(P) ([Clark, 1978]). Clark's idea was that a program P consists not only of the implications, but also of the information that *these are the only ones.* Roughly speaking, he argues that one should interpret the "$\leftarrow$"-arrows in rules as equivalences "$\equiv$" in classical logic. We do not give the exact definitions here, as they are very complex; in the non-propositional case, a symbol for equality, together with axioms describing it,[6] has to be introduced. However, for the propositional case, *comp*(P) is obtained from P by just

1. collecting all given clauses with the same head into one new "clause" with this respective head and a disjunctive body (containing all bodies of the old clauses), and

2. replacing the implication-symbols "$\leftarrow$" by "$\equiv$".

DEFINITION 12 (Clark's Completion *comp*(P)).
Clark's semantics for a program P is given by the set of all classical models of the theory *comp*(P).

We can now see the classical theory *comp*(P) as the information contained in the program P. *comp*(P) is like a sort of closed world assumption applied to P. We are now able to derive negative information from P by deriving it from *comp*(P). In fact, the following soundness and completeness result for definite programs P and definite queries $Q = \bigwedge_i A_i$ (consisting of only *positive* atoms) holds:

THEOREM 13 (COMP and Fair FF-Trees).
The following conditions are equivalent:

- *comp*$(P) \models \forall \neg Q$
- *Every fair SLD-tree for P with respect to Q is finitely failed.*

Note that in the last theorem we did not use default negation but classical negation $\neg$ because we just mapped all formulae into classical logic. We need the fairness assumption to ensure that the selection of atoms is reasonably well-behaving: we want that every atom or default-atom occurring in the list of preliminary goals will eventually be selected.

But even this result is still very weak—after all we want to handle not only negative queries but programs containing default-atoms. From now on

[6] CET: Clark's Equational Theory. CET($\mathcal{L}_P$) axiomatizes the equality theory of all Herbrand($\mathcal{L}_P$)-models. See [Mancarella *et al.*, 1988; Shepherdson, 1992] for the problem of equality and the underlying language.