ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/277670164

History of Logic Programming

Chapter · January 2014

CITATIONS
10

READS
1,452

1 author:

Robert Kowalski
Imperial College London
157 PUBLICATIONS **14,925** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Logical English View project

Project Non-modal deontic logic View project

All content following this page was uploaded by Robert Kowalski on 04 June 2015.

The user has requested enhancement of the downloaded file.

LOGIC PROGRAMMING

Robert Kowalski

1 INTRODUCTION

The driving force behind logic programming is the idea that a single formalism suffices for both logic and computation, and that logic subsumes computation. But logic, as this series of volumes proves, is a broad church, with many denominations and communities, coexisting in varying degrees of harmony. Computing is, similarly, made up of many competing approaches and divided into largely disjoint areas, such as programming, databases, and artificial intelligence.

On the surface, it might seem that both logic and computing suffer from a similar lack of cohesion. But logic is in better shape, with well-understood relationships between different formalisms. For example, first-order logic extends propositional logic, higher-order logic extends first-order logic, and modal logic extends classical logic. In contrast, in Computing, there is hardly any relationship between, for example, Turing machines as a model of computation and relational algebra as a model of database queries. Logic programming aims to remedy this deficiency and to unify different areas of computing by exploiting the greater generality of logic. It does so by building upon and extending one of the simplest, yet most powerful logics imaginable, namely the logic of Horn clauses.

In this paper, which extends a shorter history of logic programming (LP) in the 1970s [Kowalski, 2013], I present a personal view of the history of LP, focusing on logical, rather than on technological issues. I assume that the reader has some background in logic, but not necessarily in LP. As a consequence, this paper might also serve a secondary function, as a survey of some of the main developments in the logic of LP.

Inevitably, a history of this restricted length has to omit a number of important topics. In this case, the topics omitted include meta LP, high-order LP, concurrent LP, disjunctive LP and complexity. Other histories and surveys that cover some of these topics and give other perspectives include [Apt and Bol, 1994; Brewka *et al.*, 2011; Bry *et al.* 2007; Ceri *et al.*, 1990; Cohen, 1988; Colmerauer and Roussel, 1996; Costantini, 2002; Dantsin *et al.*, 1997; Eiter *et al.*, 2009; Elcock, 1990; van Emden, 2006; Hewitt, 2009; Minker, 1996; Ramakrishnan and Ullman, 1993].

Perhaps more significantly and more regrettably, in omitting coverage of technological issues, I may be giving a misleading impression of their significance. Without Colmerauer's practical insights [Colmerauer *et al.*, 1973], Boyer and Moore's

[1972] structure sharing implementation of resolution [Robinson, 1965a], and Warren's abstract machine and Prolog compiler [Warren, 1978, 1983; Warren *et al.*, 1977], logic programming would have had far less impact in the field of Computing, and this history would not be worth writing.

1.1 The Horn clause basis of logic programming

Horn clauses are named after the logician Alfred Horn, who studied some of their mathematical properties. A *Horn clause logic program* is a set of sentences (or clauses) each of which can be written in the form:

$$A_0 \leftarrow A_1 \wedge \ldots \wedge A_n \text{ where } n \geq 0.$$

Each A_i is an atomic formula of the form $p(t_1, ..., t_m)$, where p is a predicate symbol and the t_i are terms. Each term is either a constant symbol, variable, or composite term of the form $f(t_1, ..., t_m)$, where f is a function symbol and the t_i are terms. Every variable occurring in a clause is universally quantified, and its scope is the clause in which the variable occurs. The backward arrow $\leftarrow$ is read as "if", and $\wedge$ as "and". The atom A_0 is called the *conclusion* (or *head*) of the clause, and the conjunction $A_1 \wedge ... \wedge A_n$ is the *body* of the clause. The atoms $A_1, ..., A_n$ in the body are called *conditions*. If $n = 0$, then the body is equivalent to *true*, and the clause $A_0 \leftarrow$ *true* is abbreviated to A_0 and is called a *fact*. Otherwise if $n \neq 0$, the clause is called a *rule*.

It is also useful to allow the head A_0 of a clause to be *false*, in which case, the clause is abbreviated to $\leftarrow A_1 \wedge ... \wedge A_n$ and is called a *goal clause*. Intuitively, a goal clause can be understood as denying that the *goal* $A_1 \wedge ... \wedge A_n$ has a solution, thereby issuing a challenge to refute the denial by finding a solution.

Predicate symbols represent the relations that are defined (or computed) by a program, and functions are treated as a special case of relations, as in relational databases. Thus the mother function, exemplified by *mother(john) = mary*, is represented by a fact such as *mother(john, mary)*. The definition of maternal grandmother, which in functional notion is written as an equation:

$$\textit{maternal-grandmother}(X) \;=\; \textit{mother}(\textit{mother}(X))$$

is written as a rule in relational notation:

$$\textit{maternal-grandmother}(X) \leftarrow \textit{mother}(X, Z) \wedge \textit{mother}(Z, Y)^1$$

Although all variables in a rule are universally quantified, it is often more natural to read variables in the conditions that are not in the conclusion as existentially quantified with the body of the rule as their scope. For example, the following two sentences are equivalent:

[1] In this paper, I use the Prolog notation for clauses: Predicate symbols, function symbols and constants start with a lower case letter, and variables start with an upper case letter. Numbers can be treated as constants.

$$\forall XYZ\,[maternal\text{-}grandmother(X) \leftarrow \quad mother(X,Z) \wedge mother(Z,Y)]$$
$$\forall XY \quad [maternal\text{-}grandmother(X) \leftarrow \exists Z\,[mother(X,Z) \wedge mother(Z,Y)]]$$

Function symbols are not used for function definitions, but are used to construct composite data structures. For example, the composite term $cons(s,t)$ can be used to construct a list with first element s followed by the list t. Thus the term $cons(john, cons(mary, nil))$ represents the list $[john, mary]$, where nil represents the empty list.

Terms can contain variables, and logic programs can compute input-output relations containing variables. However, for the semantics, it is convenient to regard terms that do not contain variables, called *ground terms*, as the basic data structures of logic programs. Similarly, a clause or other expression is said to be *ground*, if it does not contain any variables.

Logic programs that do not contain function symbols are also called *Datalog* programs. Datalog is more expressive than relational databases, but is also decidable. Horn clause programs with function symbols have the expressive power of Turing machines, and consequently are undecidable. Horn clauses are sufficient

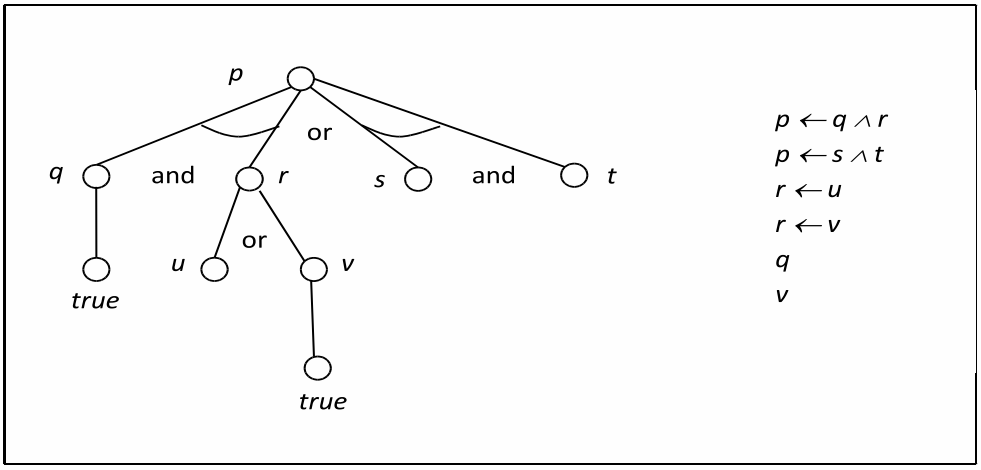

Figure 1. An and-or tree and corresponding propositional Horn clause program.

for many applications in artificial intelligence. For example, and-or trees can be represented by ground Horn clauses.[2] See figure 1.

[2]And-or trees were employed in many early artificial intelligence programs, including the geometry theorem proving machine of Gelernter [1963]. Search strategies for and-or trees were investigated by Nils Nilsson [1968], and in a theorem-proving context by Kowalski [1970].

1.2 Logic programs with negation

Although Horn clauses are the underlying basis of LP and are theoretically sufficient for all programming and database applications, they are not adequate for artificial intelligence, most importantly because they fail to capture non-monotonic reasoning. For non-monotonic reasoning, it is necessary to extend Horn clauses to clauses of the form:

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

Each A_i and B_i is an atomic formula, and "*not*" is read as *not*. Atomic formulas and their negations are also called *literals*. Here the A_i are *positive* literals, and the *not* B_i are *negative* literals. Sets of clauses in this form are called *normal logic programs*, or just *logic programs* for short.

Normal logic programs, with appropriate semantics for negation, are sufficient to solve the *frame problem* in artificial intelligence. Here is a solution using an LP representation of the situation calculus [McCarthy and Hayes, 1969]:

$$\mathit{holds}(F, \mathit{do}(A, S)) \leftarrow \mathit{poss}(A, S) \wedge \mathit{initiates}(A, F, S)$$
$$\mathit{holds}(F, \mathit{do}(A, S)) \leftarrow \mathit{poss}(A, S) \wedge \mathit{holds}(F, S) \wedge \mathit{not}\ \mathit{terminates}(A, F, S)$$

Here $\mathit{holds}(F, S)$ expresses that a fact F (also called a *fluent*) holds in a state (or *situation*) S; $\mathit{poss}(A, S)$ that the action A is possible in state S; $\mathit{initiates}(A, F, S)$ that the action A performed in state S initiates F in the resulting state $\mathit{do}(A, S)$; and $\mathit{terminates}(A, F, S)$ that A terminates F. Together, the two clauses assert that a fact holds in a state either if it is initiated by an action or if it held in the previous state and was not terminated by an action.

This representation of the situation calculus also illustrates meta-logic programming, because the predicates *holds*, *poss*, *initiates* and *terminates* can be understood as meta-predicates, where the variable F ranges over names of sentences. Alternatively, they can be interpreted as second-order predicates, where F ranges over first-order predicates.

1.3 Logic programming issues

In this article, I will discuss the development of LP and its extensions, their semantics, and their proof theories. We will see that lurking beneath the deceptively simple syntax of logic programs are profound issues concerning semantics, proof theory and knowledge representation.

For example, what does it mean for a logic program P to solve a goal G? Does it mean that P logically implies G, in the sense that G is true in all models of P? Does it mean that some larger theory than P, which includes assumptions implicit in P, logically implies G? Or does it mean that G is true in some natural, intended model of P?

And how should G be solved? *Top-down* by using the clauses in P as goal-reduction procedures, to reduce goals that match the conclusions of clauses to

sub-goals that correspond to their conditions? Or *bottom-up* to generate new conclusions from conditions, until the generated conclusions include all the information needed to solve the goal G in one step?

We will see that these two issues — what it means to solve a goal G, and whether to solve G top-down or bottom-up — are related. In particular, bottom-up reasoning can be interpreted as generating a model in which G is true.

These issues are hard enough for Horn clause programs. But they are much harder for logic programs with negative conditions. In some semantics, a negative condition *not* B has the same meaning as classical negation $\neg B$, and solving a negative goal *not* B is interpreted as reasoning with $\neg B$. But in most proof theories, *not* B is interpreted as some form of *negation as failure*:

not B holds if all attempts to show B fail.

In addition to these purely logical problems concerning semantics and proof theory, LP has been plagued by controversies concerning declarative versus procedural representations. Declarative representations are naturally supported by bottom-up model generation. But both declarative and procedural representations can be supported by top-down execution. For many advocates of purely declarative representations, such exploitation of procedural representations undermines the logic programming ideal.

These issues of semantics, proof theory and knowledge representation have been a recurring theme in the history of LP, and they continue to be relevant today. They are reflected, in particular, by the growing divergence between Prolog-style systems that employ top-down execution and answer set programming and Datalog systems that employ bottom-up model generation.

2 THE HISTORICAL BACKGROUND

The discovery of the top-down method for executing logic programs occurred in the summer of 1972, as the result of my collaboration with Alain Colmerauer in Marseille. Colmerauer was developing natural language question-answering systems, and I was developing resolution theorem-provers, and trying to reconcile them with procedural representations of knowledge in artificial intelligence.

2.1 Resolution

Resolution was developed by John Alan Robinson [1965a] as a technique for automated theorem-proving, with a view to mechanising mathematical proofs. It consists of a single inference rule for proving that a set of assumptions P logically implies a theorem G. The resolution method is a *refutation procedure*, which does so by *reductio ad absurdum*, converting P and the negation $\neg G$ of the theorem into a set of clauses and deriving the empty clause, which represents *falsity*.

Clauses are disjunctions of literals. In Robinson's original definition, clauses were represented as sets. In the propositional case:

given two clauses $\{A\} \cup F$ and $\{\neg A\} \cup G$
the *resolvent* is the clause $F \cup G$.

The two clauses $\{A\} \cup F$ and $\{\neg A\} \cup G$ are said to be the *parents* of the resolvent, and the literals A and $\neg A$ are said to be the literals *resolved upon*. If F and G are both empty, then the resolvent of $\{A\}$ and $\{\neg A\}$ is the empty clause, representing a contradiction or *falsity*.

In the first-order case, in which all variables are universally quantified with scope the clause in which they occur, it is necessary to unify sets of literals to make them complementary:

given two clauses $K \cup F$ and $L \cup G$
the *resolvent* is the clause $F\theta \cup G\theta$.

where θ is a most general substitution of terms for variables that unifies the atoms in K and L, in the sense that $K\theta = \{A\}$ and $L\theta = \{\neg A\}$. It is an important property of resolution, which greatly contributes to its efficiency, that if there is any substitution that unifies K and L, then there is a most general such unifying substitution, which is unique up to renaming of variables.

The set representation of clauses (and sets of clauses) builds in the inference rules of commutativity, associativity and idempotency of disjunction (and conjunction). The resolution rule itself generalises *modus ponens*, *modus tollens*, disjunctive syllogism, and many other separate inference rules of classical logic. The use of the most general unifier, moreover, subsumes in one operation, the infinitely many inferences of the form "derive $P(t)$ from $\forall X P(X)$" that are possible with the inference rule of universal instantiation. Other inference rules are eliminated (or used) in the conversion of sentences of standard first-order logic into clausal form.

Set notation for clauses is not user-friendly. It is more common to write clauses $\{A_1, \ldots, A_n, \neg B_1, \ldots, \neg B_m\}$ as disjunctions $A_1 \vee \ldots \vee A_n \vee \neg B_1 \vee \ldots \vee \neg B_m$. However, sets of clauses, representing conjunctions of clauses, are commonly written simply as sets. Clauses can also be represented as conditionals in the form:

$$A_1 \vee \ldots \vee A_n \leftarrow B_1 \wedge \ldots \wedge B_m.$$

where $\leftarrow$ is material implication $\rightarrow$ (or $\supset$) written backwards.

The discovery of resolution revolutionised research on automated theorem proving, as many researchers turned their hands towards developing refinements of the resolution rule. It also inspired other applications of logic in artificial intelligence, most notably to the development of question-answering systems, which represent data or knowledge in logical form, and query that knowledge using logical inference. One of the most successful and most influential such system was QA3, developed by Cordell Green [1969].

In QA3, given a knowledge base and goal to be solved, both expressed in clausal form, an extra literal $answer(X)$ is added to the clause or clauses representing the negation of the goal, where the variables X represent some value

of interest in the goal. For example, to find the capital of the *usa*, the goal $\exists X capital(X, usa)$ is negated and the answer literal is added, turning it into the clause $\neg capital(X, usa) \vee answer(X)$. The goal is solved by deriving a clause consisting only of answer literals. The substitutions of terms for variables used in the derivation determine values for the variables X. In this example, if the knowledge base contains the clause $capital(washington, usa)$, the answer $answer(washington)$ is obtained in one resolution step.

Green also showed that resolution has many other problem-solving applications, including robot plan formation. Moreover, he showed how resolution could be used to automatically generate a program written in a conventional programming language, such as LISP, from a specification of its input-output relation written in the clausal form of logic. As he put it:

> "In general, our approach to using a theorem prover to solve programming problems in LISP requires that we give the theorem prover two sets of initial axioms:
>
> 1. Axioms defining the functions and constructs of the subset of LISP to be used
> 2. Axioms defining an input-output relation such as the relation $R(x, y)$, which is to be true if and only if x is any input of the appropriate form for some LISP program and y is the corresponding output to be produced by such a program."

Green also seems to have anticipated the possibility of dispensing with (1) and using only the representation (2) of the relation $R(x, y)$, writing:

> "The theorem prover may be considered an 'interpreter' for a high-level assertional or declarative language — logic. As is the case with most high-level programming languages the user may be somewhat distant from the efficiency of 'logic' programs unless he knows something about the strategies of the system."
>
> "I believe that in some problem solving applications the 'high-level language' of logic along with a theorem-proving program can be a quick programming method for testing ideas."

However, he does not seem to have pursued these ideas much further. Moreover, there was an additional problem, namely that the resolution strategies of that time behaved unintuitively and were very redundant and inefficient. For example, given a clause of the form $L_1 \vee \ldots \vee L_n$, and n clauses of the form $\neg L_i \vee C_i$, resolution would derive the same clause $C_1 \vee \ldots \vee C_n$ redundantly in $n!$ different ways.

2.2 Procedural representations of knowledge

Green's ideas fired the enthusiasm of researchers working in contact with him at Stanford and Edinburgh, but they also attracted fire from MIT, where researchers

were advocating procedural representations of knowledge. Terry Winograd's PhD thesis gave the most compelling and most influential voice to this opposition. Winograd [1971] argued (page 232):

> "Our heads don't contain neat sets of logical axioms from which we can deduce everything through a 'proof procedure'. Instead we have a large set of heuristics and procedures for solving problems at different levels of generality."

He quoted (pages 232-3) Green's own admission of some of the difficulties:

> "It might be possible to add strategy information to a predicate calculus theorem prover, but with current systems such as QA3, 'To change strategies in the current version, the user must know about set-of-support and other program parameters such as level bound and term depth. To radically change the strategy, the user presently has to know the LISP language and must be able to modify certain strategy sections of the program.' (<Green 1969>p. 236)."[3]

Winograd's procedural alternative to purely "uniform" logical representations was based on Carl Hewitt's language Planner. Winograd [1971] describes Planner in the following terms (page 238):

> "The language is designed so that if we want, we can write theorems in a form which is almost identical to the predicate calculus, so we have the benefits of a uniform system. On the other hand, we have the capability to add as much subject-dependent knowledge as we want, telling theorems about other theorems and proof procedures. The system has an automatic goal-tree backup system, so that even when we are specifying a particular order in which to do things, we may not know how the system will go about doing them. It will be able to follow our suggestions and try many different theorems to establish a goal, backing up and trying another automatically if one of them leads to a failure (see section 3.3)."

In contrast (page 215):

> "Most 'theorem-proving' systems do not have any way to include this additional intelligence. Instead, they are limited to a kind of 'working in the dark'. A uniform proof procedure gropes its way through the collection of theorems and assertions, according to some general procedure which does not depend on the subject matter. It tries to combine facts which might be relevant, working from the bottom-up."

[3] We will see later that the set-of-support strategy was critical, because it allowed QA3 to incorporate a form of backward reasoning from the theorem to be proved.

Winograd's PhD thesis presented a natural language understanding system that was a great advance at the time, and its advocacy of Planner was enormously influential. Even Stanford and Edinburgh were affected by these ideas.

Pat Hayes and I had been working in Edinburgh on a book [Hayes and Kowalski, 1971] about resolution theorem-proving, when he returned from a second visit to Stanford (after the first visit, during which he and John McCarthy wrote the famous situation calculus paper [McCarthy and Hayes, 1968]). He was greatly impressed by Planner, and wanted to rewrite the book to take Planner into account. I was not enthusiastic, and we spent many hours discussing and arguing about the relationship between Planner and resolution theorem proving. Eventually, we abandoned the book, unable to agree.

2.3 Resolution, part two

At the time that QA3 and Planner were being developed, resolution was not well understood. In particular, it was not understood that a proof procedure, in general, is composed of an inference system that defines the space of all proofs and a search strategy that explores the proof space looking for a solution of a goal. We can represent this combination as an equation:

$$\textit{proof procedure} = \textit{proof space} + \textit{search strategy}$$

A typical proof space has the structure of an and-or tree turned upside down. Typical search strategies include breadth-first search, depth-first search and some form of best-first or heuristic search.

In the case of the resolution systems at the time, the proof spaces were horrendously redundant, and most search strategies used breadth-first search. Attempts to improve efficiency focussed on restricting (or refining) the resolution rule without losing completeness, to reduce the size of the proof space. The best known refinements were hyper-resolution and set of support.

Hyper-resolution [Robinson, 1965b] is a generalised form of bottom-up (or forward) reasoning. In the propositional case, given an input clause:

$$D_0 \vee \neg B_1 \vee \ldots \vee \neg B_m$$

and m input or derived positive clauses:

$$B_1 \vee D_1,\ \ldots,\ B_m \vee D_m$$

where each B_i is an atom and each D_i is a disjunction of atoms, *hyper-resolution* derives the positive clause:

$$D_0 \vee D_1 \vee \ldots \vee D_m.$$

Bottom-up reasoning with Horn clauses is the special case in which D_0 is a single atom and each other D_i is an empty disjunction, equivalent to *false*. In this special

case, rewriting disjunctions as conditionals, hyper-resolution derives B_0 from the input clause:

$$B_0 \leftarrow B_1 \wedge \ldots \wedge B_m$$

and the input or derived facts, $B_1, \ldots, B_m$.

The problem with hyper-resolution, as Winograd observed, is that it derives new clauses from the input clauses, without paying attention to the problem to be solved. It is "uniform" in the sense that, given a theorem to be proved, it uniformly performs the same inferences bottom-up from the axioms, ignoring the theorem until it generates it, as if by accident.

In contrast with hyper-resolution, the set of support strategy [Wos *et al.*, 1965] focuses on a subset of clauses that are relevant to the problem at hand:

> A subset S' of an input set S of clauses is a *set of support* for S iff $S - S'$ is satisfiable. The *set of support strategy* restricts resolution so that at least one parent clause belongs to the set of support or is derived by the set of support restriction.

If the satisfiable set of clauses $S - S'$ represents a set of axioms, and the set of support S' represents the negation of a theorem, then the set of support strategy implements an approximation of top-down reasoning by *reductio ad absurdum*. It also ensures that any input clauses (or axioms) used in a derivation are *relevant* to the theorem, in the spirit of relevance logics [Anderson and Belnap, 1962].[4]

The set of support strategy only approximates top-down reasoning. A better approximation is obtained by linear resolution, which was discovered independently by Loveland [1970], Luckham [1970] and Zamov and Sharonov [1969]. Linear resolution addresses the problem of relevance by focusing on a top clause C_0, which could represent an initial goal:

> Let S be a set of clauses. A *linear derivation* of a clause C_n from a top clause $C_0 \in S$ is a sequence of clauses $C_0, ..., C_n$ such that every clause C_{i+1} is a resolvent of C_i with some input clause in S or with some ancestor clause C_j where $j < i$. (It was later realised that ancestor resolution is unnecessary if S is a set of Horn clauses.)

The top clause C_0 in a linear derivation can be restricted to one belonging to a set of support. The resulting space of all linear derivations from a given top clause C_0 has the structure of a proof tree whose nodes are clauses and whose branches are linear derivations. Using linear resolution to extend the derivation of a clause C_i to the derivation of a clause C_{i+1} generates the derived node C_{i+1} as a child of

[4] Another important case is the one in which $S - S'$ represents a database (or knowledge base) together with a set of integrity constraints that are satisfied by the database, and S' represents a set of updates to be added to the database. The set of support restriction then implements a form of bottom-up reasoning from the updates, to check that the updated database continues to satisfy the integrity constraints. Moreover, it builds in the assumption that the database satisfied the integrity constraints prior to the updates, and therefore if there is an inconsistency, the update must be "relevant" to the inconsistency.

the node C_i. The same node C_i can have different children C_{i+1}, corresponding to different linear resolutions.

In retrospect, the relationship with Planner is obvious. If the top clause C_0 represents an initial goal, then the tree of all linear derivations is a goal tree, and generating the tree top-down is a form of goal-reduction. The tree can be explored using different search strategies. Depth-first search, in particular, can be informed by Planner-like strategies that both specify "a particular order in which to do things", but also "back up" automatically in the case of failure.

The relationship with Planner was not obvious at the time. Even as recently as 2005, Paul Thagard in *Mind: Introduction to Cognitive Science*, compares logic unfavourably with production systems, stating on page 45:

> "In logic-based systems, the fundamental operation of thinking is logical deduction, but from the perspective of rule-based systems, the fundamental operation of thinking is search."[5]

But it wasn't just this lack of foresight that stood in the way of understanding the relationship with Planner: there was still the $n!$ redundant ways of resolving upon n literals in the clauses C_i. This redundancy was recognized and eliminated without the loss of completeness by Loveland [1972], Reiter [1971], and Kowalski and Kuehner [1971], independently at about the same time. The obvious solution was simply to resolve upon the literals in the clauses C_i in a single order. This order can be determined statically, by ordering the literals in the input clauses, and imposing the same order on the resolvents. Or it could be determined dynamically, as in selected linear (SL) resolution [Kowalski and Kuehner, 1971], by selecting a single literal to resolve upon in a clause C_i when the clause is chosen for resolution. Both methods eliminate redundancy, but dynamic selection can lead to smaller search spaces.[6]

Ironically, both Loveland [1972] and Kowalski and Kuehner [1971] also noted that linear resolution with an ordering restriction is equivalent to Loveland's [1968] earlier model elimination proof procedure. The original model elimination procedure was presented so differently that it took years even for its author to recognise the equivalence. The SL resolution paper also pointed out that the set of all SL derivations forms a search space, and described a heuristic search strategy for finding simplest proofs. In the conclusions, with implicit reference to Planner, it claimed:

[5] This claim makes more sense if Thagard, like Winograd before him, associates logic exclusively with forward reasoning. 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."

[6] Dynamic selection is useful, for example, to solve goals with different input-output arguments. For example, given the clause $p(X,Y) \leftarrow q(X,Z) \wedge r(Z,Y)$ and the goal $p(a,Y)$, then the subgoal $q(a,Z)$ should be selected before $r(Z,Y)$. But given the goal $p(X,b)$, the subgoal $r(Z,b)$ should be selected before $q(X,Z)$.

> "Moreover, the amenability of SL-resolution to the application of heuristic methods suggests that, on these grounds alone, it is at least competitive with theorem-proving procedures designed solely from heuristic considerations."

3 THE PROCEDURAL INTERPRETATION OF HORN CLAUSES

The development of various forms of linear resolution with set of support and ordering restrictions brought resolution systems closer to Planner-like theorem-provers. But these resolution systems did not yet have an explicit procedural interpretation.

3.1 The representation of grammars in logical form

Among the various confusions that prevented a better understanding of the relationship between logical and procedural representations was the fact that Winograd's thesis, which so advanced the Planner cause, employed a different procedural language Programmar, for natural language grammars. Moreover, Winograd's natural language understanding system was implemented in a combination of micro-Planner (a subset of Planner), Programmar and LISP. So it wasn't obvious whether Planner was supposed to be a general-purpose programming language, or a special purpose language for proving theorems, for writing plans or for some other purpose.

In the theorem-proving group in Edinburgh, where I was working at the time, much of the debate surrounding Planner focused on whether "uniform", resolution proof procedures are adequate for proving theorems, or whether they need to be augmented with Planner-like, domain-specific control information. In particular, I was puzzled by the relationship between Planner and Programmar, and began to investigate whether grammars could be written in a logical form. This was auspicious, because in the summer of 1971 Alain Colmerauer invited me for a short visit to Marseille.

Colmerauer knew everything there was to know about formal grammars and their application to programming language compilers. During 1967–1970 at the University of Montreal, he developed Q-systems [1969] as a rule-based formalism for processing natural language. Q-systems were later used on a daily basis from 1982 to 2001 to translate English weather forecasts into French for Environment Canada. Since 1970, he had been in Marseille, building up a team working on natural language question-answering, investigating SL-resolution for the question-answering component.

I arrived in Marseille, anxious to get Colmerauer's feedback on my preliminary ideas about representing grammars in logical form. My representation used a function symbol to concatenate words into strings of words, and axioms to express that concatenation is associative. It was obvious that reasoning with such associativity axioms was inefficient. Colmerauer immediately saw how to avoid the

axioms of associativity, in a representation that later came to be known as metamorphosis grammars [Colmerauer, 1975] (or definite clause grammars [Pereira and Warren, 1980]). We saw that different kinds of resolution applied to the resulting grammars give rise to different kinds of parsers. For example, forward reasoning with hyper-resolution performs bottom-up parsing, while backward reasoning with SL-resolution performs top-down parsing.[7]

3.2 Horn clauses and SLD-resolution

It was during my second visit to Marseille in April and May of 1972 that the idea of using SL-resolution to execute Horn clause programs emerged. By the end of the summer, Colmerauer's group had developed the first version of Prolog, and used it to implement a natural language question-answering system [Colmerauer *et al.*, 1973]. I reported an abstract of my own findings at the MFCS conference in Poland in August 1972 [Kowalski, 1972].[8]

The first Prolog system was an implementation of SL-resolution for the full clausal form of first-order logic, including ancestor resolution. But the idea that Horn clauses were an interesting case was already in the air. Donald Kuehner [1969], in particular, had already been working on bi-directional strategies for Horn clauses. However, the first explicit reference to the procedural interpretation of Horn clauses appeared in [Kowalski, 1974]. The abstract begins:

> "The interpretation of predicate logic as a programming language is based upon the interpretation of implications: B if A_1 *and*... *and* A_n as procedure declarations, where B is the procedure name and A_1 *and* ... *and* A_n is the set of procedure calls constituting the procedure body."

The theorem-prover described in the paper is a variant of SL-resolution, to which Maarten van Emden later attached the name SLD-resolution, standing for "selected linear resolution with definite clauses":

> A *definite clause* is a Horn clause of the form $B \leftarrow B_1 \wedge \ldots \wedge B_n$.
>
> A *goal clause* is a Horn clause of the form $\leftarrow A_1 \wedge \ldots \wedge A_n$.
>
> Given a goal clause $\leftarrow A_1 \wedge \ldots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \ldots \wedge A_n$ with selected atom A_i and a definite clause $B \leftarrow B_1 \wedge \ldots \wedge B_m$, where θ is a most general substitution that unifies A_i and B, the *SLD-resolvent* is the goal clause $\leftarrow (A_1 \wedge \ldots \wedge A_{i-1} \wedge B_1 \wedge \ldots \wedge B_m \wedge A_{i+1} \wedge \ldots \wedge A_n)\theta$.
>
> Given a set of definite clauses S and an initial goal clause C_0, an *SLD-derivation* of a goal clause C_n is a sequence of goal clauses $C_0, \ldots, C_n$

[7] However, Colmerauer [1991] remembers coming up with the alternative representation of grammars, not during my visit in 1971, but after my visit in 1972.

[8] In the abstract, I used a predicate $val(f(X), Y)$ instead of a predicate $f(X, Y)$, using Phillip Roussel's idea of *val* as "formal equality". Roussel was Colmerauer's PhD student and the main implementer of the first Prolog system.

such that every C_{i+1} is the SLD-resolvent of C_i with some input clause in S.

An *SLD-refutation* is an SLD-derivation of the empty clause.

SLD-resolution is more flexible than SL-resolution restricted to Horn clauses.[9] In SL-resolution the atoms A_i must be selected last-in-first-out, but in SLD-resolution, there is no restriction on their selection. Both refinements of linear resolution avoid the redundancy of unrestricted linear resolution, and both are complete, in the sense that if a set of Horn clauses is unsatisfiable, then there exists both an SL-resolution refutation and an SLD-resolution refutation in their respective search spaces. In both cases, different selection strategies give rise to different, complete search spaces. But the more flexible selection strategy of SLD-resolution means that search spaces can be smaller, and therefore more efficient to search.

In SLD resolution, goal clauses have a dual interpretation. In the strictly logic interpretation, the symbol $\leftarrow$ in a goal clause $\leftarrow A_1 \wedge \ldots \wedge A_n$ is equivalent to classical negation; the empty clause is equivalent to falsity; and a refutation indicates that the top clause is inconsistent with the initial set of clauses S.

However, in a problem-solving context, it is natural to think of the symbol $\leftarrow$ in a goal clause $\leftarrow A_1 \wedge \ldots \wedge A_n$ as a question mark ? or command !, and the conjunction $A_1 \wedge \ldots \wedge A_n$ as a set of subgoals, whose variables are all existentially quantified. The empty clause represents an empty set of subgoals, and a "refutation" indicates that the top clause has been solved. The solution is represented by the substitutions of terms for variables in the top clause, generated by the most general unifiers used in the refutation — similar to, but without the answer literals of QA3.

As in the case of linear resolution more generally, the space of all SLD-derivations with a given top clause has the structure of a goal tree, which can be explored using different search strategies. From a logical point of view, it is desirable that the search strategy be complete, so that the proof procedure is guaranteed to find a solution if there is one in the search space. Complete search strategies include breadth-first search and various kinds of best-first and heuristic search. Depth-first search is incomplete in the general case, but it takes up much less space than the alternatives. Moreover, it is complete if the search space is finite, or if there is only one infinite branch that is explored after all of the others.

Notice that there are two different, but related notions of completeness: one for search spaces, and the other for search strategies. A search space is *complete* if it contains a solution whenever the semantics dictates that there is a solution; and a search strategy is *complete* if it finds a solution whenever there is one in the search space. For a proof procedure to be *complete*, both its search space and its search strategy need to be complete.

[9] If SL-resolution is applied to Horn clauses, with a goal clause as top clause, then ancestor resolution is not possible, because all clauses in the same SL-derivation are then goal clauses, which cannot be resolved with one another.

The different options for selecting atoms to resolve upon in SLD-resolution and for searching the space of SLD-derivations were left open in [Kowalski, 1974], but were pinned down in the Marseille Prolog interpreter. In Prolog, subgoals are selected last-in-first-out in the order in which the subgoals are written, and branches of the search space are explored depth-first in the order in which the clauses are written. By choosing the order in which subgoals and clauses are written, a Prolog programmer can exercise considerable control over the efficiency of a program.

3.3 Logic + Control

In those days, it was widely believed that logic alone is inadequate for problem-solving, and that some way of controlling the theorem-prover is needed for efficiency. Planner combined logic and control in a procedural representation that made it difficult to identify the logical component. Logic programs with SLD-resolution also combine logic and control, but make it possible to read the same program both logically and procedurally. I later expressed this as Algorithm = Logic + Control ($A = L + C$) [Kowalski, 1979a], influenced by Pat Hayes' [1973] Computation = Controlled Deduction.

The most direct implication of the equation is that, given a fixed logical representation L, different algorithms can be obtained by applying different control strategies, i.e. $A_1 = L + C_1$ and $A_2 = L + C_2$. Pat Hayes [1973], in particular, argued that logic and control should be expressed in separate languages, with the logic component L providing a pure, declarative specification of the problem, and the control component C supplying the problem solving strategies needed for an efficient algorithm A. Moreover, he argued against the idea, expressed by $A_1 = L_1 + C$ and $A_2 = L_2 + C$, of using a fixed control strategy C, as in Prolog, and formulating the logic L_i of the problem to obtain a desired algorithm A_i.

This idea of combining logic and control in separate object and meta-level languages has been a recurring theme in the theorem-proving and AI literature. It was a major influence, for example, on the development of PRESS, which solved equations by expressing the rules of algebra in an object language, and the rules for controlling the search for solutions in a meta-language. According to its authors, Alan Bundy and Bob Welham [1981]:

> "PRESS consists of a collection of predicate calculus clauses which together constitute a Prolog program. As well as the procedural meaning attached to these clauses, which defines the behaviour of the PRESS program, they also have a declarative meaning - that is, they can be regarded as axioms in a logical theory."

In retrospect, PRESS was an early example of a now common use of Prolog to write meta-interpreters.

But most applications do not need such an elaborate combination of logic and control. For example, the meta-level control program in PRESS does not need a

meta-meta-level control program. In fact, for some applications, even the modest control available to the Prolog programmer is unnecessary For these applications, it suffices for the programmer to specify only the logic of the problem, and to leave it to Prolog to solve the problem without any help

But often, leaving it to Prolog alone can result, not only in unacceptable inefficiency, but even in non-terminating failure to find a solution. Here is a simple example, written in Prolog notation, where :- stands for ← and every clause ends in a full stop:

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

Prolog fails to find the solution $X = bob$, because it explores the infinite branch generated by repeatedly using the first clause, without getting a chance to explore the branch generated by the second clause. If the order of the two clauses is reversed, Prolog finds the solution. 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. Perhaps the easiest way to avoid such infinite loops in ordinary Prolog is to write a meta-interpreter, as in PRESS. [10]

Problems and inefficiencies with the Prolog control strategy led to numerous proposals for LP languages incorporating enhanced control features. Some of them, such as Colmerauer's [1982] Prolog II, which allowed insufficiently instantiated subgoals to be suspended, were developed as extensions of Prolog. Other proposals that departed more dramatically from ordinary Prolog included the use of coroutining in IC-Prolog [Clark *et al.*, 1972] selective backtracking [Bruynooghe and Pereira, 1984] and meta-level control for logic programs [Gallaire and Lasserre, 1982; Pereira, 1984]

IC-Prolog, in particular, led to the development by Clark and Gregory [1983, 1986] of the concurrent logic programming language Parlog, which led in turn to numerous variants of concurrent LP languages, one of which KL1, developed by Kazunori Ueda [1986], was adopted as the basis for the systems software of the Fifth Generation Computer Systems (FGCS) Project in Japan.

The FGCS Project was a ten year project beginning in 1982, sponsored by Japan's Ministry of International Trade and Industry and involving all the major Japanese computer manufacturers. Its main objective was to develop a new generation of computers employing massive parallelism and oriented towards artificial intelligence applications. From the start of the project, logic programming was identified as the preferred software technology.

[10] In other cases, much simpler solutions are often possible. For example, to avoid infinite loops with the program $path(X, X)$ and $path(X, Y) \leftarrow link(X, Z) \wedge path(Z, Y)$, it suffices to add an extra argument to the *path* predicate to record the list of nodes visited so far, and to add an extra condition to the second clause to check that the node Z in $link(X, Z)$ is not in this path. For some advocates of declarative programming this is considered cheating. For others, it illustrates a practical application of $A = L_1 + C_1 = L_2 + C_2$.

The FGCS project did not achieve its objectives, and all three of its main areas of research — parallel hardware, logic programming software, and AI applications — suffered a world-wide decline.

These days, however, there is growing evidence that the FGCS project was ahead of its time. In the case of logic programming, in particular, SLD-resolution extended with tabling [Tamaki and Sato, 1986; Sagonas *et al.*, 1994; Chen and Warren, 1996; Telke and Liu, 2011] avoids many infinite loops, like the one in the example above. Moreover, there also exist alternative techniques for executing logic programs that do not rely upon the procedural interpretation, including the model generation methods of Answer Set programming (ASP) and the bottom-up execution strategies of Datalog.

ASP and Datalog have greatly advanced the ideal of purely declarative representations, relegating procedural representations to the domain of imperative languages and other formalisms of dubious character. However, not everyone is convinced that purely declarative knowledge representation is adequate either for practical computing or for modelling human reasoning.

Thagard [2005], for example, claims that the following, useful procedure cannot easily be expressed in logical terms (page 45):

If you want to go home and you have the bus fare, then you can catch a bus.

On the contrary, the sentence can be expressed literally in the logical form:

$$can(you, catch\text{-}bus) \leftarrow want(you, go\text{-}home) \wedge have(you, bus\text{-}fare)$$

But this rendering requires the use of modal operators or modal predicates for *want* and *can*. More importantly, it misses the real logic of the procedure:

$$go(you, home) \leftarrow have(you, bus\text{-}fare) \wedge catch(you, bus).$$

Top-down reasoning applied to this logic generates the procedure, without sacrificing either the procedure or the declarative belief that justifies it

4 THE SEMANTICS OF HORN CLAUSE PROGRAMS

The earliest influences on the development of logic programming had come primarily from automated theorem-proving and artificial intelligence. But researchers in the School of AI in Edinburgh also had strong interests in the theory of computation, and there was a lot of excitement about Dana Scott's [1970] recent fixed point semantics for programming languages. Maarten van Emden suggested that we investigate the application of Scott's ideas to Horn clause programs and that we compare the fixed point semantics with the logical semantics.

4.1 What is the meaning of a program?

But first we needed to establish a common ground for the comparison. If we identify the data structures of a logic program P with the set of all ground terms

constructible from the vocabulary of P, also called the *Herbrand universe* of P, then we can view the "meaning" (or denotation) of P as the set of all ground atoms A that can be derived from P[11], which is expressed by:

$$P \vdash A.$$

Here $\vdash$ can represent any derivability relation. Viewed in programming terms, this is analogous to the operational semantics of a programming language. But viewed in logical terms, this is a proof-theoretic definition, which is not a semantics at all. In logical terms, it is more natural to understand the semantics of P as given by the set of all ground atoms A that are logically implied by P, written:

$$P \models A$$

The operational and model-theoretic semantics are equivalent for any sound and complete notion of derivation – the most important kinds being top-down and bottom-up.

Top-down derivations include model-elimination, SL-resolution and SLD-resolution. Model-elimination and SL-resolution are sound and complete for arbitrary clauses. So they are sound and complete for Horn clauses in particular. Moreover, ancestor resolution is impossible for Horn clauses. So model-elimination and SL-resolution without ancestor resolution are sound and complete for Horn clause programs.

The selection rule in both SL-resolution and SLD-resolution constructs a linear representation of an and-tree proof. In SL-resolution the linear representation is obtained by traversing the and-tree depth-first. In SLD-resolution the linear representation can be obtained by traversing the and-tree in any order.[12] The completeness of SLD-resolution was first proved by Robert Hill [1974].

Bottom-up derivations are a special case of hyper-resolution, which is also sound and complete for arbitrary clauses, and therefore for Horn clauses as well. Moreover, as we soon discovered, they are equivalent to the fixed point semantics.

4.2 Fixed point semantics

In Dana Scott's [1970] fixed point semantics, the denotation of a recursive function is given by its input-output relation. The denotation is constructed by approximation, starting with the empty relation, repeatedly plugging the current approximation of the denotation into the definition of the function, transforming the approximation into a better one, until the complete denotation is obtained in the limit, as the least fixed point.

[11] Notice that this excludes programs which represent perpetual processes. Moreover, it ignores the fact that, in practice, logic programs can compute input-output relations containing variables. This is sometimes referred to as the "power of the logical variable".

[12] Note that and-or trees suggest other strategies for executing logic programs, for example by decomposing goals into subgoals top-down, searching for solutions of subgoals in parallel, then collecting and combining the solutions bottom-up. This is like the MapReduce programming model used in Google [Dean and Ghemawat, 2008].

Applying the same approach to a Horn clause program P, the fixed point semantics uses a similar transformation T_P, called the *immediate consequence operator*, to map a set I of ground atoms representing an approximation of the input-output relations of P into a more complete approximation $T_P(I)$:

$$T_P(I) = \{A_0 \mid A_0 \leftarrow A_1 \wedge \ldots \wedge A_n \in \mathit{ground}(P) \text{ and } \{A_1, \ldots, A_n\} \subseteq I\}.$$

Here $\mathit{ground}(P)$ is the set of all ground instances of the clauses in P over the Herbrand universe of P. The application of T_P to I is equivalent to applying one step of hyper-resolution to the clauses in $\mathit{ground}(P) \cup I$.

Not only does every Horn clause program P have a fixed point I such that $T_P(I) = I$, but it has a *least fixed point*, $\mathit{lfp}(T_P)$, which is the denotation of P according to the *fixed point semantics*. The least fixed point is also the smallest set of ground atoms I closed under T_P, i.e. the smallest set I such that $T_P(I) \subseteq I$. This alternative characterisation provides a link with the minimal model semantics, as we will see below.

The least fixed point can be constructed, as in Scott's semantics, by starting with the empty set $\{\}$ and repeatedly applying T_P:

$$\text{If } T_P^0 = \{\} \text{ and } T_P^{i+1} = T_P(T_P^i), \text{ then } \mathit{lfp}(T_P) = \cup_{0 \leq i} T_P^i.$$

The result of the construction is equivalent to the set of all ground atoms that can be derived by applying any number of steps of hyper-resolution to the clauses in $\mathit{ground}(P)$.

The equality $\mathit{lfp}(T_P) = \cup_{0 \leq i} T_P^i$ is usually proved in fixed point theory by appealing to the Tarski-Knaster theorem. However, in [van Emden and Kowalski, 1976], we showed that the equivalence follows from the completeness of hyper-resolution and the relationship between least fixed points and minimal models. Here is a sketch of the argument:

$A \in \mathit{lfp}(T_P)$ iff $A \in \mathit{min}(P)$
i.e. least fixed points and minimal models coincide.

$A \in \mathit{min}(P)$ iff $P \vDash A$
i.e. truth in the minimal model and all models coincide.

$P \vDash A$ iff $A \in \cup_{0 \leq i} T_P^i$
i.e. hyper-resolution is complete.

4.3 Minimal model semantics

The minimal model semantics was inspired by the fixed point semantics, but it was based on the notion of Herbrand interpretation. The key idea of Herbrand interpretations is to identify an interpretation of a set of sentences with the set of all ground atomic sentences that are true in the interpretation.

In a Herbrand interpretation, the domain of individuals is the set of ground terms in the Herbrand universe of the language. A *Herbrand interpretation* is any

subset of the *Herbrand base*, which is the set of all ground atoms of the language. The most important property of Herbrand interpretations is that, in first-order logic, a set of sentences has a model if and only if it has a Herbrand model. This property is a form of the Skolem-Löwenheim-Herbrand theorem.[13]

Thus the model-theoretic denotation of a Horn clause program:

$$M(P) = \{A \mid A \text{ is a ground atom and } P \vDash A\}$$

is actually a Herbrand interpretation of P in its own right. Moreover, it is easy to show that $M(P)$ is also a Herbrand model of P. In fact, it is the smallest Herbrand model $min(P)$ of P. Therefore:

$$A \in min(P) \text{ iff } P \vDash A.$$

It is also easy to show that the Herbrand models of P coincide with the Herbrand interpretations that are closed under the operator T_P, i.e.:

$$I \text{ is a Herbrand model of } P \text{ iff } T_P(I) \subseteq I.$$

This is because the immediate consequence operator mimics, not only hyper-resolution, but also the definition of truth for Horn clauses: A set of Horn clauses P is true in a Herbrand interpretation I if and only if, for every ground instance $A_0 \leftarrow A_1 \wedge \ldots \wedge A_n$ of a clause in P, A_0 is true in I if $A_1, \ldots, A_n$ are true in I.

It follows that the least fixed point and the minimal model are identical:

$$lfp(T_P) = min(P).$$

4.4 Computability

The logicians Andréka and Németi visited Edinburgh in 1975, and wrote a report, published in [Andréka and Németi, 1978], proving the Turing completeness of Horn clause logic. Sten-Åke Tärnlund [1977] obtained a similar result independently. It was a great shock, therefore, to learn that Raymond Smullyan [1956] had already published an equivalent result. Here is the complete abstract:

> A new approach to recursive enumerability is considered based on the notion of "minimal models". A formula of the lower functional calculus of the form $F_1 \cdot F_2 \cdots F_{n-1} \cdot \supset \cdot F_n$ (or F_1 alone, if $n = 1$) in which each F_i is atomic, and F_n contains no predicate constants, is termed *regular*. Let A be a finite set of regular formulae; Σ a collection of

[13] The property can be proved in two steps: First, convert S into clausal form by using "Skolem" functions to eliminate existential quantifiers. Although the resulting set S' of clauses and S are not equivalent, S has a model iff S' has a model. A set of clauses S' has a model iff S' has a Herbrand model M, constructed using the Herbrand universe of S'. Therefore S has a model if and only if it has a Herbrand model M. (Contrary claims in the literature that S may have a model, but no Herbrand model, are based on the assumption that the Herbrand interpretations of S are constructed using the Herbrand universe of S.)

> sets and relations, on some universe U; I an interpretation of the predicate constants (occurring in A) as elements of Σ. The ordered triple $\mathcal{L}$ viz. (A, U, I) is a *recursive logic* over Σ. A *model* of $\mathcal{L}$ is an interpretation of the predicate *variables* P_i in which each formula of A is valid. Let P_i^* be the intersection of all attributes assignable to P_i in some model; these P_i^* are called *definable* in $\mathcal{L}$. If each P_i is interpreted as P_i^*, it can be proved that there is a model — this is the *minimal model*. Sets definable in some $\mathcal{L}$ over Σ are termed *recursively definable* from Σ. It is proved: (1) the recursively enumerable sets are precisely those which are recursively definable from the successor relation and the unit set $\{0\}$; (2) Post's canonical sets in an alphabet $a_1 \cdots a_n$, are those recursively definable from the concatenation relation and the unit sets $\{a_1\} \cdots \{a_n\}$.

Smullyan seems not to have published the details of his proofs. But he investigated the relationship between derivability and computability in his book on the Theory of Formal Systems [Smullyan, 1961]. These formal systems are variants of the canonical systems of Post, with strong similarities to Horn clause programs.

4.5 Logic and databases

The question-answering systems of the 1960s and 1970s represented information in logical form, and used theorem-provers to answer questions represented in logical form. It was the application of SL-resolution to such deductive question-answering that led to Colmerauer's work on Prolog. In the meanwhile, Ted Codd [1970] published his relational model, which represented data as relations in logical form, but used the "non-deductive" algebraic operations of selection, projection, Cartesian product, set union and set difference, to specify database queries. However, he also showed [Codd, 1972] that the relational algebra is equivalent to a more declarative relational calculus, in which relations are defined in first-order logic.

I first learned about relational databases in 1974 at a course on the foundations of computer science at the Mathematics Centre in Amsterdam. I was giving a short course of lectures on logic for problem solving, using a set of notes, which I later expanded into my 1979 book [Kowalski, 1979b]. Erich Neuhold was giving a course about formal properties of databases, with a focus on the relational model. It was immediately obvious that the relational model and logic programming had much in common.

I organised a five day workshop at Imperial College London in May 1976, using the term "logic programming" to describe the topic of the workshop. A full day was devoted to presentations about logic and databases. Hervé Gallaire and Jean-Marie Nicholas presented the work they were doing in Toulouse, and Keith Clark talked about his work on negation as failure.

Jack Minker visited Gallaire and Nicholas in 1976, and together they organised the first workshop on logic and databases in Toulouse in 1977. The proceedings of

the workshop, published in 1978, included Clark's results on negation as failure, and Reiter's paper on closed world databases.

5 NEGATION AS FAILURE — PART 1

The practical value of extending Horn clause programs to normal logic programs with negative conditions was recognized from the earliest days of logic programming, as was the obvious way to reason with them — by *negation as failure* (abbreviated as NAF): to prove *not* p, show that all attempts to prove p fail. Intuitively, NAF is justified by the assumption that the program contains a complete definition of its predicates. The assumption is very useful in practice, but was neglected in formal logic. The problem was to give this proof-theoretic notion a logical semantics.

Ray Reiter [1978] investigated NAF in the context of a first-order database D, interpreting it as the *closed world assumption* (*CWA*) that the negation *not* p of a ground atom p holds in D if there is no proof of p from D. He showed that the CWA can lead to inconsistencies in the general case — for example, given the database $D = \{p \vee q\}$, it implies *not* p, and *not* q; but for Horn data bases no such inconsistencies can arise.

However, Keith Clark was the first to investigate NAF in the context of logic programs with negative conditions.

5.1 The Clark completion

Clark's solution was to interpret logic programs as short hand for definitions in if-and-only-if form, as illustrated for the propositional program in figure 2.

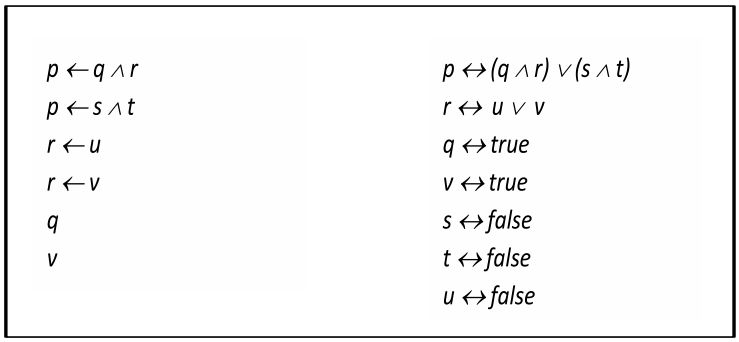

Figure 2. The logic program of figure 1, and its completion.

In the non-ground case, the logic program needs to be augmented with an equality theory, which mimics the unification algorithm, and which essentially

specifies that ground terms are equal if and only if they are syntactically identical. An example with a fragment of the necessary equality theory, is given in figure 3. Together with the equality theory, the if-and-only-if form of a logic program P is called the *completion* of P, written $comp(P)$. It is also sometimes called the *predicate completion* or the *Clark completion.*

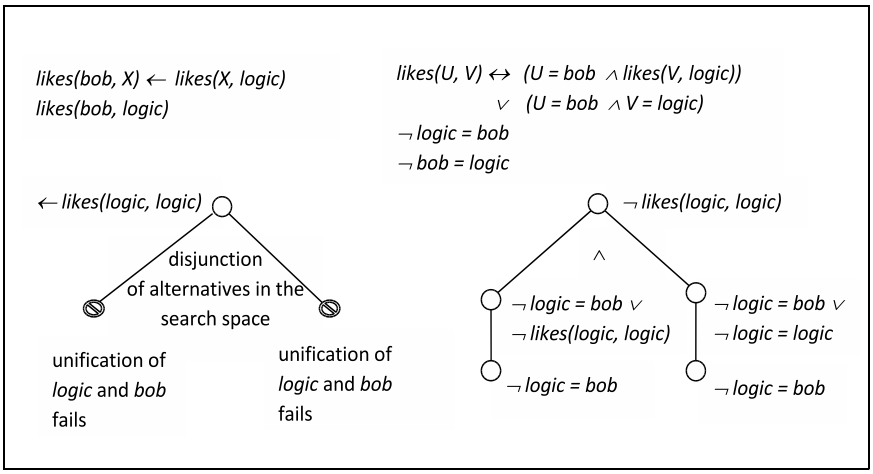

Figure 3. A proof of *not* $likes(logic, logic)$ using negation as failure and backward reasoning compared with an upside down proof of $\neg likes(logic, logic)$ using classical logic. Notice that the use of classical negation turns the disjunction of alternatives into a logical conjunction.

As figure 3 illustrates, negation as failure correctly simulates reasoning with the completion in classical logic.

Although NAF is sound with respect to the completion semantics, it is not complete. For example, if P is the program:

$$p \leftarrow q$$
$$p \leftarrow \neg q$$
$$q \leftarrow q$$

then $comp(P)$ implies p. But given the goal $\leftarrow p$, NAF goes into an infinite loop trying, but failing to show q. The completion semantics does not recognise such infinite failure, because proofs in classical logic are finite. For this reason, the completion semantics is also called the semantics of *negation as finite failure.*

In contrast with the completion semantics, the CWA formalises *negation as potentially infinite failure*, inferring $\neg q$ from the Horn clause database $q \leftarrow q$. Similarly, the minimal model semantics of Horn clauses concludes that $\neg q$ is true in the minimal model of the program $q \leftarrow q$.

Clark did not investigate the relationship between the completion semantics and the various alternative semantics of Horn clauses. Probably the first such

investigation was by Apt and van Emden [1982], who showed, among other things, that if P is a Horn clause program then:

$$I \text{ is a Herbrand model of } comp(P) \text{ iff } T_P(I) = I.$$

Compare this with the property that I is a Herbrand model of P iff $T_P(I) \subseteq I$.

5.2 The analogy with arithmetic

Clark's 1978 paper was not the first to propose the completion semantics. [Clark and Tärnlund, 1977] proposed using the completion together with induction schemas on the structure of terms to prove program properties, by analogy with the use of induction in first-order Peano arithmetic.

Consider the Horn clause definition of $append(X, Y, Z)$, which holds when the list Z is the concatenation of the list X followed by the list Y:

$$append(nil, X, X)$$
$$append(cons(U, X), Y, cons(U, Z)) \leftarrow \ append(X, Y, Z)$$

This is analogous to the definition of $plus(X, Y, Z)$, which holds when $X + Y = Z$:

$$plus(0, X, X)$$
$$plus(s(X), Y, s(Z)) \leftarrow \ plus(X, Y, Z)$$

Here the successor function $s(X)$ represents $X + 1$, as in Peano arithmetic.

These definitions alone are adequate for computing their denotations. More generally, they are adequate for solving any goal clause (which is an existentially quantified conjunction of atoms). However, to prove program properties expressed in the full syntax of first-order logic, the definitions need to be augmented with their completions and induction axioms. For example, the completion and induction over the natural numbers are both needed to show that the *plus* relation defined above is functional:

$$\forall XYUV \ [plus(X, Y, U) \wedge plus(X, Y, V) \rightarrow U = V]$$

Similarly, to show that append is associative, the definition of append needs to be augmented both with the completion and induction over lists.

Because many program properties can be expressed in the logic programming sublanguage of first-order logic, it can be hard to distinguish between clauses that are needed for computation, and clauses that are emergent properties. A similar problem arises with deductive databases. As Nicolas and Gallaire [1978] observed, it can be hard to distinguish between clauses that define data, and integrity constraints that restrict data.

For real applications, these distinctions are essential. For example, without making these distinctions, a programmer can easily write a program that includes both the definition of *append* and the property that *append* is associative. The resulting logic program would be impossibly inefficient.

The analogy with arithmetic helps to clarify the relationships between the different semantics of logic programs: It suggests that the completion augmented with induction schemas is like the first-order axioms for Peano arithmetic, and the minimal model is like the standard model of arithmetic. The fact that both notions of arithmetic have a place in mathematics suggests that both kinds of "semantics" also have a place in logic programming.

Interestingly, the analogy also works in the other direction. The fact that minimal models are the denotations of logic programs shows that the standard model of arithmetic has a syntactic core, which consists of the Horn clauses that define addition and multiplication. Martin Davis [1980] makes a similar point, but his core is essentially the Horn clause definitions of addition and multiplication augmented with the Clark Equality Theory:

$$\exists x.Z(x)$$
$$\forall xy.[Z(x) \wedge Z(y) \supset x = y]$$
$$\forall x.\exists y.S(x, y)$$
$$\forall xy.[S(x, y) \supset \neg Z(y)]$$
$$\forall xy.[Z(y) \supset A(x, y, x)]$$
$$\forall xyzuv.[A(x, y, z) \wedge S(y, u) \wedge S(z, v) \supset A(x, u, v)]$$
$$\forall xy.[Z(y) \supset P(x, y, y)]$$
$$\forall xyzuv.[P(x, y, z) \wedge S(y, u) \wedge A(z, x, v) \supset P(x, u, v)]$$

Here $Z(x)$ stands for "x is zero", $S(x, y)$ for "y is the successor of x", $A(x, y, z)$ for "$x + y = z$" and $P(x, y, z)$ for "$xy = z$".

Arguably, the syntactic core of the standard model of arithmetic explains how we can understand what it means for a sentence to be true, even if we cannot prove that the sentence is true.

5.3 Database semantics

In the same workshop in which Clark presented his work, Nicolas and Gallaire [1978] considered related issues from a database perspective. They characterised the relational database approach as viewing databases as model-theoretic structures (or interpretations), and the deductive database approach as viewing databases as theories. They argued that, in relational databases, both query evaluation and integrity constraint satisfaction are understood as evaluating the truth value of a sentence in an interpretation. But in deductive databases, they are understood as determining whether the sentence is a theorem, logically implied by the database viewed as a theory. Hence the term "deductive". In retrospect, it is now clear that both kinds of databases, whether relational or "deductive", can be viewed either as an interpretation or as a theory.

A more fundamental issue at the time of the 1978 workshop was the inability of the relational calculus and relational algebra to define recursive relations, such as the transitive closure of a binary relation. Aho and Ullman [1979] proposed to remedy this by extending the relational algebra with fixed point operators. This

proposal was pursued by Chandra and Harel [1982], who classified and analysed the complexity of the resulting hierarchy of query languages. Previously, Harel [1980] had published a harsh review of the logic and databases workshop proceedings [Gallaire and Minker, 1979], criticising it for claiming that deductive databases define relations in first-order logic despite the fact that transitive closure cannot be defined in first-order logic.

During the 1980s, the deductive database community, with roots mainly in artificial intelligence, became assimilated into a new Datalog community, influenced by logic programming, but with its roots firmly in the database field. In keeping with its database perspective, Datalog excludes function symbols. So all Herbrand models are finite, and are computable bottom-up. But pure bottom-up computation, whether viewed as model generation or as theorem-proving, ignores the query until it derives it as though by accident. To make model generation relevant to the query, Datalog uses transformations such as Magic Sets [Bancilhon, et al 1985] to incorporate the query into the transformed database rules.

As a consequence of its model generation approach, Datalog ignores the completion semantics in favour of the minimal model and fixed point semantics. For example, the surveys by Ceri, Gottlob and Tanca [1989], and Ramakrishnan and Ullman [1993], and even the more general survey of the complexity and expressive power of logic programming by Dantsin, Eiter, Gottlob and Voronkov [2001] mention the completion only in passing.

Minker's [1996] retrospective on Logic and Databases acknowledges the distinctive character of Datalog, but also includes the completion semantics. In particular, the completion semantics contributed to investigations of the semantics of integrity constraints, which was an important topic in deductive databases, before the field of Datalog fully emerged.

6 NEGATION AS FAILURE — PART 2

Theoretical investigations of the completion semantics continued, and were highlighted in John Lloyd's [1985, 1987] influential Foundations of Logic Programming book, which included results from Keith Clark's [1980] unpublished PhD thesis. Especially important among the later results were the three-valued completion semantics of Fitting [1985] and Kunen [1987], which gives, for example, the truth value *undefined* to p in the program $p \leftarrow not\ p$, whose completion is inconsistent in two-valued logic. This and other work on the completion semantics are presented in Shepherdson's [1988] survey. Much of this work concerns the correctness and completeness of SLDNF resolution (SLD resolution extended with negation as finite failure), relative to the completion semantics.

6.1 Stratification

The most significant next step in the investigation of negation was the study of stratified negation in database queries by Chandra and Harel [1985] and Naqvi

[1986].

The simplest example of a stratified logic program is that of a deductive database $E \cup I$ whose predicates are partitioned into extensional predicates, defined by facts E, and intensional predicates, defined in terms of the extensional predicates by facts and rules I. Consider, for example, a network of nodes, some of whose links at any given time may be broken[14]. This can be represented by an extensional database, say:

E: $\quad link(a,b) \quad link(a,c) \quad link(b,c) \quad broken(a,c)$

Two nodes in the network are connected if there is a path of unbroken links. This can be represented intensionally by the clauses:

I: $\quad connected(X,Y) \leftarrow link(X,Y) \wedge not\ broken(X,Y)$
$\quad\quad connected(X,Y) \leftarrow connected(X,Z) \wedge connected(Z,Y)$

The conditions of the first clause in I are completely defined by E. So they can be evaluated independently of I. The use of E to evaluate these conditions results in a set of Horn clauses I', which intuitively has the same meaning as I in the context of E:

I': $\quad connected(a,b) \quad connected(b,c)$
$\quad\quad connected(X,Y) \leftarrow connected(X,Z) \wedge connected(Z,Y)$

The natural, intended model of the original deductive database $E \cup I$ is the minimal model M of the resulting set of Horn clauses $E \cup I'$:

M: $\quad link(a,b) \quad link(a,c) \quad link(b,c) \quad broken(a,c)$
$\quad\quad connected(a,b) \quad connected(b,c) \quad connected(a,c)$

This construction can be iterated if the intensional part of the database is also partitioned into layers (or strata). The further generalisation from databases to logic programs with function symbols was investigated independently by van Gelder [1989] and by Apt, Blair and Walker [1988].

Let P be a logic program, and let $Pred = Pred_0 \cup \ldots \cup Pred_n$ be a partitioning and ordering of the predicate symbols of P. If A is an atomic formula, let $stratum(A) = i$ if and only if the predicate symbol of A is in $Pred_i$. Then P is *stratified* (with respect to this stratification of the predicate symbols), if and only if for every clause $head \leftarrow body$ in P and for every condition C in $body$:

if C is an atomic condition, then $stratum(C) \leq stratum(head)$
if C is a negative condition $notA$, then $stratum(A) < stratum(head)$.

[14]This example is inspired by the following quote from Hellerstein [2010]: "Classic discussions of Datalog start with examples of transitive closure on family trees: the dreaded *anc* and *desc* relations that afflicted a generation of graduate students. My group's work with Datalog began with the observation that more interesting examples were becoming hot topics: Web infrastructure such as webcrawlers and PageRank computation were essentially transitive closure computations, and recursive queries should simplify their implementation."

The stratification $Pred = Pred_0 \cup \ldots \cup Pred_n$ of the predicate symbols of P induces a corresponding stratification of the program $P = P_0 \cup \ldots \cup P_n$ where $head \leftarrow body$ in P_i if and only if $stratum(head) = i$.

The *perfect model* of a stratified program P is constructed by starting from the minimal model M_0 of the Horn clause program P_0 and iteratively extending the perfect model M_{i-1} of $P_0 \cup ... \cup P_{i-1}$ to the perfect model M_i of $P_0 \cup ... \cup P_i$. Assuming that the perfect model M_{i-1} has already been constructed, then M_i is constructed by using M_{i-1} to evaluate the conditions in P_i that are already defined in $P_0 \cup ... \cup P_{i-1}$ obtaining a set of Horn clauses P_i', and generating M_i as the minimal model of the Horn clauses $P_i' \cup M_{i-1}$. Constructed in this way, M_n is the perfect model of P.

For example, suppose we want to extend the logic program above by a clause that says a pair of nodes is unconnected if it is not connected:

$$unconnected(X, Y) \leftarrow not\ connected(X, Y)$$

The resulting program is stratified, with its predicates partitioned into the strata $Pred_0 = \{link,\ broken\}$, $Pred_1 = \{connected\}$, $Pred_2 = \{unconnected\}$. The perfect model M_2 of the program is constructed as follows:

$$\begin{array}{lcl} M_0 & = & \{link(a,b),\ link(a,c),\ link(b,c),\ broken(a,c)\} \\ M_1 & = & M_0 \cup \{connected(a,b),\ connected(b,c),\ connected(a,c)\} \\ M_2 & = & M_1 \cup \{unconnected(a,a),\ unconnected(b,b),\ unconnected(c,c), \\ & & \quad unconnected(b,a),\ unconnected(c,b),\ unconnected(c,a)\} \end{array}$$

It is useful to express the construction of perfect models using the notion of *reduct*, which relates it to the later construction of stable models by Gelfond and Lifschitz: Given a set of ground atoms E and a logic program P, the reduct of P by E, written $reduct(P, E)$ is the set of Horn clauses obtained from P by using E to evaluate the negative literals in P and using classical logic to simplify the resulting program. This construction is equivalent to deleting all clauses containing a condition *not* B that is false in E and deleting all conditions *not* B that are true in E. Intuitively, if E defines all of the non-atomic conditions in P, then $reduct(P, E)$ has the same meaning as P in the context of E. With this notion of *reduct*, the definition of perfect model can be expressed more concisely:

> Given a stratified logic program $P = P_0 \cup \ldots \cup P_n$, the *perfect model* of P is M_n where:
>
> $M_0 = min(P_0)$
> $M_i = min(reduct(P_i, M_{i-1}) \cup M_{i-1})$

Interestingly, this definition exploits the ambiguity of sets of atomic sentences, which can be viewed both as theories and as Herbrand interpretations. In its first occurrence in $min(reduct(P_i, M_{i-1}) \cup M_{i-1})$, M_{i-1} is treated as a Herbrand interpretation. In its second occurrence, M_{i-1} is treated as part of a Horn clause program.

6.2 Local stratification

Przymusinski [1988] extended the notion of stratification from predicate symbols to ground atoms. In effect, this replaces a program P by the program $ground(P)$. In $ground(P)$, different atoms with the same predicate symbol are treated as distinct 0-ary predicates, which can be assigned to different strata. Because of function symbols, $ground(P)$ can be countably infinite. Here is possibly the simplest sensible example that illustrates this:

$Even:\quad even(0) \qquad even(s(X)) \leftarrow not\ even(X))$

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

$Even_0:\quad even(0)$
$Even_i:\quad even(s^i(0)) \leftarrow not\ even(s^{i-1}(0))$ for $i > 0$

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

$M_0 = min(Even_0) = \{even(0)\}$
$M_1 = min(reduct(Even_1, M_0) \cup M_0) = min(\{\}\ \cup\ M_0) = M_0$
$M_2 = min(reduct(Even_2, M_1) \cup M_1) = min(\{even(s(s(0)))\}\ \cup\ M_0)$
$\quad = \{even(0), even(s(s(0)))\}$
... etc.

In general, let P be a logic program, and let $H = \cup_{i<\alpha} H_i$ be a partitioning and ordering of the Herbrand base H of P, where α is a countable, possibly transfinite ordinal. If $A \in H$, let $stratum(A) = i$ if and only if $A \in H_i$. Then P is *locally stratified* (with respect to this stratification of H) if and only if for every clause $head \leftarrow\ body$ in $ground(P)$ and for every condition C in $body$:

if C is an atomic condition, then $stratum(C) \leq stratum(head)$
if C is a negative condition $notA$, then $stratum(A) < stratum(head)$.

The stratification $\cup_{i<\alpha} H_i$ of H induces a corresponding stratification of $ground(P) = \cup_{i<\alpha} P_i$ where $head \leftarrow\ body$ is in P_i if and only if $stratum(head) = i$. The *perfect model* of P is M_α where:

$M_0 =\ min(P_0)$
$M_i =\ min(reduct(P_i, M_{i-1}) \cup M_{i-1})$
$M_\beta = \cup_{i<\beta} M_i$ if β is a limit ordinal.

Unfortunately, although this construction gives the intended model for many natural programs, like *Even* above, it can fail even for minor syntactic variants of those programs. For example:

$successor(X, s(X))$
$even(0)$
$even(Y) \leftarrow successor(X, Y) \wedge not\ even(X)$

This program cannot be locally stratified, because its ground instances contain such unstratifiable clauses as $even(0) \leftarrow successor(0,0) \wedge not\ even(0)$.

Having recognised the problem, a number of authors proposed further refinements of stratification. However, it now seems to be generally agreed that these refinements are superseded by the well-founded semantics of [Van Gelder, Ross and Schlipf 1991]. In particular, [Denecker *et al.*, 2001] argues that the well-founded semantics "provides a more general and more robust formalization of the principle of iterated inductive definition that applies beyond the stratified case."

6.3 Well-founded semantics

[Denecker *et al.*, 2001] presents a simplified definition of the well-founded semantics in terms of candidate proofs. Here is a further simplification, in which candidate proofs are viewed as arguments supported by sets Δ of assumptions that are negative literals:

Given a ground normal program P, an *argument* for an atom p supported by assumptions Δ is a finite tree T labelled with literals such that:

- p is the root of T.
- Each non-leaf node q of T is the head of some clause $q \leftarrow B$ in P, and its children are the literals in the body B of the clause. If q is a fact, with an empty body B, then it has a single child labelled by *true*.
- Each leaf is either the label *true* or a negative literal contained in Δ.

Arguments can be used to construct a three-valued Herbrand model M of P, represented by the set of all ground literals that are *true* in M. In general a three-valued Herbrand interpretation $I = I^{pos} \cup I^{neg}$ is a set of ground literals, such that every atom A in I^{pos} is *true* in I, and every atom A whose negation *not* A is in I^{neg} is *false* in I. No atom A is both *true* and *false*, and an atom A that is neither *true* nor *false* is *undefined*.

The well-founded model can be generated bottom-up, starting with the empty set $\{\}$ and repeatedly applying a three-valued consequence operator Con_P, which extends a partial three-valued interpretation I of P to a more complete three-valued interpretation:

$$Con_P(I) = \{p \mid \text{there exists an argument for } p \text{ supported by } I^{neg}\} \cup \{not\ p \mid \text{every argument for } p \text{ has a leaf } not\ q \text{ with } q \in I^{pos}\}$$

The *well-founded model* of P is the smallest three-valued interpretation I such that $Con_P(I) \subseteq I$.

The well-founded model can also be queried top-down using SLG resolution [Chen and Warren, 1996], which is a variant of SLDNF with tabling. We will see later that the well-founded semantics also has an intuitive argumentation-theoretic interpretation, in which the negative literals *not* p in $Con_P(I)$ are all the assumptions defended by I.

[Denecker, 1998] argued that the well-founded semantics formalizes the informal notion of inductive definition. In particular, the survey by [Denecker *et al.*, 2001] of theories of inductive definitions in mathematical logic identifies two main approaches: inflationary inductive definitions and iterated inductive definitions. The abstract version of inflationary induction was investigated by Moschovakis [1974], and iterated induction was introduced by Kreisel [1963] and studied by Feferman [1970] and others.

Van Gelder [1993] observed that even simple concepts can be difficult to express using inflationary induction. For example, the complement of the transitive closure of a graph can be defined simply by a stratified logic program (as in the definition of *unconnected* in terms of *connected*). But it was considered a significant achievement when a solution was found using inflationary induction. [Denecker *et al.*, 2001] argued that, in contrast, the well-founded semantics builds on the same principle as iterated inductive definitions, but is "more general and more robust".

6.4 Stable model semantics

Whereas the development of the well-founded semantics was influenced by stratification in deductive databases, the development of the stable model semantics [Gelfond and Lifschitz, 1988] was influenced by the problem of formalising default reasoning.

The earliest attempts to formalise default reasoning in artificial intelligence employed non-logical, object-oriented representations, such as semantic networks and frames [Minsky, 1975]. The first workshop dealing with logic-based approaches was held at Stanford in November 1978. A special issue of the Artificial Intelligence journal on non-monotonic reasoning, based on the workshop, was published in 1980. It contained papers by John McCarthy [1980] on circumscription, Ray Reiter [1980] on default logic, and Drew McDermott and Jon Doyle [1980] on non-monotonic modal logic. It was later shown that circumscription and default logic have interesting relationships with the semantics of negation in logic programming.

Robert Moore [1985] developed a simple and elegant reconstruction of non-monotonic modal logic, which was used later by Michael Gelfond [1987] to give an autoepistemic interpretation to normal logic programs. In Gelfond's translation, a logic program of the form $p \leftarrow q \ \wedge \ not\ r$, for example, is translated into the sentence $p \leftarrow q \wedge \neg\ Lr$ of autoepistemic logic, where Lr means r is believed. Gelfond and Lifschitz [1988] further simplified this translation in the stable model semantics, interpreting a negative literal $not\ p$ as meaning that p is not believed, in effect because the assumption that p is not believed correctly leads to the conclusion that p is not believed.

In general, given a normal logic program P:

> a Herbrand interpretation M of P is a *stable model* of P iff
> $M = min(reduct(ground(P), M))$.

For example, given $P = \{q, \quad p \leftarrow q \wedge not\ r\}$, and the Herbrand interpretation

$M = \{p,\ q\}$, the condition *not* r is true in M, and therefore the set of Horn clauses $reduct(P, M) = \{q,\ \ p \leftarrow q\}$ has the same meaning as P in the context of M. This meaning is the minimal model of $reduct(P, M)$, which is identical to M. Therefore M is a stable model of P. Moreover, it is the only stable model of P.

A program can have one, many or no stable models. For example the program $\{p \leftarrow not\ p\}$ has no stable models, but $\{p \leftarrow not\ q,\ \ q \leftarrow not\ p\}$ has two stable models, $\{p\}$ and $\{q\}$.

The stable model semantics is non-monotonic, because adding clauses to a program can non-monotonically decrease the consequences of the program. For example, adding r to the program $\{q,\ \ p \leftarrow q \wedge not\ r\}$ changes the minimal model from $\{p,\ q\}$ to $\{q\}$.

The stable model semantics has been extended to programs of the form:

$$D_1 \vee \ldots \vee D_l \leftarrow A_1 \wedge \ldots \wedge A_n \wedge\ not\ B_1 \wedge \ldots \wedge\ not\ B_m$$

where $l \geq 0,\ n \geq 0$ and $m \geq 0$. Here each D_i, A_i and B_i is an atomic formula or the "explicit" negation $\neg A$ of an atomic formula A. If $l = 0$, then the conclusion is equivalent to *false*.

The use of explicit negation (also called *strong*, or even *classical* negation) enables more natural knowledge representation, including the representation of rules and exceptions. For example:

$$canfly(X) \leftarrow bird(X) \wedge not\ \neg canfly(X)$$
$$\neg canfly(X) \leftarrow penguin(X)$$

In the extended stable model semantics, explicit negations $\neg A$ are treated as syntactic sugar for "contrary" positive atoms, say A^*. For example, the literal $\neg canfly(X)$ can be renamed as a positive atom, say $abnormal(X)$. With this renaming and the addition of clauses $\leftarrow A \wedge A^*$ for every pair of contrary atoms, the stable models of a program with explicit negation are isomorphic to the stable models of the same program without explicit negation.

Arguably, the extension to disjunctive conclusions is more problematic, because it creates the need to decide between alternative representations, for example between the two representations:

$$p \leftarrow not\ q$$
$$p\ \vee not\ q$$

In the general case, different representations have different stable models.

In any case, the extension of the stable model semantics to include disjunctive conclusions clearly distances it from the stratified and well-founded semantics, which are naturally understood as the semantics of inductive definitions. The primer by Eiter *et al.*, [2009], which presents a comprehensive survey of the stable model semantics and its associated Answer Set Programming paradigm, refers to the difference between these two kinds of semantics as the *Great Logic Programming Schism*.

It might be more appropriate to call this the *Second Great Schism*, with the *First Great Schism* being between two different views of what it means for a logic program P to solve a goal G. In the theorem-proving view, solving G means showing that P (or $comp(P)$) logically implies G. But both in the stratified/well-founded semantics and in the stable model semantics, solving G means showing that G is true in some canonical model of P.

6.5 Answer set programming

In the stratified/well-founded semantics, the intended model M of a program is unique, the goal G typically contains (existentially quantified) variables, and a solution of G is a substitution σ of terms for variables such that $G\sigma$ is true in M. In the paper that introduced the stable model semantics, Gelfond and Lifschitz [1988] similarly viewed the purpose of the stable model semantics as identifying a unique (or canonical) model of a program, writing:

> "The stable model semantics is defined for a logic program Π, if Π has exactly one stable model, and it declares that model to be the canonical model of Π."

According to this view, programs that have multiple stable models do not have a semantics at all.

The answer set programming (ASP) paradigm, proposed independently by Ilkka Niemelä [1999] and by Victor Marek and Miroslaw Truszczyński [1999] turns this point of view upside down. In ASP, the program itself is the problem to be solved, a stable model of the program is a solution, and different stable models are different solutions.

One of the simplest and most typical ASP examples is the map colouring problem. 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, ignoring the constraint that no country should have two different colours, is a simple representation:

$$colour(X, red) \vee colour(X, yellow) \vee colour(X, green) \leftarrow country(X)$$
$$\leftarrow colour(X, C) \wedge colour(Y, C) \wedge adjacent(X, Y)$$

Clearly, the first clause can be rewritten as three normal clauses with negative conditions, in this case without affecting the stable models of the program. The second clause, which implicitly has the conclusion *false*, is a constraint, which excludes models that satisfy the conditions of the clause.

ASP is almost certainly the most active area of research in logic programming today. Because, for practical applications, solutions (and therefore stable models) need to be finite, it is common to restrict programs P to ones whose grounding $ground(P)$ is finite. To ensure finite groundings, ASP programs are often restricted to ones without function symbols, as in Datalog. For this reason, and because ASP

and Datalog both employ bottom-up problem solving methods, the two areas have much in common.

7 ABDUCTIVE LOGIC PROGRAMMING

ASP also overlaps with constraint logic programming (CLP) and abductive logic programming (ALP). In a recent ASP programming competition [Denecker *et al.*, 2009], competitors included one CLP system and three ALP systems. The top two systems were both ASP solvers, but an ALP-like solver achieved a respectable third place.

7.1 Abduction

Abduction was identified by the logician and philosopher Charles Sanders Peirce (1839–1914) as a form of logical reasoning, comparable to, but distinct from, deduction and induction. Whereas *deduction* derives conclusions from assumptions (e.g. $p(a)$ from $p(X) \leftarrow q(X)$ and $q(a)$), and *induction* derives general rules from facts (e.g. $p(X) \leftarrow q(X)$ from $p(a)$, $p(b)$ and $q(a)$), *abduction* derives assumptions from rules and conclusions (e.g. $q(a)$ from $p(X) \leftarrow q(X)$ and $p(a)$). As the last example shows, abduction is closely related to top-down reasoning in logic programming.

Peirce's notion of abduction has had a big influence on epistemology and the philosophy of science, and inspired numerous applications in artificial intelligence. In my case, it was one of the inspirations of the concluding chapter of my 1979 book. It also inspired the development of Theorist [Poole *et al.*, 1987] and the application of abduction to default reasoning.

In Poole [1988], abduction is used to extend 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:

> Given T, A, I and observations G, an *abductive explanation* of G is a subset Δ of A, such that
>
> $T \cup \Delta \models G$
> $T \cup \Delta \cup I$ is consistent.

The implementation of Theorist used a combination of linear resolution to generate candidate Δ, by reasoning backwards from G, and a refutation proof procedure to show that $T \cup \Delta \cup I$ is consistent, by failing to refute $T \cup \Delta \cup I$. Although such a procedure is not even semi-decidable in theory, it is often sufficient in practice.

7.2 Horn clause ALP and the relationship with stable models

[Eshghi, 1988] and [Eshghi and Kowalski, 1989] reformulated Theorist in a logic programming setting, defining an abductive framework as a triple $\langle P, I, A \rangle$, where

P is a Horn clause program, I is a set of integrity constraints, and A is a set of ground atoms (whose predicates are called *abducible*):

> Given an abductive framework $\langle P, I, A\rangle$ and a set G of goal clauses, an *abductive solution* (or *explanation*) of G is a subset Δ of A, such that
>
> $P \cup \Delta$ solves G
> $P \cup \Delta$ satisfies I.

The requirement that $P \cup \Delta$ solves G was defined as in Theorist, namely as $P \cup \Delta \models G$. But because $P \cup \Delta$ is a set of Horn clauses, it can also be defined as G being true in the minimal model of $P \cup \Delta$. For integrity constraints I in the form of denials $\leftarrow A_1 \wedge \ldots \wedge A_n$, the requirement that $P \cup \Delta$ satisfies I was also defined as in Theorist, namely as $P \cup \Delta \cup I$ is consistent.

Whereas Poole [1988] investigated a translation of default logic into Theorist, Eshghi and Kowalski [1989] investigated a translation of the stable model semantics into ALP. The translation treats the set of all ground negative literals as a set *Neg* of abducible atoms. In doing so, it treats a normal logic program$P$ with negative conditions as a Horn clause program P^*. It uses integrity constraints I to ensure that ordinary atoms a and their abducible negations *not* a are complementary. This translates P into an ALP program $\langle P^*, I, Neg\rangle$. The correspondence is given by the relationship:

> Let H be the Herbrand base of a normal logic program P.
> Let $M \subseteq H$ and $\Delta \subseteq Neg$.
> Then M is a stable model of P if and only if $P^* \cup \Delta$ satisfies I.

The integrity constraints I needed for this correspondence include both all the denials $\leftarrow a \wedge not\ a$ and all the disjunctions $a \vee not\ a$. But at the time, we did not realise that the disjunctive constraints could be represented so simply. Instead, we represented them in a meta-logical form, in the spirit of Reiter's [1988] epistemic, modal representation of integrity constraints. I will come back to this problem of representing disjunctive constraints in the next subsection, 7.3.

Although [Eshghi and Kowalski, 1989] showed how to translate stable models into ALP, many ASP programs can be represented in ALP directly without the translation. For example, the map colouring program of section 6.5 can be represented by the ALP framework $\langle P, I, A\rangle$ where:

> P contains the definitions of the predicates *country* and *adjacent*.
> A is the predicate *colour*.
> I is the denial and disjunctive clause of the ASP program.

[Satoh and Iwayama, 1991] showed that the correspondence between stable models and ALP also works in the opposite direction: Let $\langle P, I, A\rangle$ be an ALP framework with denial integrity constraints I. For every $a \in A$, let *not-a* be a distinct atom not occurring in P. Let A^* be the set of all clauses:

$a(X) \leftarrow$ *not not-a(X)*
not-a(X) $\leftarrow$ *not a(X)*

Then M is a stable model of $P \cup I \cup A^*$ if and only if $P \cup \Delta \models I$

7.3 The ALP tower of Babel

The definition of an ALP framework $\langle P, I, A\rangle$ given in section 7.2 can be generalized so that P is a normal logic program with negation. But then the requirements that $P \cup \Delta$ *solves* G and $P \cup \Delta$ *satisfies* I have even more interpretations than before. In particular, the requirement that $P \cup \Delta$ *solves* G can be interpreted either as $comp(P \cup \Delta) \models G$, or as G is true in some appropriate canonical model of $P \cup \Delta$.

The requirement that $P \cup \Delta$ *satisfies* I is even more problematic. The problem was already the subject of extensive debate in the 1980s in the context of deductive databases D. The alternatives included the interpretation that the constraints I are consistent with D [Kowalski, 1978], that they are consistent with $comp(D)$ [Sadri and Kowalski, 1988], that they are logically implied by $comp(D)$ [Reiter, 1984; 1988; Lloyd and Topor, 1985], and that they are epistemic sentences that are true in D [Reiter, 1988].

In the context of ALP, the different interpretations of *solving* a goal are multiplied by the different interpretations of *satisfying* integrity constraints. Compared with the stable model semantics and ASP, where everyone speaks with one voice, in ALP everyone argues in a different language. So is there any prospect of clearing up the confusion?

We can start by eliminating the semantic distinction between goals and integrity constraints. Arguably, the distinction is mainly a pragmatic one between goals G that are one off (or ad hoc) and goals I that are persistent and need to be maintained. As a consequence, the definition of abductive solution can be simplified:

> Given an abductive framework $\langle P, I, A\rangle$ and a set G of goal clauses, an *abductive solution* is a subset Δ of A, such that $P \cup \Delta$ *satisfies* $G \cup I$.

The hard part of the problem remains: How to understand *integrity satisfaction*? Having been involved in the early debates about the semantics of integrity constraints, and contributed to proof procedures for both integrity checking [Sadri and Kowalski, 1988] and ALP [Fung and Kowalski, 1997], I am now convinced that the requirement that $P \cup \Delta$ *satisfies* $G \cup I$ is best understood as $G \cup I$ is true in some appropriate model M of $P \cup \Delta$. This interpretation has the added attraction that, no matter how M is defined, $G \cup I$ can include arbitrary first-order sentences. In particular, I can include the disjunctive constraints $a \vee$ *not* a, needed to simulate the stable model semantics, as was shown by Kakas and Mancarella [1990].

It remains to identify the nature of the model M. In the case of Horn clause programs P, it is obvious that M should be the minimal model of $P \cup \Delta$. Similarly, in the case of locally stratified programs, M should be the perfect model of $P \cup \Delta$.

But if P is an arbitrary normal logic program, then it is not immediately obvious whether M should be a stable model or some canonical model, such as the well-founded model.

However, the correspondence between the stable model semantics and abduction shows that stable models and abduction are different ways of achieving the same functionality. Allowing M to be any stable model would be double counting. This leaves only one sensible alternative, namely restricting M to some canonical model, with the well-founded model being the strongest candidate. The resulting combination of logic programs $P \cup \Delta$ defining well-founded models M in which first-order sentences $G \cup I$ are true is closely related to the combination of first-order logic with inductive definitions developed by Marc Denecker [1998] and his colleagues [Denecker *et al.*, 2001]. The argument for understanding abduction in terms of the well-founded semantics was also made, from the viewpoint of representing default and hypothetical reasoning, by [Pereira *et al.*, 1991].

8 CONSTRAINT LOGIC PROGRAMMING

Proof procedures for ALP and Constraint Logic Programming (CLP) have much in common: Both generate a set (or conjunction) C of conditions (abducible or constraint formulas) such that C solves the initial goal G and C is satisfiable. In both cases, they do so by reasoning top-down, backwards from G, incrementally generating C and testing C for satisfiability.

Constraints were first introduced into LP by Colmerauer [1982] in Prolog II. Their introduction was motivated mainly by the inefficiency of the "occur check" need to ensure that a term, such as $f(X)$, does not unify with a subterm, such as X. Clark [1978] had shown that unification with the occur check implements the identity relation for the domain D of Herbrand interpretations, in which ground terms can be viewed as finite trees. Colmerauer [1982] showed that unification without the occur check implements the identity relation for the domain D in which terms can be viewed as possibly infinite, rational trees.

Jaffar and Lassez [1987] introduced the CLP Scheme, which generalized the domain D to an arbitrary model-theoretic structure defining the semantics of constraint predicates. The resulting programs are sets of Horn clauses whose bodies contain both user-defined predicates and constraint predicates, but whose heads contain only user-defined predicates. The most important new instance of the scheme was CLP(R) [Jaffar *et al.*, 1992], in which the constraint domain is the set of real numbers with addition, multiplication, identity and inequality.

The semantics of a CLP program [Jaffar *et al.*, 1998] is given both by a structure D and a theory T, which is a first-order axiomatization of D. The relationship between D and T is analogous to the relationship between the standard model of arithmetic and first-order Peano arithmetic.

The "algebraic" semantics is defined in terms of truth in the minimal model of $P \cup D$, where D is the set of all ground constraint atoms that are true in D.

According to the algebraic semantics, given a constraint logic program P, goal clause G and constraint formula C:

> G is satisfiable iff G is true in the minimal model of $P \cup D$.
> C solves G iff C and $C \rightarrow G$ are true in the minimal model of $P \cup D$.

According to the "logical" semantics:

> G is satisfiable iff $P \cup T \models G$.
> C solves G iff $P \cup T \models C \rightarrow G$ and C is consistent with $P \cup T$.

According to the operational semantics:

> C solves G iff C can be generated by reasoning backwards from G, and C is satisfiable, where satisfiability is determined by a constraint solver $solve(C)$, which may also simplify C.

There is an obvious parallel here, not only with proof procedures for ALP, but also with the two main alternative ways of defining the semantics of ALP.

The "algebraic" and "logical" semantics of the CLP Scheme coincide for Horn clause programs and theories T that satisfy certain natural completeness conditions. But, in the case of CLP programs with negation, the relationship between the two kinds of semantics is much more problematic, but seems to have received relatively little attention.

If all these problems and confusions about the semantics of negation are not enough, there is one more twist to the story.

9 ARGUMENTATION

The proof procedure in the [Eshghi and Kowalski, 1989] paper was intended to compute the stable model semantics, but failed to implement the disjunctive integrity constraint in its totality. Phan Minh Dung [1991] showed that it implemented instead a localized form of the disjunctive constraint, to which he and [Kakas *et al.*, 1992] gave an argumentation interpretation. Dung [1993, 1995] generalized this interpretation and developed an abstract argumentation theory, which has wide-ranging applications beyond logic programming.

In the case of logic programming, given a ground normal program P, an *argument for a claim p supported by assumptions* $\Delta \subseteq Neg$ is a finite tree T labelled with literals such that:

- p is the root of T.
- Each non-leaf node q of T is the head of some clause $q \leftarrow B$ in P, and its children are the literals in the body B of the clause. If q is a fact, with an empty body B, then it has a single child labelled by *true*.
- Each leaf is either the label *true* or an assumption in Δ.

Viewed abstractly, it is the set Δ of assumptions supporting an argument that determines whether the argument and its claim are acceptable, and this depends, in turn, on whether or not Δ is able to defend itself against attack. Given a ground normal program P:

> $\Delta_1 \subseteq Neg$ *attacks* $\Delta_2 \subseteq Neg$ iff there exist an argument for a claim p supported by Δ_1 and an assumption *not* $p \in \Delta_2$.
>
> $\Delta_1 \subseteq Neg$ *defends* $\Delta_2 \subseteq Neg$ iff
> Δ_1 attacks every $\Delta_3 \subseteq Neg$ that attacks Δ_2.

The notions of argument, assumption, attack and defence are sufficient to reconstruct not only most logic programming semantics, but also most semantics for non-monotonic reasoning [Bondarenko *et al.*, 1997]. In the case of logic programming, sets $\Delta \subseteq Neg$ of assumptions that do not attack themselves correspond to three-valued Herbrand interpretations $M = M^{pos} \cup M^{neg}$ whose true atoms M^{pos} are supported by their false atoms $M^{neg} = \Delta$. If Δ also contains all the assumptions that it defends, then the corresponding interpretation is a three-valued stable model, as defined by Przymusinski [1990] and shown by [Wu *et al.*, 2009], and therefore a partial stable model, as defined by Sacca and Zaniolo [1990, 1991].

Sets Δ that not only contain all the assumptions that they defend, but also attack all the assumptions that they do not contain, correspond to two-valued, stable models:

> Given a ground logic program P:
>
> If $\Delta \subseteq Neg$ does not attack Δ, and Δ attacks $Neg - \Delta$, then
> $M = \{p \mid \Delta$ supports an argument for $p\}$ is a stable model of P.
>
> If M is a stable model of P, then
> $\Delta = \{not\ p \in Neg \mid p \notin M\}$ does not attack Δ, and Δ attacks $Neg - \Delta$.

From an argumentation point of view, the stable model semantics is all-out warfare: For a set of assumptions Δ to correspond to a stable model, every assumption *not* p has to take a side: Either *not* p is in Δ, or Δ attacks $\{notp\}$.

Dung [1993; 1995] argued that the stable model semantics is too extreme: It is sufficient for a set of assumptions to defend itself against all attacks:

> $\Delta \subseteq Neg$ is *admissible* iff Δ defends Δ, and Δ does not attack Δ.

Dung [1991] showed that the proof procedure of [Eshghi and Kowalski, 1989] is sound with respect to the admissibility semantics.

Dung also gave an abductive interpretation of the well-founded semantics in [Dung, 1991] and an argumentation interpretation in [Dung, 1995]. Whereas in the admissibility and stable semantics an assumption can be used in its own self-defense, in the well-founded semantics an assumption has to be defended by other assumptions:

> A set Δ of assumptions is *well-founded* iff Δ is the smallest set of assumptions that contains all the assumptions that it defends.

This is similar to Przymusinski's [1990] characterization of the well-founded semantics in terms of three-valued stable models.

The well-founded set can be constructed bottom-up by starting with the empty set {} of assumptions and repeatedly adding new assumptions defended by the previously added set of assumptions, until no further assumptions can be added. This bottom-up construction is similar to van Gelder's [1993] alternating fixed point characterisation of the well-founded semantics.

10 CONCLUSIONS

This history covers some of the highlights of the development of logic programming from the late 1960s into the 21st century. It focuses on a number of issues that are still relevant today, in particular on:

- the difference between solving a goal by theorem-proving and solving it by model generation,
- the difference between solving a goal top-down and solving it bottom-up.
- the relationship between declarative and procedural representations.

Perhaps the biggest change over the years has been the move away from viewing computation as deduction to viewing it as model generation. The seeds of this change were planted with the minimal model semantics of Horn clauses in 1976, but really got going only in the 1980s, when it was applied to the semantics of negation as failure. As a consequence, except perhaps in the context of CLP, the completion semantics of negation has been overshadowed by the model-theoretic approach.

The recent revival of Datalog [Huang *et al.*, 2011] suggests that the old promise that logic programming can unify programming and databases may have new prospects. However, the query evaluation strategies of Datalog are mainly bottom-up with magic set transformations used to simulate top-down execution. Is this simulation of top-down execution really necessary? Or might some more direct combination of top-down and bottom-up execution be more useful.

Recent years have also seen a shift away from reconciling declarative and procedural representations to a more purely declarative approach. In the meanwhile, imperative languages dominate the world of practical computing. Does this mean that logic programming is destined to become a niche technology concerned exclusively with declarative representations, based on ASP and Datalog? Or will it split into separate declarative and procedural camps, with procedural representations being relegated to the domain of Prolog? Or might it still be possible to reconcile declarative and procedural representations, perhaps by combining Prolog-like top-down execution with tabling?

ACKNOWLEDGEMENTS

Many thanks to Maurice Bruynooghe, Keith Clark, Marc Denecker, Phan Minh Dung, Maarten van Emden, Michael Gelfond, Tony Kakas, Vladimir Lifschitz, Luis Pereira, Alan Robinson, John Schlipf, Jörg Siekmann, Allen Van Gelder and David Scott Warren for their helpful comments on the paper. Special thanks to Luis and Maarten for acting as official readers of the paper.

It is important to stress that, in this case more than in most, the author alone is fully responsible for any errors of fact or judgement.

BIBLIOGRAPHY

[Anderson and Belnap, 1962] A. R. Anderson and N. D. Belnap. The pure calculus of entailment.*Journal of Symbolic Logic*, 19-52, 1962.

[Andréka and Németi, 1978] H. Andréka and I. Németi. The Generalized Completeness of Horn Predicate Logic as a Programming Language. Acta Cybernetica, 4:3–10, 1978. (This is the publish version of a 1975 report entitled "General Completeness of Prolog" Department of Artificial Intelligence, University of Edinburgh).

[Apt and Bol, 1994] K. R. Apt and R. Bol. Logic programming and negation: a survey. *Journal of Logic Programming 19-20*, 9–71, 1994.

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

[Apt and van Emden, 1982] K. R. Apt and M. van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM 29,* 3, 841–862, 1982.

[Bancilhon *et al.*, 1985] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems* (pp. 1-15). ACM, 1985.

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

[Boyer and Moore, 1972] R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. *Machine Intelligence*, *7*, 101–116, 1972.

[Brewka *et al.*, 2011] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, *54*(12), 92-103, 2011.

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. Deduction revision through intelligent backtracking. In J. Campbell, ed., "Issues in Prolog Implementation". Ellis Horwood, 1984.

[Bry *et al.*, 2007] F. Bry, N. Eisinger, T. Eiter, T. Furche, G. Gottlob, C. Ley, and F. Wei Foundations of rule-based query answering. In *Proceedings of the Third international summer school conference on Reasoning Web* (pp. 1-153). Springer-Verlag, 2007.

[Bundy and Welham, 1981] A. Bundy and B. Welham. Using meta-level inference for selective application of multiple rewrite rule sets in algebraic manipulation. *Artificial Intelligence*, *16*(2), 189-211, 1981.

[Ceri *et al.*, 1990] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Surveys in Computer Science. Springer-Verlag, 1990.

[Chandra and Harel, 1982] A. K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Sciences 25*, 1 (Aug.), 99–128, 1982.

[Chandra and Harel, 1985] A. K. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming 2*, 1 (April), 1–15, 1985. A preliminary version of this paper, "Horn Clauses and the Fixpoint Query Hierarchy," appeared in the ACM Symp. on Principles of Database Systems, 1982.

[Chen and Warren, 1996] W. Chen and D. Warren. Tabled Evaluation with Delaying for General Logic Programs. JACM 43, 20–74, 1996.

[Clark, 1978] K. L. Clark/ Negation by failure. In Gallaire, H. and Minker, J. [eds], *Logic and Databases*, Plenum Press, 293-322, 1978.

[Clark, 1980] K. L. Clark. *Predicate logic as a computational formalism* (Doctoral dissertation, Queen Mary, University of London), 1980.

[Clark and Gregory, 1983] K. L. Clark and S. Gregory). *Parlog: A parallel logic programming language*. Imperial College of Science and Technology Department of Computing, 1983.

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

[Clark *et al.*, 1982] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog language features, Logic programming, ed. *Clark/Tarnlund*, 254-266, 1982.

[Clark and Tärnlund, 1978] K. L. Clark and S.-A. Tärnlund. A first-order theory of data and programs. In Proceedings of the IFIP Congress 77, 939-944, 1978.

[Codd, 1970] E. F. Codd. Relational Completeness of Data Base Sublanguages. *Database Systems*: 65–98, 1970.

[Codd, 1972] E. F. Codd. *Relational completeness of data base sublanguages* (pp. 65-98). IBM Corporation, 1972.

[Cohen, 1988] J. Cohen. A View of the Origins and Development of Prolog, Communications ACM, vol 31, pp 26-36, 1988.

[Colmerauer, 1969] A. Colmerauer. Les systèmes Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Mimeo, Montréal, 1969.

[Colmerauer, 1970] A. Colmerauer. Total Precedence Relations, Journal ACM 1970, vol 17, pp 14-30, 1970.

[Colmerauer, 1975] A. Colmerauer. Les grammaires de metamorphose, Groupe d'Intelligence Artificielle, University de Marseille-Luminy (November 1975). Appears as 'Metamorphosis Grammars" in: L. Bolc (Ed.), Natural Language Communication with Computers, (Springer, 1978).

[Colmerauer, 1982] A. Colmerauer. Prolog II: Reference manual and theoretical model. *Groupe D'intelligence Artificielle, Faculté Des Sciences De Luminy, Marseille*, 1982.

[Colmerauer *et al.*, 1973] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en français. Research report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, Luminy, 1973.

[Colmerauer and Roussel, 1996] A. Colmerauer and P. Roussel. The birth of Prolog. In *History of programming languages—II* (pp. 331-367). ACM, 1996.

[Console *et al.*, 1991] L. Console, D. Theseider Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation* 2(5) 661-690, 1991.

[Costantini, 2002] S. Costantini. Meta-reasoning: A Survey. In Kakas, A.C., Sadri, F. (Eds.): *Computational Logic: Logic Programming and Beyond*. Springer Verlag. Vol. 2. 253-288, 2002.

[Chen and Warren, 1996] W. Chen and D. Warren. Tabled evaluation with delaying for general logic programs. *JACM 43,* 1 (January), 20–74, 1996.

[Dantsin *et al.*, 1997] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity, June 24–27, 1997, Ulm, Germany, pp. 82–101. IEEE Computer Society Press, 1997.

[Davis, 1980] M. Davis. The mathematics of non-monotonic reasoning. *Artificial Intelligence*, *13*(1), 73-80, 1980.

[Dean and Ghemawat, 2008] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113, 2008.

[Denecker, 1998] M. Denecker. The well-founded semantics is the principle of inductive definition. In *Logics in Artificial Intelligence*, J. Dix, L. Farinas del Cerro, and U. Furbach, Eds. Lecture Notes in Artificial Intelligence, vol. 1489. Springer-Verlag, 1–16, 1998.

[Denecker *et al.*, 2001] M. Denecker, M. Bruynooghe, and V. Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Transactions on Computational Logic*, *2*(4), 623-654, 2001.

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

[Denecker *et al.*, 2009] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski. The second Answer Set Programming competition. In LPNMR, E. Erdem, F. Lin, and T. Schaub, Eds. LNCS, vol. 5753. Springer, 637-654, 2009.

[Dung, 1991] P. M. Dung. Negation as hypothesis: an abductive foundation for logic programming. Proc. 8th International Conference on Logic Programming. MIT Press, 1991.

[Dung, 1993] P. M. Dung. On the acceptability of arguments and its fundamental roles in nonmonotonic reasoning and logic programming, Proceedings of IJCAI 1993, pp. 852-857, Morgan Kaufmann, 1993.

[Dung, 1995] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial intelligence*, *77*(2), 321-357, 1995.

[Eiter *et al.*, 2009] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems* (pp. 40-110). Springer Berlin Heidelberg, 2009.

[Elcock, 1990] E. W. Elcock. Absys: The First Logic Programming Language—a Retrospective and a Commentary. Journal of Logic Programming, 9(1), 1-17, 1990.

[van Emden, 2006] M. van Emden. The Early Days of Logic Programming: A Personal Perspective. *The Association of Logic Programming Newsletter,* Vol. 19 n. 3, August 2006. `http://www.cs.kuleuven.ac.be/\textasciitildedtai/projects/ALP/newsletter/aug06/`

[van Emden and Kowalski, 1976] M. van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language *JACM* , Vol. 23, No. 4, 733-742. 1976. Earlier version DCL Memo. School of Artificial Intelligence, University of Edinburgh (1974)

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

[Feferman, 1970] S. Feferman. Formal theories for transfinite iterations of generalised inductive definitions and some subsystems of analysis. In *Intuitionism and Proof theory*, A. Kino, J. Myhill, and R. Vesley, Eds. North Holland, 303–326, 1970.

[Fitting, 1985] M. Fitting. A Kripke-Kleene semantics for logic programs*. *The Journal of Logic Programming*, *2*(4), 295-312, 1985.

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

[Gallaire and Minker, 1978] H. Gallaire and J. Minker. *Logic and Data Bases.* Plenum Press, New York, 1978.

[Gallaire and Lasserre, 1982] H. Gallaire and C. Lasserre. Metalevel control for logic programs.*Logic Programming*, 173-185, 1982.

[Gelernter, 1963] H. Gelernter. Machine generated problem solving graphs. In *Proc. Symp, Math. Theory of Automata* (pp. 179-203), 1963.

[Gelfond, 1987] M. Gelfond. On Stratified Autoepistemic Theories. In *Proc. of AAAI87.* Morgan Kaufman, 207–211, 1987.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. *The stable model semantics for logic programming*. In: Proceedings of the Fifth International Conference on Logic Programming (ICLP), 1070-1080, 1988.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 3–4, 365–386, 1991.

[Green, 1969] C. Green. Application of theorem proving to problem solving. Proceedings of the 1st International Joint Conference on Artificial Intelligence. Morgan Kaufmann. 219-239, 1996.

[Harel, 1980] D. Harel. Review on Logic and Data Bases, Computing Reviews #36,671 (Aug 1980), 367-369.

[Hayes, 1973] P. J. Hayes. Computation as Deduction, Proceedings 2nd MFCS Symposium, Czechoslovakia Academy of Sciences, Prague, Czechoslovakia, 1973.

[Hayes and Kowalski, 1971] P. J. Hayes and R. A. Kowalski. Lecture Notes on Automatic Theorem-Proving. DCL Memo 40. School of Artificial Intelligence, University of Edinburgh, 1971.

[Hellerstein, 2010] J. M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic, SIGMOD Record 39(1), 2010.

[Hewitt, 1971] C. Hewitt. Procedural Embedding of Knowledge In Planner. Proceedings of the 2^{nd} International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1971.

[Hewitt, 2009] C. Hewitt. Middle History of Logic Programming: Resolution, Planner, Edinburgh LCF, Prolog, Simula, and the Japanese Fifth Generation Project, 2009. arXiv preprint arXiv:0904.3036

[Hill, 1974] R. Hill. LUSH Resolution and its Completeness. DCL Memo 78. School of Artificial Intelligence, University of Edinburgh, 1974.

[Huang *et al.*, 2011] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and Emerging Applications: an Interactive Tutorial. Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data pp. 1213-1216, 2011.

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

[Jaffar *et al.*, 1998] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programs1. *The Journal of Logic Programming*, *37*(1-3), 1-46, 1998.

[Jaffar *et al.*, 1992] J. Jaffar, S. Michaylov, P. Stuckey, and R. H. Yap The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *14*(3), 339-395, 1992.

[Kakas and Mancarella, 1990] A. C. Kakas and P. Mancarella. Generalized Stable Models: A Semantics for Abduction. In *ECAI* (Vol. 90, pp. 385-391), 1990.

[Kakas *et al.*, 1992] A. C. Kakas, R. Kowalski, and F. Toni. Abductive logic programming.*Journal of Logic and Computation*, *2*(6), 719-770, 1992.

[Kakas *et al.*, 1998] A. C. Kakas, R. Kowalski, and F. Toni. The Role of Logic Programming in Abduction, Handbook of Logic in Artificial Intelligence and Programming 5, Oxford University Press, 235-324, 1998.

[Kowalski, 1970] R. Kowalski. Search strategies for theorem proving. *Machine Intelligence*, *5*, 181-201, 1970.

[Kowalski, 1972] R. Kowalski. The Predicate Calculus as a Programming Language (abstract). Procedings of the First MFCS Symposium, Jablonna, Poland, 1972.

[Kowalski, 1974] R. Kowalski. Predicate logic as a programming language. In Proceedings of IFIP 1974 [Stockholm, Sweden). North-Holland, Amsterdam, 1974, 569-574. This is the published version of DCL Memo 70, School of Artificial Intelligence, Univ. of Edinburgh, U.K., Nov. 1973.

[Kowalski, 1978] R. Kowalski. Logic for data description, in: H. Gallaire and J. Minker (eds.), *Logic and Data Bases,* Plenum Press, New York, pp. 77-103, 1978.

[Kowalski, 1979a] R. Kowalski. Algorithm = Logic+ Control. CACM, 22(7), 424-436, 1979.

[Kowalski, 1979b] R. Kowalski. *Logic for Problem Solving.* North Holland Elsevier, 1979. This is an expanded version of DCL Memo 75, Department of Artificial Intelligence, U. of Edinburgh (1974). Also `http://www.doc.ic.ac.uk/\textasciitilderak/`.

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

[Kowalski, 2013] R. Kowalski. Logic Programming in the 1970s. In: P. Cabalar and T.C. Son (eds.) LPNMR 2013. Springer Verlag, 2013.

[Kowalski and Kuehner, 1971] R. Kowalski and D. Kuehner. Linear Resolution with selection function, *Artificial Intelligence*, vol 2, 1971, 227-260, 1971.

[Kreisel, 1963] G. Kreisel. Generalized inductive definitions. Tech. rep., Stanford University, 1963.

[Kuehner, 1969] D. Kuehner. Bi-directional search with Horn clauses. Edinburgh University, 1969.

[Kunen, 1987] K. Kunen. Negation in logic programming. *The Journal of Logic Programming*, *4*(4), 289-308, 1987.

[Lifschitz, 1988] V. Lifschitz. On the Declarative Semantics of Logic Programs with Negation, in: J. Minker (ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1988, pp. 177-192.

[Lloyd, 1985/1987] J. W. Lloyd. *Foundations of Logic Programming.* New York: Springer Verlag (1985 and 1987)

[Lloyd and Torpor, 1985] J. W. Lloyd and R. W. Topor. A Basis for Deductive Database Systems. J. Logic Programming 2: 93–109, 1985.

[Loveland, 1968] D. W. Loveland. Mechanical theorem-proving by model elimination. Journal of the ACM, 15, 236-251, 1968.

[Loveland, 1970] D. W. Loveland. A Linear Format for Resolution. In Symposium on Automatic Demonstra-tion, pp. 147-162. Springer, Berlin Heidelberg, 1970.

[Loveland, 1972] D. W. Loveland. A Unifying View of Some Linear Herbrand Procedures. JACM, 19(2), 366-384, 1972.

[Luckham, 1970] D. Luckham. Refinement Theorems in Resolution Theory. In Symposium on Automatic Demonstration (pp. 163-190). Springer, Berlin Heidelberg, 1970.

[Mariën *et al.*, 2004] M. Mariën, D. Gilis, & M. Denecker. On the relation between ID-logic and answer set programming. In *Logics in Artificial Intelligence* (pp. 108-120). Springer Berlin Heidelberg, 2004.

[Marek and Truszczyński, 1999] V. W. Marek and M. Truszczyński Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm* (pp. 375-398). Springer Berlin Heidelberg, 1999.

[McCarthy and Hayes, 1969] J. McCarthy and P. Hayes. *Some philosophical problems from the standpoint of artificial intelligence.* In Meltzer, B. and Michie, D. and Swann, M. (eds.) Machine intelligence 4, Edinburgh University Press (pp. 463-502), 1969.

[McCarthy, 1980] J. McCarthy. Circumscription — a form of non-monotonic reasoning. *Artificial intelligence*, *13*(1), 27-39, 1980.

[McDermott and Doyle, 1980] D. McDermott and J. Doyle. Non-monotonic logic I. *Artificial intelligence*,*13*(1), 41-72, 1980.

[Minker, 1996] J. Minker. *Logic and databases: A 20 year retrospective* (pp. 1-57). Springer Berlin Heidelberg, 1996.

[Minsky, 1975] M. Minsky. A framework for representing knowledge. In Winston, P. H. (ed.) The psychology of Computer Vistino. McGraw-Hill, New York, 211-277, 1975.

[Moore, 1985] R. C. Moore. Semantical considerations on nonmonotonic logic. *Artificial intelligence*, *25*(1), 75-94, 1985.

[Naqvi, 1988] S. A. Naqvi. A Logic for Negation in Database Systems, in: J. Minker (ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, Los Altos, CA, 1988, pp. 378-387.

[Nicolas and Gallaire, 1978] J. M. Nicolas and H. Gallaire. Database: Theory vs. Interpretation. In: Gallaire, H., Minker, J. (eds.), Logic and Databases, Plenum, New York, 1978.

[Niemelä, 1999] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, *25*(3-4), 241-273, 1999.

[Nilsson, 1968] N. J. Nilsson. Searching problem-solving and game-playing trees for minimal cost solutions. In *IFIP Congress (2)* (pp. 1556-1562, 1968).

[Peirce, 1931] C. S. Peirce. *Collected Papers.* C. Hartshorn & P. Weiss (eds.) Cambridge, MA: Harvard University Press, 1931.

[Pereira, 1984] L. M. Pereira. Logic Control with Logic, in: Implementations of Prolog, pp. 177-193, J. Campbell (ed.), Ellis Horwood, 1984.

[Pereira *et al.*, 1991] L. M. Pereira, J. N. Aparício, and J. J. Alferes. Hypothetical Reasoning with Well Founded Semantics. In *SCAI* (pp. 289-300), 1991.

[Pereira and Warren, 1980] F. C. Pereira and D. H. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, *13*(3), 231-278, 1980.

[Poole, 1988] D. Poole. A logical framework for default reasoning. *Artificial intelligence*, *36*(1), 27-47, 1988.

[Poole *et al.*, 1987] D. Poole, R. Goebel, and R. Aleliunas. 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, 1987.

[Przymusinski, 1988] T. C. Przymusinski. 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, 1988, pp. 193-216.

[Przymusinski, 1990] T. C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. Fundamenta Informaticae, 13(4):445–463, 1990.

[Ramakrishnan and Ullman, 1995] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The journal of logic programming*, *23*(2), 125-149, 1995.

[Reiter, 1971] R. Reiter. Two Results on Ordering for Resolution with Merging and Linear Format. JACM 18(4), 630-646, 1971.

[Reiter, 1978] R. Reiter. On Closed World Data Bases. In: Gallaire H. and Minker J. (eds.), Logic and Data Bases, Plenum Press, New York, pp. 55-76, 1978.
[Reiter, 1980] R. Reiter. A logic for default reasoning. *Artificial intelligence*, *13*(1), 81-132, 1980.
[Reiter, 1988] R. Reiter. On Integrity Constraints. In: 2nd Conference on Theoretical Aspects of Reasoning about Knowledge, 97—111, 1988.
[Robinson, 1965a] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. JACM, 12(1) 23–41, 1965.
[Robinson, 1965b] J. A. Robinson. Automatic deduction with hyper-resolution, *International J. Computer Math.* 1, 3. 227-234, 1965.
[Sacca and Zaniolo, 1990] D. Sacca and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, ACM, 205-217, 1990.
[Sacca and Zaniolo, 1991] D. Sacca and C. Zaniolo. Partial Models and Three-Valued Models in Logic Programs with Negation. In LPNMR, 87-101, 1991.
[Sadri and Kowalski, 1988] F. Sadri and R. Kowalski. A Theorem-Proving Approach to Database Integrity. In: Minker, J. [ed.], Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 313-362, 1988.
[Sagonas *et al.*, 1994] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. Pro-ceedings of the ACM SIGMOD International Conference on the Management of Data pp. 442-453, 1994.
[Satoh and Iwayama, 1992] K. Satoh and N. Iwayama. Computing abduction using the TMS. Proceedings of ICLP. MIT Press 505-518, 1992.
[Scott, 1970] D. Scott. Outline of a mathematical theory of computation. Proc. of the Fourth Annual Princeton Conference on Information 5ciences and Systems, pp. 169-176, 1970.
[Shepherdson, 1988] J. Shepherdson. Negation in Logic Programming. In: Minker, J. [ed.], Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 19-88, 1988.
[Smullyan, 1956] R. M. Smullyan. On definability by recursion (Abstract 782t). *Bulletin AMS 62*, 601, 1956.
[Smullyan, 1961] R. M. Smullyan. *Theory of Formal Systems*. Annals of Mathematical Studies Vol 47. Princeton University Press, Princeton, New Jersey, 1961.
[Tamaki and Sato, 1986] H. Tamaki and T. Sato. OLD Resolution with Tabulation. Third International Conference on Logic Programming pp. 84-98. Springer, Berlin, Heidelberg, 1986.
[Tekle and Liu, 2011] K. T. Tekle and Y. A. Liu. More Efficient Datalog Queries: Subsumptive Tabling beats Magic Sets. In Proceedings of SIGMOD International Conference on Management of Data 661-672, 2011.
[Thagard, 2005] P. Thagard. *Mind: Introduction to Cognitive Science.* Second edition. MIT Press, 2005.
[Ueda, 1986] K. Ueda. 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, 1986. Revised version in Programming of Future Generation Computers, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, pp.441-456, 1988.
[Van Gelder, 1989] A. Van Gelder. Negation as failure using tight derivations for general logic programs, The Journal of Logic Programming, Volume 6, Issues 1–2, Pages 109-133, 1989.
[Van Gelder, 1993] A. Van Gelder. The alternating fixpoint of logic programs with negation, Journal of Computer and System Sciences, Volume 47, Issue 1, Pages 185-221, 1993.
[Van Gelder *et al.*, 1991] A. Van Gelder, K. A. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. JACM *38,* 3, 620–650, 1991.
[Warren, 1978] D. H. Warren. Applied logic: its use and implementation as a programming tool. Ph.D. thesis. University of Edinburgh. Also Technical Note 290, AI Center, SRI International, 1978.
[Warren, 1983] D. H. Warren). *An abstract Prolog instruction set* (Vol. 309). Menlo Park, California: SRI International, 1983.
[Warren *et al.*, 1977] D. H. Warren, L. M. Pereira, and F. Pereira. Prolog − the language and its implementation compared with Lisp. In *ACM SIGPLAN Notices* (Vol. 12, No. 8, pp. 109-115). ACM. page 48. Warren *et al.*,. 1977.
[Winograd, 1971] T. Winograd. Procedures as a Representation for Data in a Computer Program for Under-standing Natural Language. MIT AI TR-235 (1971) Also: Understanding Natural Language. Academic Press, New York, 1972.

[Wos *et al.*, 1965] L. Wos, G. A . Robinson, and D. F. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM (JACM)*, *12*(4), 536-541, 1965.

[Wu *et al.*, 2009] Y. Wu, M. Caminada, and D. M. Gabbay. Complete extensions in argumentation coincide with 3-valued stable models in logic programming.*Studia logica*, *93*(2-3), 383-403, 2009.

[Zamov and Sharonov, 1969] N. K. Zamov and V. I. Sharonov. On a Class of Strategies for the Resolution Method. Zapiski Nauchnykh Seminarov POMI, 16, 54-64, 1969.

View publication stats