Foundations of Rule-Based Query Answering

François Bry[1], Norbert Eisinger[1], Thomas Eiter[2],
Tim Furche[1], Georg Gottlob[2,3], Clemens Ley[1],
Benedikt Linse[1], Reinhard Pichler[2], and Fang Wei[2]

[1] Institute for Informatics, University of Munich,
Oettingenstraße 67, D-80538 München, Germany
http://www.pms.ifi.lmu.de/
[2] Institute of Information Systems, Vienna University of Technology,
Favoritenstraße 11/184-3, A-1040 Vienna, Austria
http://www.kr.tuwien.ac.at/ http://www.dbai.tuwien.ac.at/
[3] Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford, OX1 3QD, England
http://web.comlab.ox.ac.uk/oucl/people/georg.gottlob.html

Abstract. This survey article introduces into the essential concepts and methods underlying rule-based query languages. It covers four complementary areas: declarative semantics based on adaptations of mathematical logic, operational semantics, complexity and expressive power, and optimisation of query evaluation.
The treatment of these areas is foundation-oriented, the foundations having resulted from over four decades of research in the logic programming and database communities on combinations of query languages and rules. These results have later formed the basis for conceiving, improving, and implementing several Web and Semantic Web technologies, in particular query languages such as XQuery or SPARQL for querying relational, XML, and RDF data, and rule languages like the "Rule Interchange Framework (RIF)" currently being developed in a working group of the W3C.
Coverage of the article is deliberately limited to declarative languages in a classical setting: issues such as query answering in F-Logic or in description logics, or the relationship of query answering to reactive rules and events, are not addressed.

1 Introduction

The foundations of query languages mostly stem from logic and complexity theory. The research on query languages has enriched these two fields with novel issues, original approaches, and a respectable body of specific results. Thus, the foundations of query languages are arguably more than applications of these two fields. They can be seen as a research field in its own right with interesting results and, possibly, even more interesting perspectives. In this field, basic and applied research often are so tightly connected that distinguishing between the two would be rather arbitrary. Furthermore, this field has been very lively since the late 1970s and is currently undergoing a renaissance, the Web motivating query and rule languages with novel capabilities. This article aims at introducing into this active field of research.

Query languages have emerged with database systems, greatly contributing to their success, in the late 1970s. First approaches to query languages were inspired by mathematical logic. As time went by, query languages offering syntactical constructs and concepts that depart from classical logic were being developed, but still, query languages kept an undeniably logical flavour. The main strengths of this flavour are: compound queries constructed using connectives such as "and" and "or"; rules expressed as implications; declarative semantics of queries and query programs reminiscent of Tarski's model-theoretic truth definition; query optimisation techniques modelled on equivalences of logical formulas; and query evaluators based on methods and heuristics similar to, even though in some cases simpler than, those of theorem provers.

With the advent of the Web in the early 1990s things have changed. Query languages are undergoing a renaissance motivated by new objectives: Web query languages have to access structured data that are subject to structural irregularities – so-called "semi-structured data" – to take into account rich textual contents while retrieving data, to deliver structured answers that may require very significant reorganisations of the data retrieved, and to perform more or less sophisticated forms of automated reasoning while accessing or delivering meta-data. All these issues have been investigated since the mid 1990s and still are. Further issues of considerable relevance, which, up till now, have received limited attention, include: query processing in a distributed and decentralised environment, query languages for search engines, search as a query primitive, and semantical data alignment.

The current query language renaissance both, takes distance from the logical setting of query languages, and builds upon it. On the one hand, recent Web query languages such as XPath and XQuery seem to be much less related to logic than former relational query languages such as SQL and former object-oriented query languages such as OQL. On the other hand, expressly logic-based Web query languages such as the experimental language Xcerpt [28, 141, 30, 29] have been proposed, and Semantic Web query languages such as RQL, RDQL, and SPARQL clearly have logical roots (see [14, 73] for surveys on Web and Semantic Web query languages). Furthermore, language optimisers and evaluators of XPath and XQuery exploit techniques formerly developed, thus bringing these

languages back to the logical roots of query languages. At the beginning of this ongoing query language renaissance, a principled and summarised presentation of query language foundations surely makes sense.

1.1 What Are Query Languages? Tentative Definitions

A first definition of what query languages are considers what they are used for: they can be defined as specialised programming languages for selecting and retrieving data from "information systems". These are (possibly very large) data repositories such as file systems, databases, and (all or part of) the World Wide Web. Query languages are specialised inasmuch as they are simpler to use or offer only limited programming functionalities that aim at easing the selection and retrieval of data from information systems.

A second attempt at defining what query languages are is to consider their programming paradigms, i.e., the brand of programming languages they belong to: query languages are declarative languages, i.e., languages abstracting out how (query) programs are to be evaluated. This makes query languages both easier to use – an advantage for the human user – and easier to optimise – an advantage for the computer. The declarativity of query languages is the reason for their close relationship to logic: declarative languages are all in some way or other based on logic.

A third approach to define what query languages are considers their major representatives: SQL for relational databases, OQL for object-oriented databases, XPath and XQuery for HTML and XML data, and RQL, RDQL, and SPARQL for RDF data. Forthcoming are query languages for OWL ontologies. Viewed from this angle, what have query languages in common? First, a separation between query programs and accessed data, requiring to compile query programs without any knowledge at all or with only limited knowledge of the data the compiled query programs are to access. Second, a dedication to data models, many, if not all, of which are strongly rooted in logic.

Query languages, as a research field, can also be defined by the issues being investigated. Central issues in query languages research include:

- query paradigms (e.g., visual, relational, object-oriented, and navigational query languages),
- declarative semantics,
- complexity and expressive power,
- procedural semantics,
- implementations of query evaluators,
- query optimisation (e.g., equivalence of queries).

Further query language issues include: integrity constraints (languages, expressive power, complexity, evaluation, satisfiability, maintenance); incremental or distributed evaluation of queries; evaluation of queries against data streams; storage of large collections of queries (e.g., for publish-subscribe systems); approximate answers; query answering in presence of uncertain information; query answering in presence of inconsistent information; querying special data (e.g.,

constraints, spatial data, graphic data); algorithms and data structures for efficient data storage and retrieval.

1.2 Coverage of This Survey

This survey article on the foundations of query languages is focused on logic, complexity and expressive power, and query optimisation. The reasons for such an admittedly limited focus are manifold. First, this focus arguably provides with a corner stone for most of the past and current research on query languages. Second, this focus covers a rather large field that could hardly be enlarged in a survey and introductory article. Third, such a focus provides with a unity of concerns and methods.

1.3 Structure of This Survey

This survey article is organised as follows. Section 1 is this introduction. Section 2 introduces a few general mathematical notions that are used in later sections. Section 3 is devoted to syntax. It introduces the syntax of classical first-order predicate logic,[4] then of fragments of first-order predicate logic that characterise classes of query languages. This section shows how the syntax of the various practical query languages can be conveyed by the syntax of first-order predicate logic. Section 4 introduces into classical first-order model theory, starting with Tarski model theory, the notion of entailment, Herbrand interpretations, and similar standard notions, explaining the relevance of these notions to query languages. After this main part, the section covers Herbrand model theory and finite model theory, which have a number of interesting and rather surprising properties that are relevant to query languages. Section 5 then treats the adaptations of classical model theory to query and rule languages, covering minimal model semantics and fixpoint semantics and discussing approaches to the declarative semantics of rule sets with negation. A subsection on RDF model theory rounds out this section. Sections 6 and 7 introduce into the operational semantics of query programs, considering positive rule sets and rule sets with non-monotonic negation, respectively. These two sections present (terminating) algorithms for the (efficient) evaluation of query programs of various types. Section 8 is devoted to complexity and expressive power of query language fragments. Section 9 introduces into query optimisation, successively considering query containment, query rewriting, and query algebras.

The purpose of Sections 3 and 4 is to make the article self-contained. Therefore these sections are entirely expository. They should make it possible for readers with limited knowledge of mathematical logic – especially of classical first-order predicate logic – to understand the foundations of query languages. For those readers who already have some of that knowledge and mathematical practice, the sections should help recall notions and state terminologies and notations.

[4] Sometimes called simply "first-order logic", a short form avoided in this article.

2 Preliminaries

2.1 General Mathematical Notions

By a *function* we mean, unless otherwise stated, a *total function*.

We consider zero to be the smallest *natural number*. The set of natural numbers is $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.

Definition 1 (Enumerable). *A set S is called* enumerable*, if there is a surjection $\mathbb{N} \to S$. A set S is called* computably enumerable *(or* recursively enumerable*), if it is enumerable with a surjection that is computable by some algorithm.*

Note that any finite set is enumerable and computably enumerable. The infinite set of all syntactically correct C programs is computably enumerable and thus enumerable. Its subset consisting of all syntactically correct C programs that do not terminate for each input is enumerable, but not computably enumerable.

2.2 Logic vs. Logics

The development of logic started in antiquity and continued through mediaeval times as an activity of philosophy aimed at analysing rational reasoning. In the late 19th century parts of logic were mathematically formalised, and in the early 20th century logic turned into a tool used in a (not fully successful) attempt to overcome a foundational crisis of mathematics. The fact that logic is not restricted to analysing reasoning in mathematics became somewhat eclipsed during those decades of extensive mathematisation, but came to the fore again when computer science discovered its close ties to logic. Today, logic provides the foundations in many areas of computer science, such as knowledge representation, database theory, programming languages, and query languages.

Logic is concerned with statements, which are utterances that may be true or false. The key features of logic are the use of formal languages for representing statements (so as to avoid ambiguities inherent to natural languages) and the quest for computable reasoning about those statements. *"Logic"* is the name of the scientific discipline investigating such formal languages for statements, but any of those languages is also called *"a logic"* – logic investigates logics.

3 Syntax: From First-Order Predicate Logic to Query Language Fragments of First-Order Predicate Logic

This section introduces the syntax of *first-order predicate logic*, which is the most prominent of logics (formal languages) and occupies a central position in logic (the scientific discipline) for several reasons: it is the most widely used and most thoroughly studied logic; it is the basis for the definition of most other logics; its expressive power is adequate for many essential issues in mathematics and computer science; its reasoning is computable in a sense to be made precise in Section 4; it is the most expressive logic featuring this kind of computability [110].

Practical query and rule languages depart from first-order predicate logic in many respects, but nonetheless they have their roots in and can conveniently be described and investigated in first-order predicate logic.

Subsection 3.1 below contains the standard definitions of first-order predicate logic syntax. The second subsection 3.2 discusses fragments (or sublanguages) of first-order predicate logic that correspond to common query or rule languages. The last subsection 3.3 discusses several modifications of the standard syntax that are used in some areas of computer science.

3.1 Syntax of First-Order Predicate Logic

First-order predicate logic is not just a single formal language, because some of its symbols may depend on the intended applications. The symbols common to all languages of first-order predicate logic are called *logical symbols*, the symbols that have to be specified in order to determine a specific language are called the *signature* (or *vocabulary*) of that language.

Definition 2 (Logical symbol). *The* logical symbols *of first-order predicate logic are:*

symbol class		*symbols*	*pronounced*
punctuation symbols		,) (	
connectives	0-*ary*	$\bot$	*bottom, falsity symbol*
		$\top$	*top, truth symbol*
	1-*ary*	$\neg$	*not, negation symbol*
	2-*ary*	$\wedge$	*and, conjunction symbol*
		$\vee$	*or, disjunction symbol*
		$\Rightarrow$	*implies, implication symbol*
quantifiers		$\forall$	*for all, universal quantifier*
		$\exists$	*exists, existential quantifier*
variables		$u\ v\ w\ x\ y\ z\ \ldots$ *possibly subscripted*	

The set of variables is infinite and computably enumerable.

Definition 3 (Signature). *A* signature *or vocabulary for first-order predicate logic is a pair* $\mathcal{L} = \left(\{Fun^n_{\mathcal{L}}\}_{n\in\mathbb{N}}, \{Rel^n_{\mathcal{L}}\}_{n\in\mathbb{N}}\right)$ *of two families of computably enumerable symbol sets, called* n*-ary* function symbols *of* $\mathcal{L}$ *and* n*-ary* relation symbols *or predicate symbols of* $\mathcal{L}$*. The* 0*-ary function symbols are called* constants *of* $\mathcal{L}$*. The* 0*-ary relation symbols are called* propositional *relation symbols of* $\mathcal{L}$*.*

Note that any of the symbol sets constituting a signature may be empty. Moreover, they need not be disjoint. If they are not, the signature is called *overloaded*. Overloading is usually uncritical, moreover it can be undone by annotating each signature symbol with its symbol class (*Fun* or *Rel*) and arity whenever required.

First-order predicate logic comes in two versions: *equality* may or may not be built-in. The version with built-in equality defines a special 2-ary relation

symbol for equality, written $=$ by some authors and written $\doteq$ or differently by authors who want to avoid confusion with the same symbol at the meta level. In this article we assume first-order predicate logic without equality, unless built-in equality is explicitly mentioned.

Definition 4 ($\mathcal{L}$-term). *Let $\mathcal{L}$ be a signature. We define inductively:*

1. *Each variable x is an $\mathcal{L}$-term.*
2. *Each constant c of $\mathcal{L}$ is an $\mathcal{L}$-term.*
3. *For each $n \geq 1$, if f is an n-ary function symbol of $\mathcal{L}$ and $t_1, \ldots, t_n$ are $\mathcal{L}$-terms, then $f(t_1, \ldots, t_n)$ is an $\mathcal{L}$-term.*

Definition 5 ($\mathcal{L}$-atom). *Let $\mathcal{L}$ be a signature. For $n \in \mathbb{N}$, if p is an n-ary relation symbol of $\mathcal{L}$ and $t_1, \ldots, t_n$ are $\mathcal{L}$-terms, then $p(t_1, \ldots, t_n)$ is an* $\mathcal{L}$-atom *or atomic $\mathcal{L}$-formula. For $n = 0$ the atom may be written $p()$ or p and is called a* propositional $\mathcal{L}$-atom*.*

Definition 6 ($\mathcal{L}$-formula). *Let $\mathcal{L}$ be a signature. We define inductively:*

1. *Each $\mathcal{L}$-atom is an $\mathcal{L}$-formula.* *(atoms)*
2. *$\bot$ and $\top$ are $\mathcal{L}$-formulas.* *(0-ary connectives)*
3. *If φ is an $\mathcal{L}$-formula, then $\neg\varphi$ is an $\mathcal{L}$-formula.* *(1-ary connectives)*
4. *If φ and ψ are $\mathcal{L}$-formulas, then $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$ and $(\varphi \Rightarrow \psi)$ are $\mathcal{L}$-formulas.* *(2-ary connectives)*
5. *If x is a variable and φ is an $\mathcal{L}$-formula, then $\forall x\varphi$ and $\exists x\varphi$ are $\mathcal{L}$-formulas.* *(quantifiers)*

In most cases the signature $\mathcal{L}$ is clear from context, and we simply speak of terms, atoms, and formulas without the prefix "$\mathcal{L}$-". If no signature is specified, one usually assumes the conventions:

$p, q, r, \ldots$ are relation symbols with appropriate arities.
$f, g, h, \ldots$ are function symbols with appropriate arities $\neq 0$.
$a, b, c, \ldots$ are constants, i.e., function symbols with arity 0.

The set of terms is a formal language for representing individuals about which statements can be made. The set of formulas is the formal language for representing such statements. For example, constants a and b might represent numbers, function symbol f an arithmetic operation, and relation symbol p an arithmetic comparison relation. Then the term $f(a, f(a, b))$ would also represent a number, whereas the atomic formula $p(a, f(a, b))$ would represent a statement about two numbers.

Unique Parsing of Terms and Formulas. The definitions above, in particular the fact that parentheses enclose formulas constructed with a binary connective, ensure an unambiguous syntactical structure of any term or formula. For the sake of readability this strict syntax definition can be relaxed by the convention that $\wedge$ takes precedence over $\vee$ and both of them take precedence over $\Rightarrow$. Thus, $q(a) \vee q(b) \wedge r(b) \Rightarrow p(a, f(a, b))$ is a shorthand for the fully parenthesised form

$((q(a) \vee (q(b) \wedge r(b))) \Rightarrow p(a, f(a, b)))$. Likewise, one usually assumes that $\wedge$ and $\vee$ associate to the left and $\Rightarrow$ associates to the right. As a further means to improve readability, some of the parentheses may be written as square brackets or curly braces.

Definition 7 (Subformula). *The* subformulas *of a formula φ are φ itself and all subformulas of* immediate subformulas *of φ.*

- *Atomic formulas and $\bot$ and $\top$ have no immediate subformulas.*
- *The only immediate subformula of $\neg\psi$ is ψ.*
- *The immediate subformulas of $(\psi_1 \wedge \psi_2)$ or $(\psi_1 \vee \psi_2)$ or $(\psi_1 \Rightarrow \psi_2)$ are ψ_1 and ψ_2.*
- *The only immediate subformula of $\forall x\psi$ or $\exists x\psi$ is ψ.*

Definition 8 (Scope). *Let φ be a formula, Q a quantifier, and $Qx\psi$ a subformula of φ. Then Qx is called a* quantifier for x. *Its* scope *in φ is the subformula ψ except subformulas of ψ that begin with a quantifier for the same variable x.*

Each occurrence of x in the scope of Qx is bound *in φ by Qx. Each occurrence of x that is not in the scope of any quantifier for x is a* free *occurrence of x in φ.*

Example 9 (Bound/free variable). Let φ be $\big(\forall x[\exists x p(x) \wedge q(x)] \Rightarrow [r(x) \vee \forall x s(x)]\big)$. The x in $p(x)$ is bound in φ by $\exists x$. The x in $q(x)$ is bound in φ by the first $\forall x$. The x in $r(x)$ is free in φ. The x in $s(x)$ is bound in φ by the last $\forall x$.

Let φ' be $\forall x\big([\exists x p(x) \wedge q(x)] \Rightarrow [r(x) \vee \forall x s(x)]\big)$. Here both the x in $p(x)$ and the x in $r(x)$ are bound in φ' by the first $\forall x$.

Note that being bound or free is not a property of just a variable occurrence, but of a variable occurrence relative to a formula. For instance, x is bound in the formula $\forall x\, p(x)$, but free in its subformula $p(x)$.

Definition 10 (Rectified formula). *A formula φ is* rectified*, if for each occurrence Qx of a quantifier for a variable x, there is neither any free occurrence of x in φ nor any other occurrence of a quantifier for the same variable x.*

Any formula can be rectified by consistently renaming its quantified variables. The formula $(\forall x[\exists x p(x) \wedge q(x)] \Rightarrow [r(x) \vee \forall x s(x)])$ from the example above can be rectified to $(\forall u[\exists v p(v) \wedge q(u)] \Rightarrow [r(x) \vee \forall w s(w)])$. Note that rectification leaves any free variables free and unrenamed. Another name for rectification, mainly used in special cases with implicit quantification, is *standardisation apart*.

Definition 11 (Ground term or formula, closed formula). *A* ground term *is a term containing no variable. A* ground formula *is a formula containing no variable. A* closed formula *or sentence is a formula containing no free variable.*

For example, $p(a)$ is a ground atom and therefore closed. The formula $\forall x\, p(x)$ is not ground, but closed. The atom $p(x)$ is neither ground nor closed. In Example 9 above, the formula φ is not closed and the formula φ' is closed.

Definition 12 (Propositional formula). *A* propositional formula *is a formula containing no quantifier and no relation symbol of arity > 0.*

Propositional vs. Ground. Propositional formulas are composed of connectives and 0-ary relation symbols only. Obviously, each propositional formula is ground. The converse is not correct in the strict formal sense, but ground formulas can be regarded as propositional in a broader sense:

If $\mathcal{L}$ is a signature for first-order predicate logic, the set of ground $\mathcal{L}$-atoms is computably enumerable. Let $\mathcal{L}'$ be a new signature defining each ground $\mathcal{L}$-atom as a 0-ary relation "symbol" of $\mathcal{L}'$. Now each ground $\mathcal{L}$-formula can also be read as a propositional $\mathcal{L}'$-formula.

Note that this simple switch of viewpoints works only for ground formulas, because it cannot capture the dependencies between quantifiers and variables.

Definition 13 (Polarity). *Let φ be a formula. The* polarities *of occurrences of its subformulas are positive or negative as follows:*
- *The polarity of φ in φ is positive.*
- *If ψ is $\neg\psi_1$ or $(\psi_1 \Rightarrow \psi_2)$ and occurs in φ, the polarity of ψ_1 in φ is the opposite of the polarity of ψ in φ.*
- *In all other cases, if ψ is an occurrence in φ of a subformula with immediate subformula ψ', the polarity of ψ' in φ is the same as the polarity of ψ in φ.*

The polarity counts whether an occurrence of a subformula is within the scope of an even or odd number of negations. The left-hand immediate subformula of an implication counts as an implicitly negated subformula.

Definition 14 (Universal formula). *A formula φ is* universal*, iff each occurrence of $\forall$ has positive and each occurrence of $\exists$ has negative polarity in φ.*

For instance, $\forall x\,([\neg\forall y\, p(x,y)] \Rightarrow [\neg\exists z\, p(x,z)])$ is a universal closed formula, whereas $\forall x\,([\neg\forall y\, p(x,y)] \Rightarrow [\neg\forall z\, p(x,z)])$ is not universal.

Definition 15 (Prenex form). *A formula φ is in* prenex form*, iff it has the form $Q_1x_1 \ldots Q_nx_n\,\psi$ where $n \geq 0$ and the Q_i are quantifiers and ψ contains no quantifier. The quantifier-free subformula ψ is called the* matrix *of φ.*

Obviously, a formula in prenex form is universal iff it does not contain $\exists$. Each formula can be transformed into an equivalent formula in prenex form (equivalent in the sense of $\models\!\!\models$ from Section 4).

Notation 16 (Term list notation). *Let $\boldsymbol{u} = t_1, \ldots, t_k$ be a list of terms, let f and p be a k-ary function and relation symbol. Then $f(\boldsymbol{u})$ is a short notation for the term $f(t_1, \ldots, t_k)$ and $p(\boldsymbol{u})$ for the atom $p(t_1, \ldots, t_k)$.*

Let $\boldsymbol{x} = x_1, \ldots, x_n$ be a list of variables and φ a formula. Then $\forall\boldsymbol{x}\varphi$ is a short notation for $\forall x_1 \ldots \forall x_n\,\varphi$ and $\exists\boldsymbol{x}\varphi$ for $\exists x_1 \ldots \exists x_n\,\varphi$. In the case $n = 0$ both $\forall\boldsymbol{x}\varphi$ and $\exists\boldsymbol{x}\varphi$ stand for φ.

Definition 17 (Universal/existential closure). *Let φ be a formula. Let $\boldsymbol{x}$ be the list of all variables having a free occurrence in φ. The* universal closure $\forall^*\varphi$ *is defined as $\forall\boldsymbol{x}\varphi$ and the* existential closure $\exists^*\varphi$ *as $\exists\boldsymbol{x}\varphi$.*

Technically, a quantifier-free formula such as $((p(x,y) \wedge p(y,z)) \Rightarrow p(x,z))$ contains free variables. It is fairly common to use quantifier-free notations as shorthand for their universal closure, which is a closed universal formula in prenex form, in this case $\forall x \forall y \forall z((p(x,y) \wedge p(y,z)) \Rightarrow p(x,z))$.

3.2 Query and Rule Language Fragments of First-Order Predicate Logic

Notation 18 (Rule). *A* rule $\psi \leftarrow \varphi$ *is a notation for a not necessarily closed formula* $(\varphi \Rightarrow \psi)$*. The subformula* φ *is called the* antecedent *or body and* ψ *the* consequent *or head of the rule. A rule* $\psi \leftarrow \top$ *may be written* $\psi \leftarrow$ *with empty antecedent. A rule* $\bot \leftarrow \varphi$ *may be written* $\leftarrow \varphi$ *with empty consequent.*

Implicit Quantification. Typically, a rule is a shorthand notation for its universal closure. The set of free variables in a rule $\psi \leftarrow \varphi$ can be partitioned into the variables $\boldsymbol{x}$ that occur in ψ and the variables $\boldsymbol{y}$ that occur in φ but not in ψ. Then the universal closure $\forall \boldsymbol{x} \forall \boldsymbol{y}(\psi \leftarrow \varphi)$ is equivalent to $\forall \boldsymbol{x}(\psi \leftarrow \exists \boldsymbol{y} \varphi)$ in the sense of $\models\!\!\models$ (Section 4). Thus, the free variables occurring only in the rule antecedent can be described as implicitly *universally* quantified *in the entire rule* or implicitly *existentially* quantified *in the rule antecedent*. The two alternative descriptions mean the same, but they can be confusing, especially for rules with empty consequent.

Definition 19 (Literal, complement). *If* A *is an atom, both* A *and* $\neg A$ *are* literals*. The literal* A *is* positive*, the literal* $\neg A$ *is* negative*, and the two are a pair of* complementary *literals. The* complement *of* A*, written* $\overline{A}$*, is* $\neg A$*, the complement of* $\neg A$*, written* $\overline{\neg A}$*, is* A*.*

Definition 20 (Clause). *A* clause *is a disjunction of finitely many literals. A clause is written* $A_1 \vee \ldots \vee A_k \leftarrow L_1 \wedge \ldots \wedge L_n$ *in rule notation, which stands for the disjunction* $A_1 \vee \ldots \vee A_k \vee \overline{L_1} \vee \ldots \vee \overline{L_n}$ *with atoms* A_i *and literals* L_j*,* $k \geq 0$*,* $n \geq 0$*. A clause represents its universal closure.*

Any formula of first-order predicate logic can be transformed into a finite set of clauses with essentially the same meaning (see Section 4).

3.2.1 Logic Programming.

Logic programming considers clauses with non-empty consequent as *programs* and clauses with empty consequent as *goals* used for program invocation. The operational and declarative semantics of logic programs depend on whether the antecedent is a conjunction of atoms or of arbitrary literals and whether the consequent is just a single atom or a disjunction of several atoms.

Definition 21 (Clause classification). *Let* $k, n \in \mathbb{N}$*, let* A, A_j, B_i *be atoms and* L_i *be literals. The following names are defined for special forms of clauses.*

	Name		*Form*	
Horn clause	definite clause		$A \leftarrow B_1 \wedge \ldots \wedge B_n$	$k = 1, n \geq 0$
		unit clause[5]	$A \leftarrow$	$k = 1, n = 0$
	definite goal		$\leftarrow B_1 \wedge \ldots \wedge B_n$	$k = 0, n \geq 0$
		empty clause[6]	$\leftarrow$	$k = 0, n = 0$

normal clause	$A \leftarrow L_1 \wedge \ldots \wedge L_n$	$k = 1, n \geq 0$
normal goal	$\leftarrow L_1 \wedge \ldots \wedge L_n$	$k = 0, n \geq 0$
disjunctive clause	$A_1 \vee \ldots \vee A_k \leftarrow B_1 \wedge \ldots \wedge B_n$	$k \geq 0, n \geq 0$
general clause	$A_1 \vee \ldots \vee A_k \leftarrow L_1 \wedge \ldots \wedge L_n$	$k \geq 0, n \geq 0$

A finite set of definite clauses is called a *definite program*. Definite programs invoked by definite queries represent a fragment of first-order predicate logic with especially nice semantic properties. In the context of the programming language Prolog this fragment is sometimes called "pure Prolog". The generalisation to normal clauses and normal queries allows to use negation in antecedents, which may be handled as in Prolog by negation as failure. Programs with disjunctive clauses are investigated in the field of disjunctive logic programming.

Definite programs do not have the full expressive power of first-order predicate logic. For example, given relation symbols *person*, *male*, *female*, it is not possible to express with definite clauses that each person is male or female. This requires a disjunctive clause $male(x) \vee female(x) \leftarrow person(x)$ or a normal clause $male(x) \leftarrow person(x) \wedge \neg female(x)$. The two are equivalent in the classical sense of $\models\!\!\models$ (see Section 4), but their operational treatment might be different.

3.2.2 Datalog. Logic-based formalisations of query languages can be based on concepts of logic programming, but with a number of specifics:

- Function symbols other than constants are excluded. Thus, the only terms are variables and constants.
- Relation symbols are partitioned into those that may occur in the data to be queried, called *extensional*, and those that may not, called *intensional*.
- Clauses are assumed to be *range restricted*, which essentially requires that all variables in the consequent of a clause also occur in its antecedent.

Definition 22 (Database schema and instance, extensional, intensional). *Let* $\mathcal{L} = \big(\{Fun^n_{\mathcal{L}}\}_{n\in\mathbb{N}}, \{Rel^n_{\mathcal{L}}\}_{n\in\mathbb{N}}\big)$ *be a signature with* $Fun^n_{\mathcal{L}} = \emptyset$ *for* $n > 0$.

A database schema *over* $\mathcal{L}$ *is a nonempty, finite subset* $\mathcal{D} \subseteq \bigcup_{n\in\mathbb{N}} Rel^n_{\mathcal{L}}$. *The relation symbols in* $\mathcal{D}$ *and any atoms constructed with them are called* extensional. *The other relation symbols and atoms are called* intensional.

A database instance for $\mathcal{D}$ *is a finite set of extensional ground atoms.*

Definition 23 (Range restricted). *A general clause is* range restricted *if each variable occurring anywhere in it occurs in a positive literal of its antecedent.*

Specialised to definite clauses, range restriction means that each variable occurring in the consequent also occurs in the antecedent of the clause.

[5] Unit clauses are also called *facts*, which is meant in a purely syntactic sense although the word suggests a semantic sense.

[6] The empty clause is usually denoted $\square$ in the literature on automated deduction.

Definition 24 (Datalog). *A* datalog clause *is a range restricted definite clause*[7] *whose consequent is an intensional atom.*

A datalog program *is a finite set of datalog clauses.*

In a database instance, the set of all extensional atoms sharing the same n-ary relation symbol amounts to an extensional specification of an n-ary relation. These relations are also referred to as the *extensional database (EDB)*. In a datalog program, the set of all datalog clauses whose consequent atoms share the same n-ary relation symbol amounts to an intensional specification of an n-ary relation. These relations are also referred to as the *intensional database (IDB)*. The antecedent of a datalog clause may contain both extensional and intensional atoms, thus accessing both kinds of relations.

Note that range restriction implies that datalog unit clauses are ground. Such clauses are typically needed as base cases for recursive definitions.

The distinction between extensional and intensional is a pragmatic one. It is useful in processing datalog programs written for querying: clauses whose consequents contain only intensional atoms can be pre-processed, e.g., rewritten or compiled, without knowledge of the database instance to be queried.

On the other hand the distinction is no point of principle. Any pair of a datalog program and a database instance can be fused into an exclusively intensional form by writing each extensional atom from the database instance as a datalog unit clause and redeclaring all relation symbols as intensional. Conversely, any such "fused" datalog program can be separated into an intensional and a (new) extensional part: provided that a relation symbol r occurs in the consequents of unit clauses only, each of these unit clauses $r(c_1, \ldots, c_k) \leftarrow$ is written as an extensional atom $r'(c_1, \ldots, c_k)$ with an additional interfacing datalog clause $r(x_1, \ldots, x_k) \leftarrow r'(x_1, \ldots, x_k)$ for a new relation symbol r'. The new relation symbols are then declared to be extensional.

Many variants of datalog have been defined by modifying the plain version introduced here. Such restricted or extended versions of datalog are motivated by their interesting expressive power and/or complexity or by their correspondence to classes of queries defined by other formalisation approaches. See Section 8 for more details and [46] for a survey. Here we list just some of these versions of datalog.

- *Monadic datalog* is datalog where all intensional relation symbols are unary.
- *Nonrecursive datalog* is datalog without direct or indirect recursion.
- *Linear datalog* is datalog where each clause antecedent contains at most one intensional atom (thus restricting the form of recursion).
- *Disjunctive datalog* is datalog with disjunctive instead of definite clauses.
- *Datalog*$^\neg$ is datalog with normal instead of definite clauses.
- *Nonrecursive datalog*$^\neg$ is datalog$^\neg$ without direct or indirect recursion.
- *Disjunctive datalog*$^\neg$ is datalog$^\neg$ with general instead of normal clauses.

[7] Some authors define datalog clauses without requiring that they are range restricted.

3.2.3 Conjunctive Queries. The most trivial form of nonrecursive datalog is obtained by disallowing intensional relation symbols in rule antecedents. If in addition the datalog program is restricted to just one clause, the consequent of this clause contains the only occurrence of an intensional relation symbol. By a wide-spread convention this unique intensional relation symbol is called the answer relation symbol and the rule is called a conjunctive query.

Definition 25 (Conjunctive query). *A* conjunctive query *is a datalog rule* $ans(\boldsymbol{u}) \leftarrow r_1(\boldsymbol{u}_1) \wedge \ldots \wedge r_n(\boldsymbol{u}_n)$ *where* $n \geq 0$*, the* r_i *are extensional and ans is an intensional relation symbol,* $\boldsymbol{u}, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$ *are lists of terms of appropriate length, and the rule is range restricted, i.e., each variable in* $\boldsymbol{u}$ *also occurs in at least one of* $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$*.*

A boolean conjunctive query *is a conjunctive query where* $\boldsymbol{u}$ *is the empty list, i.e., the answer relation symbol ans is propositional.*

The following examples of conjunctive queries assume that *parent* is a 2-ary and *male* and *female* are 1-ary extensional relation symbols. The first two are examples of boolean conjunctive queries.

$ans() \leftarrow parent(mary, tom)$	*is Mary a parent of Tom?*
$ans() \leftarrow parent(mary, y)$	*does Mary have children?*
$ans(x) \leftarrow parent(x, tom)$	*who are Tom's parents?*
$ans(x) \leftarrow female(x) \wedge parent(x, y) \wedge parent(y, tom)$	*who are Tom's grandmothers?*
$ans(x, z) \leftarrow male(x) \wedge parent(x, y) \wedge parent(y, z)$	*who are grandfathers and their grandchildren?*

The class of conjunctive queries enjoys interesting complexity properties (see Section 8), but its expressive power is limited. Given only the extensional relation symbols above, the following query types cannot be expressed as conjunctive queries:

1. *who are parents of Tom or Mary?*
   requires disjunction in rule antecedents or more than a single rule.
2. *who are parents, but not of Tom?*
   requires negation in rule antecedents.
3. *who are women all of whose children are sons?*
   requires universal quantification in rule antecedents. Note that variables occurring only in the antecedent of a conjunctive query (such as y in the examples above) are interpreted as if existentially quantified in the antecedent.
4. *who are ancestors of Tom?*
   requires recursion, i.e., intensional relation symbols in rule antecedents.

Conjunctive queries have been extended to make some of these query types expressible and to allow comparisons with classes of relational algebra queries.

Basic conjunctive queries as defined above correspond to the *SPC subclass* of relational algebra queries constructed with selection, projection, cartesian product (or, alternatively, join).

Conjunctive queries extended with disjunction in rule antecedents correspond to the *SPCU subclass* of relational algebra queries, which incorporates union.

Conjunctive queries extended with negation, disjunction, and quantification in rule antecedents (but no recursion) are known as *first-order queries* and correspond to the full class of relational algebra queries.

However, for an exact correspondence first-order queries have to be restricted to *domain independent* queries. This is a semantic characterisation that excludes queries for which the answers depend on information that may not be completely available. For instance, the set of answers to $ans(x) \leftarrow \neg parent(x, x)$ comprises all of humanity – or, depending on the domain of individuals under consideration, all mammals or all vertebrates. This problem does not arise iff the query is domain independent. Unfortunately, domain independence is undecidable.[8] But there are several syntactic, i.e., decidable, criteria that are sufficient for domain independence. Range restricted general clauses (Definition 21), for example, are domain independent.

Another subclass of first-order queries restricts rule antecedents to formulas from the so-called *guarded fragment* of first-order predicate logic, where quantifiers have to be "guarded" by atoms containing the quantified variables (see the subsection on range restricted quantification on page 18). The guarded fragment corresponds to the *semijoin algebra*, the variant of the full relational algebra obtained by replacing the join operator by the semijoin operator [105]. See Section 9 for more details.

In summary, the various extensions to conjunctive queries cover all of the query types above except the last one, which requires recursion. Some of the extensions are obviously not datalog, but only from a syntactic point of view. They can be expressed with nonrecursive datalog$^\neg$ provided that more than one rule is allowed. In other words, the expressive power of datalog is strictly greater than that of relational algebra, and the add-on is due to recursion.

3.2.4 Single-Rule Programs (sirups). Datalog programs containing a single non-unit clause and possibly some unit clauses have been introduced in order to study various "degrees" of recursion. They are called *single-rule programs*, sirups for short.

Pure sirups are datalog programs consisting of a single rule and no unit-clause. In particular, conjunctive queries are pure sirups. *Single ground fact sirups* are datalog programs consisting of a single rule and at most one ground unit clause. *General sirups* are datalog programs consisting of a single rule and some unit clauses. For each of these classes its linear subclass is also of interest.

It turns out that even such strongly restricted classes of sirups have essentially the same complexity and expressive power as general datalog programs [79].

[8] Basically, because if φ is not domain independent, then $(\varphi \wedge \psi)$ is domain independent iff ψ is unsatisfiable, an undecidable property.

3.3 Syntactic Variations of First-Order Predicate Logic Relevant to Query Languages

Programming and modelling languages from various areas of computer science often provide constructs that resemble terms or formulas of first-order predicate logic. Such constructs sometimes deviate from their logical counterparts in certain aspects, which in most cases are more a matter of convenience than of fundamental principles. By considering such constructs as syntactic variants – typically, more convenient ones – of logical terms or formulas, logic's semantic apparatus becomes applicable and can be used to define the meaning of such constructs.

3.3.1 Variations from Object-Oriented and Knowledge Representation Formalisms.

Record-like Structures. Language constructs resembling records or structures of imperative programming languages are meant for collecting all data about some real object in a single syntactic unit:

```
Person
   firstName: Mary
   lastName:  Miller
   bornIn:    1984
```

Regardless of its concrete syntax, such a construct can be seen as syntactic sugar for the ground atom $person(Mary, Miller, 1984)$ or for the ground formula $person(c) \wedge firstName(c) \doteq Mary \wedge lastName(c) \doteq Miller \wedge bornIn(c) \doteq 1984$. The former translation is sufficient for simple structures, the latter, assuming built-in equality, can represent more complex ones.

Cyclic Structures. Some object-oriented programming or modelling languages allow self-references such as:

```
Employee
   firstName: Mary
   lastName:  Miller
   bornIn:    1984
   superior:  SELF
```

Here the keyword `SELF` refers to the very syntactic unit in which it occurs. First-order predicate logic does not support "cyclic terms" or "cyclic formulas", thus there is no direct translation to logic.

Constructs like `SELF` represent *implicit references*, in contrast to *explicit references* such as unique identifiers or keys. Implicit references have the purpose to restrict the user's possibilities for manipulating references, but they are implemented using explicit references. Roughly speaking, implicit references are explicit references with information hiding. Translating explicit references to logic is straightforward, and it is just the information hiding aspect that has no counterpart in logic. Logic – like the relational and other data models – can represent information, but is not concerned with information hiding.

Object Identity. Another feature of object-oriented programming or modelling languages is a so-called "object identity", which allows to distinguish between objects that are syntactically equal, such as two objects representing two people whose personal data happens to coincide:

```
Person                    Person
   firstName: Mary           firstName: Mary
   lastName:  Miller         lastName:  Miller
   bornIn:    1984           bornIn:    1984
```

Logic is referentially transparent, meaning that it does not distinguish between syntactically equal "copies". Thus, there is no direct translation to logic of objects with object identity.

As in the case of cyclic structures, object identity is a concept for implicit reference, which is based on explicit references but hides some information. Again, it is only the information hiding aspect that cannot be translated to logic.

Positions vs. Roles. The position of arguments in a term or atom such as $person(Mary, Miller, 1984)$ is significant. Exchanging the first two arguments would result in the representation of a different person.

Some languages allow to associate an identifier, a so-called *role*, with each argument position. An argument is then written as a pair $role \rightarrowtriangle term$, such as $person(firstName \rightarrowtriangle Mary,\ \ lastName \rightarrowtriangle Miller,\ \ bornIn \rightarrowtriangle 1984)$, alternatively $person(lastName \rightarrowtriangle Miller,\ firstName \rightarrowtriangle Mary,\ bornIn \rightarrowtriangle 1984)$, which are considered to be the same. The record-like constructs above correspond most naturally to such a notation using roles.

A syntax with roles has several advantages. It admits arbitrary orderings of arguments and is therefore more flexible. It improves readability, especially with high numbers of arguments. Moreover, it can handle "don't care" arguments more conveniently by simply omitting them rather than representing them by wildcard variables. Such arguments are frequent in queries,

Nevertheless, it does not change the expressive power. The standard syntax $person(Mary, Miller, 1984)$ can be seen as an abbreviated form of a role-based notation $person(1 \rightarrowtriangle Mary,\ \ 2 \rightarrowtriangle Miller,\ \ 3 \rightarrowtriangle 1984)$ using positions as roles. Conversely, the role identifiers can simply be numbered consecutively, thus transforming the role-based notation into the notation using positions as roles, of which the standard syntax is just an abbreviated form.

3.3.2 Variations from Relational Databases.

Atoms vs. Tuples. In logic, an atom $person(Mary, Miller, 1984)$ formalises a statement about two names and a number. Logic is concerned with statements that may be true or false. Relational databases, on the other hand, are concerned with relations or tables, which are sets of tuples. Under this perspective the point of the example above is that the tuple of the two names and the number belongs to the relation, formalised $(Mary, Miller, 1984) \in person$. Obviously, the two formalisations are directly interchangeable.

The different concerns of the two fields also result in different notions of variables. In logic, variables are used as placeholders for the things about which statements are made, as in $person(Mary, Miller, x)$. In relational databases, variables are used as placeholders for tuples, as in $x \in person$, with notations like x_3 or $x.3$ for accessing a single coordinate of a tuple x, here the third one.

Positions vs. Roles, continued. The alternative to base the notation on positions or on roles also exists for the tuple notation. A standard tuple is a member of a cartesian product of sets and therefore ordered. Another possibility is to regard a tuple as an unordered collection of pairs $role \rightarrowtriangle value$. In the latter case the notation for accessing a single coordinate of a tuple x is x_{bornIn} or $x.bornIn$ using the role instead of the position.

Combined, those alternatives result in four notational variants, which are interchangeable:

Atom	
positions	$person(Mary, Miller, 1984)$
roles	$person(firstName \rightarrowtriangle Mary,\ lastName \rightarrowtriangle Miller,\ bornIn \rightarrowtriangle 1984)$
Tuple	
positions	$(Mary, Miller, 1984) \in person$
roles	$(firstName \rightarrowtriangle Mary,\ lastName \rightarrowtriangle Miller,\ bornIn \rightarrowtriangle 1984) \in person$

Relational Calculus. Relational calculus was the first logic-based formalisation of query languages. Early versions of the relational calculus were called "tuple calculus" and "domain calculus". The difference was mainly which of the notational variants above they used.

A relational calculus query has the form $\{\boldsymbol{u} \mid \varphi\}$ where $\boldsymbol{u}$ is a list of terms, i.e., variables or constants, φ is a formula, and the variables in $\boldsymbol{u}$ are exactly the free variables in φ.

Recall conjunctive queries (Definition 25) and their extensions. The relational calculus query above can be seen as simply another notation for $ans(\boldsymbol{u}) \leftarrow \varphi$. The discussion about the correspondence to relational algebra depending on the syntactic form of φ is therefore as in the subsection on conjunctive queries.

Relational Algebra. Relational algebra is not a logic-based approach and does not really belong in this subsection, but it was the first formalisation of query languages and usually the frame of reference for later ones.

Relational algebra considers relations in the mathematical sense and a small number of operators with which relational expressions can be constructed. Typical operators are selection, projection, cartesian product, union, intersection, set difference, division, join. Some of these operators can be defined in terms of others. Various classes of relational algebra queries are characterised by the subset of operators they may use.

3.3.3 Logical Variations.

Range Restricted Quantification. Both natural language and mathematics tend to exercise some control over the range of quantified variables.

Rather than saying *"for everything there is something that is the first one's parent"*, formalised as $\forall x \exists y\, parent(y, x)$, it would seem more natural to say *"for every person there is a person who is the first one's parent"*, formalised as $\forall x(person(x) \Rightarrow \exists y(person(y) \wedge parent(y, x)))$. Similar examples abound in mathematics, where theorems rarely start with *"for everything there is something such that ..."*, but more typically with *"for every polynomial P there is an integer n such that ..."*

The characteristic of formulas resulting from such statements is that each quantifier for a variable is combined with an atom that restricts the range of the quantified variable. This intuition can be formalised as follows.

Definition 26 (Formula with range restricted quantification). *Let $\mathcal{L}$ be a signature.* $\mathcal{L}$-formulas with range restricted quantification*, here abbreviated* RR-formulas*, are defined inductively:*

1. *Each quantifier-free $\mathcal{L}$-formula is an RR-formula.*
2. *Each $\mathcal{L}$-formula constructed from a connective and an appropriate number of RR-formulas is an RR-formula.*
3. *If φ is an RR-formula and A is an atom and $\boldsymbol{x}$ is a subset of the free variables in A, then $\forall \boldsymbol{x}(A \Rightarrow \varphi)$ and $\exists \boldsymbol{x}(A \wedge \varphi)$ are RR-formulas. The atom A is called the* range *for the variables in $\boldsymbol{x}$.*

The atom A combined with a quantifier is also called a *guard*. The guarded fragment discussed earlier in connection with conjunctive queries is a further restriction of this class of formulas, with the additional requirement that all variables that are free in φ also occur in A. This enforces a kind of layering: all variables that are free in a subformula occur in the atom guarding the innermost quantifier in whose scope the subformula is.

[26] gives a generalisation of Definition 26 allowing for non-atomic ranges.

Many-Sorted First-Order Predicate Logic. In the field of programming languages it is advantageous to associate types with expressions. The same idea for first-order predicate logic is to associate *sorts*[9] with terms.

This requires a new symbol class called *sort symbols*. A signature then specifies for each relation symbol p not just an arity n, but an n-tuple of sort symbols written $(s_1 \times \ldots \times s_n)$, and for each function symbol f not just an arity n, but an $(n+1)$-tuple of sort symbols written $(s_1 \times \ldots \times s_n \rightarrow s_{n+1})$, where $s_1, \ldots, s_n$ are the argument sorts and s_{n+1} is the result sort of the symbol. Moreover, each variable is associated with a sort symbol.

With these modifications it is straightforward to extend the definitions of terms and atoms by the obvious compatibility requirements between argument sorts and result sorts of subterms.

[9] The word *type* would clash with historically established terminology.

Example 27 (Many-sorted first-order predicate logic).

Sort symbols	$\{person, company\}$	
Signature		
2-ary relation symbol	$married : person \times person$ $employs : company \times person$	
constant	$Tom : person$ $Mary : person$	$Web5.0 : company$
1-ary function symbol	$founder : company \to person$	
Formulas	$married(Tom, Mary) \wedge employs(Web5.0, Tom)$	
	$\exists x{:}company\ employs(x, founder(Web5.0))$	
	$\forall x{:}company\ \exists y{:}person\ employs(x, y)$	

In this example, $founder(Tom)$ is not a term because of the clash between the subterm's result sort *person* and the required argument sort *company*. Likewise, $married(Tom, Web5.0)$ is not a formula.

Classical first-order predicate logic is the special case of the many-sorted version with exactly one sort symbol, which is not explicitly written.

Many-sorted first-order predicate logic can be translated into the classical version. Each sort symbol s is translated into a unary relation symbol $\widehat{s}$. Part of the signature information from the example above translates as $\widehat{person}(Tom)$ and $\forall x(\widehat{company}(x) \Rightarrow \widehat{person}(founder(x))$. The last of the formulas from the example translates as $\forall x(\widehat{company}(x) \Rightarrow \exists y(\widehat{person}(y) \wedge employs(x, y)))$. Note that the translation results in a formula with range restricted quantification, the fragment discussed earlier.

Thus, introducing sorts does not affect the expressive power. But it allows static sort checking and thus improves error detection. Moreover, a many-sorted formalisation of a problem needs fewer and smaller formulas than the corresponding classical formalisation.

The idea of introducing sorts can be extended to hierarchies or networks of sorts without losing these advantages.

4 Declarative Semantics: Fundamentals of Classical Model Theory

The classical semantics of first-order predicate logic, i.e., the attribution of meaning to formulas of first-order predicate logic, follows an approach proposed by Alfred Tarski in the 1930s. This approach has a salient characteristic: The interpretation of a compound term and the truth value of a compound formula are defined recursively over the structure of the term or formula, respectively. As a consequence, to know the truth value in an interpretation $\mathcal{I}$ of a compound formula φ, it suffices to know the values in $\mathcal{I}$ of the immediate constituents of φ. This is clearly advantageous for computing, as it provides with a well-defined, finite, and restricted computation scope. However, this approach to semantics has a considerable drawback: its allowing for any kind of sets for interpreting terms makes it apparently incomputable.

A theorem due to Jacques Herbrand shows that this drawback is overcome if only *universal formulas* are considered: If such formulas are true in some interpretation, whose domain may well not be computably enumerable, then they are also true in a so-called *Herbrand interpretation*, whose domain is the computably enumerable set of all variable-free terms of the given signature. Furthermore, a technique known as *Skolemization*[10] transforms every formula into a universal formula while preserving satisfiability, i.e., the interpretability of a formula as true in some interpretation. Herbrand's theorem and Skolemization make entailment, and thus query answering, semi-decidable and thus amenable to computing.

The first and main subsection below introduces more precisely the notions, techniques, and results mentioned above as well as the treatment of equality in first-order predicate logic. The subsection is concluded by remarks on inadequacies of classical semantics for query languages. The shorter subsections following the main one show how some, if not all, of these inadequacies can be overcome. Alas, these solutions bear new problems.

4.1 Classical Tarski Model Theory

In the following, a signature $\mathcal{L}$ for first-order predicate logic is assumed.

4.1.1 Interpretations, Models, and Entailment.

Definition 28 (Variable assignment). *Let D be a nonempty set. A* variable assignment in D *is a function V mapping each variable to an element of D. We denote the image of a variable x under an assignment V by x^V.*

Definition 29 ($\mathcal{L}$-Interpretation). *Let $\mathcal{L}$ be a signature. An* $\mathcal{L}$-interpretation *is a triple $\mathcal{I} = (D, I, V)$ where*

- *D is a nonempty set called the* domain *or* universe (of discourse) *of $\mathcal{I}$.*
  Notation: *$dom(\mathcal{I}) := D$.*
- *I is a function defined on the symbols of $\mathcal{L}$ mapping*
  - *each n-ary function symbol f to an n-ary function $f^I : D^n \to D$. For $n = 0$ this means $f^I \in D$.*
  - *each n-ary relation symbol p to an n-ary relation $p^I \subseteq D^n$. For $n = 0$ this means either $p^I = \emptyset$ or $p^I = \{()\}$.*

  Notation: *$f^{\mathcal{I}} := f^I$ and $p^{\mathcal{I}} := p^I$.*
- *V is a variable assignment in D.*
  Notation: *$x^{\mathcal{I}} := x^V$.*

The domain is required to be nonempty because otherwise neither I nor V would be definable. Moreover, an empty domain would cause anomalies in the truth values of quantified formulas. As before, when the signature $\mathcal{L}$ is clear from context, we drop the prefix "$\mathcal{L}$-" and simply speak of interpretations.

[10] Named after Thoralf Skolem, one of its inventors. Moses Schönfinkel independently proposed it as well.

Definition 30. *The value of a term t in an interpretation $\mathcal{I}$, denoted $t^{\mathcal{I}}$, is an element of $dom(\mathcal{I})$ and inductively defined:*

1. *If t is a variable or a constant, then $t^{\mathcal{I}}$ is defined as above.*
2. *If t is a compound term $f(t_1, \ldots, t_n)$, then $t^{\mathcal{I}}$ is defined as $f^{\mathcal{I}}(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}})$*

Notation 31. *Let V be a variable assignment in D, let V' be a partial function mapping variables to elements of D, which may or may not be a total function. Then $V[V']$ is the variable assignment with*

$$x^{V[V']} = \begin{cases} x^{V'} & \text{if } x^{V'} \text{ is defined} \\ x^{V} & \text{if } x^{V'} \text{ is undefined} \end{cases}$$

Let $\mathcal{I} = (D, I, V)$ be an interpretation. Then $\mathcal{I}[V'] := (D, I, V[V'])$.

By $\{x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$ we denote the partial function that maps x_i to d_i and is undefined on other variables. In combination with the notation above, we omit the set braces and write $V[x_1 \mapsto d_1, \ldots, x_k \mapsto d_k]$ and $\mathcal{I}[x_1 \mapsto d_1, \ldots, x_k \mapsto d_k]$.

Definition 32 (Tarksi, model relationship). *Let $\mathcal{I}$ be an interpretation and φ a formula. The* relationship $\mathcal{I} \models \varphi$*, pronounced "$\mathcal{I}$* is a model of φ*" or "$\mathcal{I}$* satisfies φ*" or "φ* is true in $\mathcal{I}$*", and its negation $\mathcal{I} \not\models \varphi$, pronounced "$\mathcal{I}$* falsifies φ*" or "φ* is false in $\mathcal{I}$*", are defined inductively:*

$$\begin{array}{lll}
\mathcal{I} \models p(t_1, \ldots, t_n) & \textit{iff} & (t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) \in p^{\mathcal{I}} \qquad (n\text{-ary } p,\ n \geq 1) \\
\mathcal{I} \models p & \textit{iff} & () \in p^{\mathcal{I}} \qquad (0\text{-ary } p) \\
\mathcal{I} \not\models \bot & & \\
\mathcal{I} \models \top & & \\
\mathcal{I} \models \neg\psi & \textit{iff} & \mathcal{I} \not\models \psi \\
\mathcal{I} \models (\psi_1 \wedge \psi_2) & \textit{iff} & \mathcal{I} \models \psi_1 \textit{ and } \mathcal{I} \models \psi_2 \\
\mathcal{I} \models (\psi_1 \vee \psi_2) & \textit{iff} & \mathcal{I} \models \psi_1 \textit{ or } \mathcal{I} \models \psi_2 \\
\mathcal{I} \models (\psi_1 \Rightarrow \psi_2) & \textit{iff} & \mathcal{I} \not\models \psi_1 \textit{ or } \mathcal{I} \models \psi_2 \\
\mathcal{I} \models \forall x\, \psi & \textit{iff} & \mathcal{I}[x \mapsto d] \models \psi \textit{ for each } d \in D \\
\mathcal{I} \models \exists x\, \psi & \textit{iff} & \mathcal{I}[x \mapsto d] \models \psi \textit{ for at least one } d \in D
\end{array}$$

For a set S of formulas, $\mathcal{I} \models S$ iff $\mathcal{I} \models \varphi$ for each $\varphi \in S$.

Definition 33 (Semantic properties). *A formula, or a set of formulas, is*

valid *or a* tautology	*iff it is satisfied in each interpretation*
satisfiable	*iff it is satisfied in at least one interpretation*
falsifiable	*iff it is falsified in at least one interpretation*
unsatisfiable *or* inconsistent	*iff it is falsified in each interpretation*

Note that a formula or a set of formulas can be both satisfiable and falsifiable, for instance, any propositional atom p is. The formulas $(p \vee \neg p)$ and $\top$ are valid. The formulas $(p \wedge \neg p)$ and $\bot$ are unsatisfiable.

Definition 34 (Entailment and logical equivalence). *Let φ and ψ be formulas or sets of formulas.*

$\varphi \models \psi$, pronounced: "φ entails *ψ" or "ψ* is a (logical) consequence of *φ",*
iff for each interpretation $\mathcal{I}$: if $\mathcal{I} \models \varphi$ then $\mathcal{I} \models \psi$.

$\varphi \models\!\models \psi$, pronounced: "$\varphi$ is (logically) equivalent to *ψ",*
iff $\varphi \models \psi$ and $\psi \models \varphi$.

The following result is immediate. It shows that the semantic properties and entailment can be translated into each other. Being able to determine one of validity, unsatisfiability, or entailment, is sufficient to determine all of them.

Theorem 35 (Translatability between semantic properties and entailment). *Let φ and ψ be formulas.*

φ *is valid*	*iff* $\neg\varphi$ *is unsatisfiable*	*iff* $\top \models \varphi$.
φ *is unsatisfiable*	*iff* $\neg\varphi$ *is valid*	*iff* $\varphi \models \bot$.
$\varphi \models \psi$	*iff* $(\varphi \Rightarrow \psi)$ *is valid*	*iff* $(\varphi \wedge \neg\psi)$ *is unsatisfiable.*

Being able to determine validity or unsatisfiability is also sufficient to determine logical equivalence, which is just mutual entailment. The definition of $\varphi \models\!\!\models \psi$ means that φ and ψ have the same models. Either of them may be replaced by the other without affecting any truth values. This is often exploited for transformations in proofs or in optimising queries or rule sets.

Proposition 36 (Model-preserving transformations). *Let φ, φ', ψ, ψ', χ be formulas. The following equivalences hold:*

- $\varphi \models\!\!\models \varphi'$ *if* φ' *is a rectified form of* φ
- $\varphi \models\!\!\models \varphi'$ *if* $\psi \models\!\!\models \psi'$ *and* φ' *is obtained from* φ *by replacing an occurrence of the subformula* ψ *by* ψ'

- $(\varphi \vee \psi) \models\!\!\models (\psi \vee \varphi)$ $\qquad$ $(\varphi \wedge \psi) \models\!\!\models (\psi \wedge \varphi)$
- $((\varphi \vee \psi) \vee \chi) \models\!\!\models (\varphi \vee (\psi \vee \chi))$ $\qquad$ $((\varphi \wedge \psi) \wedge \chi) \models\!\!\models ((\varphi \wedge (\psi \wedge \chi))$
- $((\varphi \vee \psi) \wedge \chi) \models\!\!\models ((\varphi \wedge \chi) \vee (\psi \wedge \chi))$ $\qquad$ $((\varphi \wedge \psi) \vee \chi) \models\!\!\models ((\varphi \vee \chi) \wedge (\psi \vee \chi))$
- $(\bot \vee \varphi) \models\!\!\models \varphi$ $\qquad$ $(\top \wedge \varphi) \models\!\!\models \varphi$
- $(\varphi \vee \neg\varphi) \models\!\!\models \top$ $\qquad$ $(\varphi \wedge \neg\varphi) \models\!\!\models \bot$
- $(\varphi \vee \varphi) \models\!\!\models \varphi$ $\qquad$ $(\varphi \wedge \varphi) \models\!\!\models \varphi$
- $(\varphi \vee (\varphi \wedge \psi)) \models\!\!\models \varphi$ $\qquad$ $(\varphi \wedge (\varphi \vee \psi)) \models\!\!\models \varphi$

- $\neg\neg\varphi \models\!\!\models \varphi$
- $\neg(\varphi \vee \psi) \models\!\!\models (\neg\varphi \wedge \neg\psi)$ $\qquad$ $\neg(\varphi \wedge \psi) \models\!\!\models (\neg\varphi \vee \neg\psi)$
- $(\varphi \Rightarrow \psi) \models\!\!\models (\neg\varphi \vee \psi)$ $\qquad$ $\neg(\varphi \Rightarrow \psi) \models\!\!\models (\varphi \wedge \neg\psi)$
- $((\varphi \vee \varphi') \Rightarrow \psi) \models\!\!\models ((\varphi \Rightarrow \psi) \wedge (\varphi' \Rightarrow \psi))$ $\qquad$ $(\varphi \Rightarrow (\psi \wedge \psi')) \models\!\!\models ((\varphi \Rightarrow \psi) \wedge (\varphi \Rightarrow \psi'))$
- $(\varphi \Rightarrow \bot) \models\!\!\models \neg\varphi$ $\qquad$ $(\top \Rightarrow \varphi) \models\!\!\models \varphi$

- *in general* $\forall x \exists y \varphi \not\models\!\!\models \exists y \forall x \varphi$

  $\forall x \forall y \varphi \models\!\!\models \forall y \forall x \varphi$ $\qquad$ $\exists x \exists y \varphi \models\!\!\models \exists y \exists x \varphi$
- $\neg\forall x \varphi \models\!\!\models \exists x \neg\varphi$ $\qquad$ $\neg\exists x \varphi \models\!\!\models \forall x \neg\varphi$
- $\forall x(\varphi \wedge \psi) \models\!\!\models (\forall x \varphi \wedge \forall x \psi)$ $\qquad$ $\exists x(\varphi \vee \psi) \models\!\!\models (\exists x \varphi \vee \exists x \psi)$

  $\exists x(\varphi \wedge \psi) \not\models\!\!\models (\exists x \varphi \wedge \exists x \psi)$ $\qquad$ *in general* $\qquad$ $\forall x(\varphi \vee \psi) \not\models\!\!\models (\forall x \varphi \vee \forall x \psi)$

  $\exists x(\varphi \wedge \psi) \models\!\!\models (\varphi \wedge \exists x \psi)$ $\qquad$ *if* x *is not free in* φ $\qquad$ $\forall x(\varphi \vee \psi) \models\!\!\models (\varphi \vee \forall x \psi)$
- $\forall x \varphi \models\!\!\models \varphi$ $\qquad$ *if* x *is not free in* φ $\qquad$ $\exists x \varphi \models\!\!\models \varphi$

By exploiting these equivalences, one can take any formula and, without affecting truth values, rectify it, translate $\Rightarrow$ into the other connectives, move quantifiers to the front and $\neg$ into subformulas.

Theorem 37. *Every formula is equivalent to a formula in prenex form. Moreover, every formula is equivalent to a formula in prenex form whose matrix is a conjunction of disjunctions of literals.*

Every universal *formula is equivalent to a conjunction of clauses and equivalent to a finite set of clauses (each clause representing its universal closure).*

The entailment relationship $\varphi \models \psi$ formalises the concept of *logical consequence*. From premises φ follows a conclusion ψ iff every model of the premises is a model of the conclusion.

A major concern in logic used to be the development of *calculi*, also called *proof systems*, which formalise the notion of deductive inference. A calculus defines derivation rules, with which formulas can be derived from formulas by purely syntactic operations. For example, a typical derivation rule might say "from $\neg\neg\varphi$ derive φ". The derivability relationship $\varphi \vdash \psi$ for a calculus holds iff there is a finite sequence of applications of derivation rules of the calculus, which applied to φ result in ψ.

Ideally, derivability should mirror entailment: a calculus is called *sound* iff whenever $\varphi \vdash \psi$ then $\varphi \models \psi$ and *complete* iff whenever $\varphi \models \psi$ then $\varphi \vdash \psi$.

Theorem 38 (Gödel, completeness theorem). *There exist calculi for first-order predicate logic such that $S \vdash \varphi$ iff $S \models \varphi$ for any set S of closed formulas and any closed formula φ.*

Thus, the semantic notion of entailment coincides with the syntactic notion of derivability. This correspondence opens up the prospects to obtain logical consequences by computation, but there are limits to these prospects.

Theorem 39 (Church-Turing, undecidability theorem). *Derivability in a correct and complete calculus is not decidable for signatures with at least one non-propositional relation symbol and a relation or function symbol of arity ≥ 2.*

A corollary of Gödel's completeness theorem is the following famous result.

Theorem 40 (Gödel-Malcev, finiteness or compactness theorem). *Let S be an infinite set of closed formulas. If every finite subset of S is satisfiable, then S is satisfiable.*

The contrapositive of the finiteness/compactness theorem, combined with its trivial converse, is often useful: a set S of closed formulas is unsatisfiable iff some finite subset of S is unsatisfiable.

Corollary 41. *For signatures with at least one non-propositional relation symbol and a relation or function symbol of arity ≥ 2, entailment, unsatisfiability, and validity are semi-decidable but not decidable, and non-entailment, satisfiability, and falsifiability are not semi-decidable.*

4.1.2 Theories.

Definition 42 (Model class). *Let $\mathcal{L}$ be a signature and S a set of $\mathcal{L}$-formulas. $Mod_{\mathcal{L}}(S)$ is the class of all $\mathcal{L}$-interpretations $\mathcal{I}$ with $\mathcal{I} \models S$, i.e., the class of all $\mathcal{L}$-models of S.*

We simply write $Mod(S)$ without "$\mathcal{L}$" when we leave the signature $\mathcal{L}$ implicit. Note that in general the class $Mod(S)$ is not a set. For satisfiable S it is nonempty. If it were a set, $Mod(S)$ could be the domain of another model of S, which would be a member of $Mod(S)$ – a similarly ill-defined notion as "the set of all sets".

Definition 43 (Theory). *A* theory *is a set T of $\mathcal{L}$-formulas that is closed under entailment, i.e., for each $\mathcal{L}$-formula φ, if $T \models \varphi$ then $\varphi \in T$.*

The theory of a class $\mathcal{K}$ of $\mathcal{L}$-interpretations*, denoted by $Th(\mathcal{K})$, is the set of all $\mathcal{L}$-formulas φ with $\mathcal{I} \models \varphi$ for each $\mathcal{I} \in \mathcal{K}$, i.e., the set of formulas satisfied by all interpretations in $\mathcal{K}$.*

The theory of a set S of $\mathcal{L}$-formulas*, denoted by $Th(S)$, is $Th(Mod_{\mathcal{L}}(S))$.*

Proposition 44. *$Th(\mathcal{K})$ and $Th(S)$ as defined above are indeed theories, i.e., closed under entailment. In particular, $Th(S) = \{\varphi \mid S \models \varphi\}$.*

Thus, *Th* can also be regarded as a *closure operator* for the closure under entailment of a set of formulas.

Proposition 45. *Th is a closure operator, i.e., for all sets S, S' of formulas:*

- *$S \subseteq Th(S)$* *(Th is* extensive*)*
- *if $S \subseteq S'$ then $Th(S) \subseteq Th(S')$* *(Th is* monotonic*)*
- *$Th(Th(S)) = Th(S)$* *(Th is* idempotent*)*

Definition 46 (Axiomatisation). *An* axiomatisation *of a theory T is a set S of formulas with $Th(S) \models\!\models T$.*

A theory is finitely axiomatisable *if it has an axiomatisation that is finite.*

Finite axiomatisability is important in practice because proofs have finite length and are built up from axioms. If there are finitely many axioms, then the set of all possible proofs is computably enumerable.

Theorem 47. *There are theories that are finitely axiomatisable and theories that are not.*

The theory of equivalence relations is finitely axiomatisable by the three formulas for reflexivity, symmetry, and transitivity. The theory of the natural numbers with addition, multiplication, and the less-than relation is, by Gödel's famous incompleteness theorem, not finitely axiomatisable.

Two trivial theories are finitely axiomatisable. $Th(\emptyset)$ is the set of all valid $\mathcal{L}$-formulas because $Mod_{\mathcal{L}}(\emptyset)$ consists of all $\mathcal{L}$-interpretations. If $S = \{\bot\}$ or any other unsatisfiable set of $\mathcal{L}$-formulas, $Th(S)$ is the set of all $\mathcal{L}$-formulas because $Mod_{\mathcal{L}}(S)$ is empty. In the literature this case is sometimes excluded by adding the requirement of satisfiability to the definition of a theory.

4.1.3 Substitutions and Unification.

Definition 48 (Substitution). *A* substitution *is a function σ, written in postfix notation, that maps terms to terms and is*

- *homomorphous, i.e., $f(t_1,\ldots,t_n)\sigma = f(t_1\sigma,\ldots,t_n\sigma)$ for compound terms and $c\sigma = c$ for constants.*
- *identical almost everywhere, i.e., $\{x \mid x \text{ is a variable and } x\sigma \neq x\}$ is finite.*

The domain *of a substitution σ is the finite set of variables on which it is not identical. Its* codomain *is the set of terms to which it maps its domain.*

A substitution σ is represented by the finite set $\{x_1 \mapsto x_1\sigma,\ldots,x_k \mapsto x_k\sigma\}$ where $\{x_1,\ldots,x_k\}$ is its domain and $\{x_1\sigma,\ldots,x_k\sigma\}$ is its codomain.

Mind the difference between Notation 31 and Definition 48: With variable assignments, the notation $\{x_1 \mapsto d_1,\ldots,x_k \mapsto d_k\}$ represents a *partial* function, which for variables other than $x_1,\ldots,x_k$ is undefined. With substitutions, the notation $\{x_1 \mapsto t_1,\ldots,x_k \mapsto t_k\}$ represents a *total* function, which for variables other than $x_1,\ldots,x_k$ is the identity.

For example, the substitution mapping x to a and y to $f(y)$ and any other variable to itself is represented by $\{x \mapsto a,\ y \mapsto f(y)\}$. The shortened notation $\{x/a,\ y/f(y)\}$ will also be used in later sections of this survey.[11]

Substitutions can be combined by functional composition, $t(\sigma\tau) = (t\sigma)\tau$. The identity substitution ε, which maps every term to itself, is the neutral element for functional composition.

Definition 49 (Subsumption ordering). *A term s* subsumes *a term t, denoted $s \leq t$, iff there exists a substitution ϑ with $s\vartheta = t$. One also says that t is an* instance *of s, or t is* more specific *than s, or s is* more general *than t.*

A substitution σ subsumes *a substitution τ, denoted $\sigma \leq \tau$, iff there exists a substitution ϑ with $\sigma\vartheta = \tau$. One also says that τ is an* instance *of σ, or τ is* more specific *than σ, or σ is* more general *than τ.*

Two terms mutually subsume each other iff they are equal up to variable renaming. Mutual subsumption is an equivalence relation, on whose equivalence classes $\leq$ is a well-founded partial ordering. Its minimum is the equivalence class of all variables. Its maximal elements are the singleton classes of ground terms. If there are function symbols with arity > 0, there are infinite strictly increasing chains: $x \leq f(x) \leq f(f(x)) \leq f(f(f(x))) \leq \ldots$

In the case of substitutions, the ordering is also well-founded. The equivalence class of variable permutations, which includes the identity substitution ε, is the minimum. There are no maximal elements: $\sigma \leq \sigma\{x \mapsto y\}$ for any x, y not in the domain of σ.

Definition 50 (Unification). *Two terms s and t are* unifiable*, if there exists a substitution σ with $s\sigma = t\sigma$. In this case σ is called a* unifier *of s and t.*

[11] In the literature several notational conventions coexist, including the ones used here and the reverse form $\{a/x,\ f(y)/y\}$. This can be confusing in cases like $\{u/v\}$.

A most general unifier *or* mgu *is a minimal element w.r.t. the subsumption ordering among the set of all unifiers of s and t. If σ is a most general unifier of s and t, the term $s\sigma$ is called a* most general common instance *of s and t.*

Theorem 51 (Robinson [138], unification theorem). *Any most general unifier of two terms subsumes all their unifiers.*

Thus, any two most general unifiers of two terms mutually subsume each other. This implies that any two most general common instances of two terms are equal up to variable renaming. Furthermore, among the unifiers of two terms there is always an *idempotent* most general unifier σ, that is, $\sigma\sigma = \sigma$. This is the case iff none of the variables from its domain occurs in its codomain.

Definition 52 (Ground substitution, ground instance). *A* ground substitution *is a substitution whose codomain consists of ground terms only. A* grounding substitution for a term t *is a ground substitution σ whose domain includes all variables in t, such that $t\sigma$ is ground. A* ground instance *of t is an instance of t that is ground.*

The application of a substitution σ to a set or tuple of terms, to a quantifier-free formula or set of quantifier-free formulas, or to other mathematical objects having terms as constituents, is defined by canonical extension. Thus, the notion of an instance or a ground instance or a grounding substitution for such an object is also defined canonically. For example, let φ be $p(x) \wedge q(y,z) \Rightarrow r(x,f(y),z)$ and $\sigma = \{x \mapsto f(v),\ y \mapsto a\}$. Then $\varphi\sigma$ is $p(f(v)) \wedge q(a,z) \Rightarrow r(f(v),f(a),z)$. One grounding substitution for φ is $\{x \mapsto a,\ y \mapsto f(a),\ z \mapsto a,\ v \mapsto a\}$.

For formulas containing quantifiers, however, it is technically more intricate to define the corresponding notions. We define only special cases involving ground substitutions.

Definition 53 (Instance of a formula). *Let φ be a formula and σ a ground substitution. Then $\varphi\sigma$ is the formula obtained from φ by replacing each free variable occurrence x in φ by $x\sigma$.*

For example, let φ be the formula $(\forall x[\exists x p(x) \wedge q(x)] \Rightarrow [r(x,y) \vee \forall x s(x)])$ and $\sigma = \{x \mapsto a\}$. Then $\varphi\sigma$ is $(\forall x[\exists x p(x) \wedge q(x)] \Rightarrow [r(a,y) \vee \forall x s(x)])$. Note that there may be three kinds of variable occurrences in φ. Those that are free and in the domain of σ, such as x in $r(x,y)$, become ground terms in $\varphi\sigma$. Those that are free and not in the domain of σ, such as y in $r(x,y)$, remain free in $\varphi\sigma$. Those that are bound in φ, such as x in $p(x)$, remain bound in $\varphi\sigma$. Because of the latter case it does not make sense to define grounding substitutions for formulas with quantifiers.

Definition 54 (Ground instance of a formula). *Let φ be a formula. Let φ' be a rectified form of φ. Let φ'' be obtained from φ' by removing each occurrence of a quantifier for a variable. A* ground instance *of φ is a ground instance of φ''.*

For example, let φ be $(\forall x[\exists x p(x) \wedge q(x)] \Rightarrow [r(x,y) \vee \forall x s(x)])$. A rectified form with quantifiers removed is $([p(v) \wedge q(u)] \Rightarrow [r(x,y) \vee s(w)])$. Assuming a signature with constants a, b, c, d and unary function symbol f, two ground instances of φ are $([p(a) \wedge q(a)] \Rightarrow [r(a,a) \vee s(a)])$ and $([p(a) \wedge q(f(b))] \Rightarrow [r(f(f(c)),c) \vee s(d)])$.

In general the set of ground instances of a non-ground term or formula is infinite, but it is always computably enumerable.

Typically, one is only interested in ground instances of formulas in which all variables are of the same kind: free or universally quantified or existentially quantified.

4.1.4 Herbrand Interpretations. An interpretation according to Tarski's model theory may use any nonempty set as its domain. Herbrand interpretations are interpretations whose domain is the so-called Herbrand universe, the set of all ground terms constructible with the signature considered. This is a syntactic domain, and under the common assumption that all symbol sets of the signature are computably enumerable, so is the Herbrand universe. In an Herbrand interpretation quantification reduces to ground instantiation.

A fundamental result is that Herbrand interpretations can imitate interpretations with arbitrary domains provided that the formulas under consideration are universal. A set of universal formulas has an arbitrary model iff it has an Herbrand model. As a consequence, it is satisfiable iff the set of its ground instances is, which is essentially a propositional problem. These results are the key to algorithmic treatment of the semi-decidable semantic notions: validity, unsatisfiability, entailment.

Definition 55 (Herbrand universe and base). *Let $\mathcal{L}$ be a signature for first-order predicate logic. The* Herbrand universe *$HU_{\mathcal{L}}$ is the set of all ground $\mathcal{L}$-terms. The* Herbrand base *$HB_{\mathcal{L}}$ is the set of all ground $\mathcal{L}$-atoms.*

If $\mathcal{L}$ does not specify any constant, $HU_{\mathcal{L}}$ is empty. From now on we assume that $\mathcal{L}$ specifies at least one constant. As usual, we leave the signature $\mathcal{L}$ implicit when it is clear from context, and write simply HU and HB without "$\mathcal{L}$".

Definition 56 (Herbrand interpretation). *An interpretation $\mathcal{I}$ is an* Herbrand interpretation *if $dom(\mathcal{I}) = HU$ and $f^{\mathcal{I}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for each n-ary function symbol f and all $t_1, \ldots, t_n \in HU$.*

Thus, the value of a ground term t in an Herbrand interpretation is the term t itself. Furthermore, it turns out that the truth values of quantified formulas depend on the truth values of their ground instances.

Theorem 57. *Let $\mathcal{I}$ be an Herbrand interpretation and φ a formula that may or may not contain a free occurrence of the variable x.*

- $\mathcal{I} \models \forall x \varphi$ *iff* $\mathcal{I} \models \varphi\{x \mapsto t\}$ *for each $t \in HU$.*
- $\mathcal{I} \models \exists x \varphi$ *iff* $\mathcal{I} \models \varphi\{x \mapsto t\}$ *for at least one $t \in HU$.*

According to Definition 32, the right hand sides should be $\mathcal{I}[x \mapsto t] \models \varphi$. The theorem states that the effect of modifying the interpretation's variable assignment can be achieved by applying the ground substitution $\{x \mapsto t\}$ to φ. This result crucially depends on the interpretation's being an Herbrand interpretation. Here are two counter-examples for non-Herbrand interpretations:

Example 58. Consider a signature containing a unary relation symbol p and a constant a and no other symbols. Then the only member of HU is a. Let $\mathcal{I}$ be the non-Herbrand interpretation with $dom(\mathcal{I}) = \{1, 2\}$ and $a^{\mathcal{I}} = 1$ and $p^{\mathcal{I}} = \{1\}$. Then $\mathcal{I} \models p(a)$ and $\mathcal{I}[x \mapsto 1] \models p(x)$ and $\mathcal{I}[x \mapsto 2] \not\models p(x)$.

- $\mathcal{I} \not\models \forall x\, p(x)$, but $\mathcal{I} \models p(x)\{x \mapsto t\}$ for each $t \in HU = \{a\}$.
- $\mathcal{I} \models \exists x\, \neg p(x)$, but $\mathcal{I} \not\models \neg p(x)\{x \mapsto t\}$ for each $t \in HU = \{a\}$.

Example 59. Consider a signature containing a unary relation symbol p and for each arity an infinite and enumerable set of function symbols of that arity. Then the set HU is enumerable. Let $\mathcal{I}$ be a non-Herbrand interpretation with $dom(\mathcal{I}) = \mathbb{R}$, the set of real numbers, with arbitrary definitions for the constants and function symbols, and $p^{\mathcal{I}} = \{t^{\mathcal{I}} \mid t \in HU\} \subseteq \mathbb{R}$.

Since HU is enumerable, there are $r \in \mathbb{R}$ with $r \notin p^{\mathcal{I}}$, thus $\mathcal{I}[x \mapsto r] \not\models p(x)$.

- $\mathcal{I} \not\models \forall x\, p(x)$, but $\mathcal{I} \models p(x)\{x \mapsto t\}$ for each $t \in HU$.
- $\mathcal{I} \models \exists x\, \neg p(x)$, but $\mathcal{I} \not\models \neg p(x)\{x \mapsto t\}$ for each $t \in HU$.

In both examples the reason why the theorem does not hold is that the interpretation uses a domain with more elements than there are ground terms. The latter example shows that this is not only a phenomenon of finite signatures.

Corollary 60. *Let S be a set of universal closed formulas and S_{ground} the set of all ground instances of members of S. For each Herbrand interpretation $\mathcal{I}$ holds $\mathcal{I} \models S$ iff $\mathcal{I} \models S_{ground}$.*

Terms have a fixed value in all Herbrand interpretations. Only the interpretation of relation symbols is up to each particular Herbrand interpretation. This information can conveniently be represented by a set of ground atoms.

Definition 61 (Herbrand interpretation represented by ground atoms). *Let V be some fixed variable assignment in HU. Let $B \subseteq HB$ be a set of ground atoms. Then $HI(B)$ is the Herbrand interpretation with variable assignment V and $p^{HI(B)} = \{(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n) \in B\}$ for each n-ary relation symbol p.*

Thus, a set B of ground atoms represents the Herbrand interpretation $HI(B)$ that satisfies all ground atoms in B and falsifies all ground atoms not in B. Except for the variable assignment, $HI(B)$ is uniquely determined by B. The notation introduced earlier, $HI(B)[V]$, can be used to specify a particular variable assignment.

Definition 62 (Herbrand interpretation induced by an interpretation). *Let $\mathcal{I}$ be an arbitrary interpretation. The Herbrand interpretation induced by $\mathcal{I}$, denoted $HI(\mathcal{I})$, is $HI(\{A \in HB \mid \mathcal{I} \models A\})$.*

Proposition 63. *For each signature and up to the fixed variable assignment:*
$HI(\mathcal{I}) = \mathcal{I}$ *iff* $\mathcal{I}$ *is an Herbrand interpretation.*

There is a bijection between the set of all Herbrand interpretations and the set of all subsets of HB.

Theorem 64 (Herbrand model induced by a model). *Let φ be a universal closed formula. Each model of φ induces an Herbrand model of φ, that is, for each interpretation $\mathcal{I}$, if $\mathcal{I} \models \varphi$ then $HI(\mathcal{I}) \models \varphi$.*

The converse holds for ground formulas, but not in general. Reconsider Examples 58 and 59 above. In both of them $HI(\mathcal{I}) \models \forall x\, p(x)$, but $\mathcal{I} \not\models \forall x\, p(x)$. The examples also show that the correct direction does not hold for non-universal formulas. In both of them $\mathcal{I} \models \exists x\, \neg p(x)$, but $HI(\mathcal{I}) \not\models \exists x\, \neg p(x)$.

At first glance the result may seem weak, but it establishes that if there is a model, there is an Herbrand model. The converse is trivial. Taking into account the finiteness/compactness theorem, we get:

Corollary 65 ("Herbrand Theorem"). *Let S be a set of universal closed formulas and let S_{ground} be the set of all ground instances of members of S. S is unsatisfiable iff S has no Herbrand model iff S_{ground} has no Herbrand model iff there is a finite subset of S_{ground} that has no Herbrand model.*

The latter is essentially a propositional problem, see the discussion on propositional vs. ground on page 9. In the literature, the name *Herbrand theorem* usually refers to this reduction of semantic notions from the first-order level to the propositional level.

4.1.5 Skolemization. The results on Herbrand interpretations cover only universal formulas. Skolemization is a technique to transform a non-universal formula φ into a universal formula φ_{sko}. The transformation is not model-preserving, in general $\varphi \not\models\!\models \varphi_{sko}$. But it preserves the existence of models, φ is satisfiable iff φ_{sko} is satisfiable.

Here is an example of the transformation:

φ is the $\mathcal{L}$-formula $\quad \forall y \forall z(\mathit{married}(y,z) \Rightarrow \exists x\, \mathit{parent}(y,x))$
φ_{sko} is the $\mathcal{L}_{sko}$-formula $\quad \forall y \forall z(\mathit{married}(y,z) \Rightarrow \mathit{parent}(y, f(y)))$

where $\mathcal{L}$ is a signature with 2-ary relation symbols *married* and *parent*, in which the 1-ary function symbol f does not occur, and $\mathcal{L}_{sko}$ is $\mathcal{L}$ extended with f. Both formulas are satisfiable and falsifiable.

Let $\mathcal{I}$ be an $\mathcal{L}_{sko}$-interpretation with $\mathcal{I} \models \varphi_{sko}$. Intuitively, for each y for which $\mathit{parent}(y, f(y))$ is true in $\mathcal{I}$, there exists at least one x for which $\mathit{parent}(y,x)$ is true in $\mathcal{I}$, namely $f(y)$. This intuition can be used to show formally that $\mathcal{I} \models \varphi$. Thus, $\varphi_{sko} \models \varphi$.

The converse does not hold. Let $\mathcal{I}$ be the $\mathcal{L}_{sko}$-interpretation with a domain of people for whom it is indeed the case that every married person has a child, and with the "natural" interpretation of the relation symbols and $f^{\mathcal{I}}$ the identity function on the domain. Now $\mathcal{I} \models \varphi$, but $\mathcal{I} \not\models \varphi_{sko}$ because no-one from the domain is their own parent. Thus, $\varphi \not\models \varphi_{sko}$.

However, the fact that $\mathcal{I} \models \varphi$, does not depend on the interpretation of f. We construct another interpretation $\mathcal{I}'$ that differs from $\mathcal{I}$ just in that $f^{\mathcal{I}'}$ maps only childless people to themselves but maps people with children to their firstborn. Still $\mathcal{I}' \models \varphi$, but also $\mathcal{I}' \models \varphi_{sko}$. From a model of φ we have constructed another model of φ, which is also a model of φ_{sko}.

The principles illustrated with this example apply in general.

Definition 66 (Skolemization step). *Let φ be a rectified closed formula containing an occurrence of a subformula $Qx\psi$ where Q is an existential quantifier with positive or a universal quantifier with negative polarity in φ. Let the variables with free occurrences in ψ be $\{x, y_1, \ldots, y_k\}$, $k \geq 0$. Let f be a k-ary function symbol that does not occur in φ.*

Let φ_s be φ with the occurrence of $Qx\psi$ replaced by $\psi\{x \mapsto f(y_1, \ldots, y_k)\}$. Then the transformation from φ to φ_s is called a Skolemization step *with* Skolem function symbol f *and* Skolem term $f(y_1, \ldots, y_k)$.

Proposition 67. *If a Skolemization step transforms φ to φ_s, then $\varphi_s \models \varphi$, and for each interpretation $\mathcal{I}$ with $\mathcal{I} \models \varphi$ there exists an interpretation $\mathcal{I}'$ with $\mathcal{I}' \models \varphi$ and $\mathcal{I}' \models \varphi_s$. Moreover, $\mathcal{I}'$ coincides with $\mathcal{I}$ except possibly $f^{\mathcal{I}'} \neq f^{\mathcal{I}}$.*

Corollary 68 (Skolemization). *Let $\mathcal{L}$ be a signature for first-order predicate logic and S a computably enumerable set of closed $\mathcal{L}$-formulas. There is an extension $\mathcal{L}_{sko}$ of $\mathcal{L}$ by new function symbols and a computably enumerable set S_{sko} of* universal *closed $\mathcal{L}$-formulas, such that S is unsatisfiable iff S_{sko} is unsatisfiable.*

4.1.6 Equality. So far we have considered results for the version of first-order predicate logic without equality. Let us now assume a signature $\mathcal{L}$ containing a special 2-ary relation symbol $\doteq$ for equality.

Definition 69 (Normal interpretation). *An interpretation $\mathcal{I}$ is* normal, *iff it interprets the relation symbol $\doteq$ with the equality relation on its domain, i.e., iff $\doteq^{\mathcal{I}}$ is the relation $\{(d, d) \mid d \in dom(\mathcal{I})\}$. For formulas or sets of formulas φ and ψ:*

$\mathcal{I} \models_= \varphi$ iff $\mathcal{I}$ is normal and $\mathcal{I} \models \varphi$.

$\varphi \models_= \psi$ iff for each normal interpretation $\mathcal{I}$: if $\mathcal{I} \models_= \varphi$ then $\mathcal{I} \models_= \psi$.

The version of first-order predicate logic with built-in equality simply makes the normality requirement part of the definition of an interpretation. Let us now investigate the relationship between the two versions.

Definition 70 (Equality axioms). *Given a signature $\mathcal{L}$ with 2-ary relation symbol $\doteq$, the set $EQ_{\mathcal{L}}$ of* equality axioms *for $\mathcal{L}$ consists of the formulas:*

- $\forall x\ x \doteq x$ *(reflexivity of $\doteq$)*
- $\forall x \forall y (x \doteq y \Rightarrow y \doteq x)$ *(symmetry of $\doteq$)*
- $\forall x \forall y \forall z ((x \doteq y \wedge y \doteq z) \Rightarrow x \doteq z)$ *(transitivity of $\doteq$)*
- *for each n-ary function symbol f, $n > 0$* *(substitution axiom for f)*
  $\forall x_1 \ldots x_n \forall x'_1 \ldots x'_n ((x_1 \doteq x'_1 \wedge \ldots \wedge x_n \doteq x'_n) \Rightarrow f(x_1, \ldots, x_n) \doteq f(x'_1, \ldots, x'_n))$

– *for each n-ary relation symbol p, $n > 0$* *(substitution axiom for p)*
$\forall x_1 \ldots x_n \forall x'_1 \ldots x'_n((x_1 \doteq x'_1 \wedge \ldots \wedge x_n \doteq x'_n \wedge p(x_1, \ldots, x_n)) \Rightarrow p(x'_1, \ldots, x'_n))$

Note that $EQ_{\mathcal{L}}$ may be infinite, depending on $\mathcal{L}$. Actually, symmetry and transitivity of $\doteq$ follow from reflexivity of $\doteq$ and the substitution axiom for $\doteq$ and could be omitted.

Theorem 71 (Equality axioms).

– *For each interpretation $\mathcal{I}$, if $\mathcal{I}$ is normal then $\mathcal{I} \models_= EQ_{\mathcal{L}}$.*
– *For each interpretation $\mathcal{I}$ with $\mathcal{I} \models EQ_{\mathcal{L}}$ there is a normal interpretation $\mathcal{I}_=$ such that for each formula φ: $\mathcal{I} \models \varphi$ iff $\mathcal{I}_= \models_= \varphi$.*
– *For each set S of formulas and formula φ: $EQ_{\mathcal{L}} \cup S \models \varphi$ iff $S \models_= \varphi$.*

Corollary 72 (Finiteness or compactness theorem with equality). *Let S be an infinite set of closed formulas with equality. If every finite subset of S has a normal model, then S has a normal model.*

The results of Theorem 71 indicate that the equality axioms seem to define pretty much of the intended meaning of $\doteq$. However, they do not define it fully, nor does any other set of formulas.

Theorem 73 (Model extension theorem). *For each interpretation $\mathcal{I}$ and each set $D' \supseteq dom(\mathcal{I})$ there is an interpretation $\mathcal{I}'$ with $dom(\mathcal{I}') = D'$ such that for each formula φ: $\mathcal{I} \models \varphi$ iff $\mathcal{I}' \models \varphi$.*

Proof. (sketch) Fix an arbitrary element $d \in dom(\mathcal{I})$. The idea is to let all "new" elements behave exactly like d. Define an auxiliary function π mapping each "new" element to d and each "old" element to itself:

$\pi : D' \to dom(\mathcal{I})$, $\pi(d') := d$ if $d' \notin dom(\mathcal{I})$, $\pi(d') := d'$ if $d' \in dom(\mathcal{I})$.

Define $f^{\mathcal{I}'} : {D'}^n \to D'$, $f^{\mathcal{I}'}(d_1, \ldots, d_n) := f^{\mathcal{I}}(\pi(d_1), \ldots, \pi(d_n))$ and $p^{\mathcal{I}'} \subseteq {D'}^n$, $p^{\mathcal{I}'} := \{(d_1, \ldots, d_n) \in {D'}^n \mid (\pi(d_1), \ldots, \pi(d_n)) \in p^{\mathcal{I}}\}$ for all signature symbols and arities. □

By this construction, if $(d, d) \in \doteq^{\mathcal{I}}$ then $(d, d') \in \doteq^{\mathcal{I}'}$ for each $d' \in D'$ and the fixed element $d \in dom(\mathcal{I})$. Hence, any proper extension $\mathcal{I}'$ of a normal interpretation $\mathcal{I}$ does not interpret $\doteq$ with the equality relation on D'.

Corollary 74. *Every satisfiable set of formulas has non-normal models.*

Every model of $EQ_{\mathcal{L}}$ interprets $\doteq$ with a congruence relation on the domain. The equality relation is the special case with singleton congruence classes. Because of the model extension theorem, there is no way to prevent models with larger congruence classes, unless equality is treated as built-in by making interpretations normal by definition.

4.1.7 Model Cardinalities.

Theorem 75. *Lower bounds of model cardinalities can be expressed in first-order predicate logic without equality.*

Example: all models of the following satisfiable set of formulas have domains with cardinality ≥ 3:

$$\begin{array}{l}\{\ \exists x_1(\ \ p_1(x_1) \wedge \neg p_2(x_1) \wedge \neg p_3(x_1)),\\ \ \ \exists x_2(\neg p_1(x_2) \wedge\ \ p_2(x_2) \wedge \neg p_3(x_2)),\\ \ \ \exists x_3(\neg p_1(x_3) \wedge \neg p_2(x_3) \wedge\ \ p_3(x_3))\ \ \}\end{array}$$

Example: all models of the following satisfiable set of formulas have infinite domains: $\{\ \forall x\, \neg(x < x),\ \ \forall x \forall y \forall z(x < y \wedge y < z \Rightarrow x < z),\ \ \forall x \exists y\, x < y\ \}$.

Theorem 76. *Upper bounds of model cardinalities cannot be expressed in first-order predicate logic without equality.*

Theorem 77. *Each satisfiable set of formulas without equality has models with infinite domain.*

Corollary 78. *Finiteness cannot be expressed in first-order predicate logic without equality.*

The above are immediate consequences of the model extension theorem 73. The remaining results are about the version of first-order predicate logic with built-in equality.

Theorem 79. *Bounded finiteness can be expressed in first-order predicate logic with equality. That is, for any given natural number $k \geq 1$, the upper bound k of model cardinalities can be expressed.*

Example: all normal models of the following satisfiable formula have domains with cardinality ≤ 3: $\exists x_1 \exists x_2 \exists x_3 \forall y(y \doteq x_1 \vee y \doteq x_2 \vee y \doteq x_3)$.

Theorem 80. *If a set of formulas with equality has arbitrarily large finite normal models, then it has an infinite normal model.*

Proof. Let S be such that for each $k \in \mathbb{N}$ there is a normal model of S whose domain has finite cardinality $> k$. For each $n \in \mathbb{N}$ let φ_n be the formula $\forall x_0 \ldots x_n \exists y(\neg(y \doteq x_0) \wedge \ldots \wedge \neg(y \doteq x_n))$ expressing "more than $n+1$ elements". Then every finite subset of $S \cup \{\varphi_n \mid n \in \mathbb{N}\}$ has a normal model. By the finiteness/compactness theorem with equality, $S \cup \{\varphi_n \mid n \in \mathbb{N}\}$ has a normal model, which obviously cannot be finite, but is also a normal model of S. □

Corollary 81. *A satisfiable set of formulas with equality has either only finite normal models of a bounded cardinality, or infinite normal models.*

Corollary 82. *Unbounded finiteness cannot be expressed in first-order predicate logic with equality.*

Theorem 83 (Löwenheim-Skolem). *Every satisfiable enumerable set of closed formulas has a model with a finite or infinite enumerable domain.*

Theorem 84 (Löwenheim-Skolem-Tarski). *If a set of closed formulas has a model of some infinite cardinality, it has a model of every infinite cardinality.*

4.1.8 Inadequacy of Tarski Model Theory for Query Languages. The domain of an interpretation according to Tarski may be any nonempty set. On the one hand, this has a tremendous advantage: first-order predicate logic can be used to model statements about any arbitrary application domain. On the other hand, it has effects that may be undesirable in the context of query languages.

For query languages, it is desirable that the following can be expressed:

1. By default, different constants are differently interpreted. The *unique name assumption* is such a frequent requirement in applications that a mechanism making it available by default would come in handy.

Tarski model theory does not provide such a mechanism. Interpretations may well interpret different constants identically, unless formulas explicitly prevent that. An explicit formalisation is cumbersome, albeit possible.

2. Function symbols are to be interpreted as term constructors. In many applications it makes sense to group pieces of data that belong together. For instance, a function symbol *person* might be used to construct a term such as $person(Mary, Miller, 1984)$, which simply serves as a compound data structure. In such cases it does not make sense to interpret the symbol *person* with arbitrary functions. It should be interpreted as just a term constructor.

This is not expressible in first-order predicate logic with Tarski model theory.

3. No more should hold than what is explicitly specified. A *closed world assumption* makes sense in query answering applications such as transportation timetables, which are based on the tacit understanding that no other than the explicitly listed connections are actually available.

Such a minimality restriction corresponds to an induction principle, which does not necessarily hold in Tarski interpretations. In fact, it is well-known that the induction principle cannot be expressed in first-order predicate logic with Tarski model theory.

4. Disregard infinite models. Real-world query answering applications are often finite by their nature and need to consider only interpretations with finite domains. In this case, infinite domains are not only superfluous, but they bring along phenomena that may be "strange" from the viewpoint of the application and would not be found in finite domains.

Consider an application about the hierarchy in enterprises, where a boss's boss is also a boss, but nobody is their own boss. The obvious conclusion is that there must be someone at the top of the hierarchy who does not have a boss. However, this conclusion is not justified for interpretations with infinite domains. The "strange" phenomenon that such a hierarchy may have no top at all has to be taken into account because of interpretations with infinitely many employees, although such interpretations are irrelevant for this application.

For a somewhat more contrived example, consider a group of married couples, where each husband has exactly one sister among the group and no wife

has more than one brother among the group. The obvious conclusion[12] is that every wife must have a brother, because otherwise there would be more brothers than sisters in the group, contradicting the assumptions about the numbers of siblings. However, this conclusion is not justified for interpretations with infinite domains. If there are as many couples as there are natural numbers, each husband number n can be the brother of wife number $n+1$, leaving wife number 0 without a brother, while everyone else has exactly one sibling in the group. The "strange" phenomenon that there may be a bijection between a set and a proper subset can only occur in infinite domains, but for the example such domains are irrelevant.

In order to avoid such "strange" phenomena, it would be necessary to restrict interpretations to finite ones. However, finiteness is not expressible in first-order predicate logic with Tarski model theory.

5. Definability of the transitive closure of a binary relation. The transitive closure of a binary relation is relevant in many query answering applications. Consider a traffic application, where a binary relation symbol r is used to represent direct connections between junctions, e.g., direct flights between airports. Another binary relation symbol t is used to represent direct or indirect connections of any *finite* length, i.e., with finitely many stopovers. Connections of infinite length do not make sense in this application.

The mathematical concept for "all connections of any finite length" is called the "transitive closure" of a binary relation. Thus, the intention is that t be interpreted as the relation that is the transitive closure of the relation with which r is being interpreted. An attempt to express this intention in first-order predicate logic could be the following closed formula:

$$\forall x \forall z \Big(t(x,z) \Leftrightarrow \big(r(x,z) \vee \exists y \left[t(x,y) \wedge t(y,z) \right] \big) \Big) \qquad (\star)$$

Here $\varphi \Leftrightarrow \psi$ abbreviates $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$. The $\Leftarrow$ direction makes sure that in every model $\mathcal{I}$ of $(\star)$ the relation $t^{\mathcal{I}}$ is transitive and includes the relation $r^{\mathcal{I}}$. The $\Rightarrow$ direction has the purpose to add an only-if part to the definition of $t^{\mathcal{I}}$.

Each interpretation interpreting t as the transitive closure of the interpretation of r, is indeed a model of $(\star)$. However, the intended converse, that every model of $(\star)$ interprets t as the transitive closure of the interpretation of r, does not hold:

Proof. Let $D = \{1{-}2^{-n} \mid n \in \mathbb{N}\} = \{0, \frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \frac{15}{16}, \ldots\}$.

Let $dom(\mathcal{I}) = D$, let $r^{\mathcal{I}} = \{(1{-}2^{-n}, 1{-}2^{-(n+1)}) \mid n \in \mathbb{N}\}$, the "immediate successor" relation on D, and let $t^{\mathcal{I}}$ be the arithmetic $<$-relation on D. Then $t^{\mathcal{I}}$ is the transitive closure of $r^{\mathcal{I}}$ and $\mathcal{I}$ is a model of $(\star)$.

Let $dom(\mathcal{I}') = D \cup \{1\}$, let $r^{\mathcal{I}'} = r^{\mathcal{I}}$, and let $t^{\mathcal{I}'}$ be the arithmetic $<$-relation on $D \cup \{1\}$. Then $t^{\mathcal{I}'}$ is not the transitive closure of $r^{\mathcal{I}'}$, it contains the additional pairs $(d, 1)$ for all $d \in D$, but there is no connection of finite length via the "immediate successor" relation on D between any $d \in D$ and 1. Yet, $\mathcal{I}'$ is also a model of $(\star)$. □

[12] Excluding polygamy, same-sex marriage, etc.

This may appear like another instance of point 4. But there are also counter-examples with finite domain:

Proof. Let $D = \{0, 1\}$, let $dom(\mathcal{I}) = D$, let $r^{\mathcal{I}} = \{(0, 0), (1, 1)\}$, the equality relation on D, and let $t^{\mathcal{I}} = \{(0, 0), (0, 1), (1, 1)\}$, the $\leq$-relation on D. The transitive closure of $r^{\mathcal{I}}$ is $r^{\mathcal{I}}$ itself. Thus, $t^{\mathcal{I}}$ is not the transitive closure of $r^{\mathcal{I}}$. Yet, $\mathcal{I}$ is a model of $(\star)$. □

If an interpretation interprets t as intended, then it is a model of $(\star)$, but the converse does not hold. Would some fine-tuning of $(\star)$ guarantee that the converse holds, too? In fact, no set of formulas can guarantee that.

Proof. Assume there is a satisfiable set S of closed formulas such that for each interpretation $\mathcal{I}$, $\mathcal{I} \models S$ iff $t^{\mathcal{I}}$ is the transitive closure of $r^{\mathcal{I}}$. That is, for each $(d_0, d) \in t^{\mathcal{I}}$ there are finitely many "stopover" elements $d_1, \ldots, d_k$ with $k \geq 0$, and $\{(d_0, d_1), \ldots, (d_k, d)\} \subseteq r^{\mathcal{I}}$.

Let a and b be constants not occurring in S. For each $n \in \mathbb{N}$ let φ_n be the closed formula as follows:

$$
\begin{array}{lll}
\varphi_0 & \text{is} & t(a, b) \wedge \neg r(a, b) \\
\varphi_1 & \text{is} & \varphi_0 \wedge \neg \exists x_1 (r(a, x_1) \wedge r(x_1, b)) \\
\varphi_2 & \text{is} & \varphi_1 \wedge \neg \exists x_1 \exists x_2 (r(a, x_1) \wedge r(x_1, x_2) \wedge r(x_2, b)) \\
\varphi_3 & \text{is} & \varphi_2 \wedge \neg \exists x_1 \exists x_2 \exists x_3 (r(a, x_1) \wedge r(x_1, x_2) \wedge r(x_2, x_3) \wedge r(x_3, b)) \\
\vdots & &
\end{array}
$$

In the traffic scenario, each formula φ_n expresses that there is a connection between (the interpretations of) a and b, but not with n or less stopovers.

For each $n \in \mathbb{N}$ the set $S \cup \{\varphi_n\}$ is satisfied by interpreting t as the transitive closure of r, and a and b as the endpoints of an r-chain with $n + 1$ "stopover" elements in between. Thus, every finite subset of $S \cup \{\varphi_n \mid n \in \mathbb{N}\}$ has a model. By the finiteness/compactness theorem, $S \cup \{\varphi_n \mid n \in \mathbb{N}\}$ has a model $\mathcal{I}$, which is thus also a model of S. In this model, $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in t^{\mathcal{I}}$, because $\mathcal{I}$ satisfies φ_0, but there is no finite set of "stopover" elements with $\{(a^{\mathcal{I}}, d_1), (d_1, d_2), \ldots, (d_k, b^{\mathcal{I}})\} \subseteq r^{\mathcal{I}}$, because $\mathcal{I}$ also satisfies φ_k.

Thus, S has a model $\mathcal{I}$ in which $t^{\mathcal{I}}$ is not the transitive closure of $r^{\mathcal{I}}$, which contradicts the assumption. □

The transitive closure of a base relation is the *smallest* transitive relation that includes the base relation. This inductive characterisation shows that the phenomenon is in fact an instance of point 3. In first-order predicate logic with Tarski model theory one can enforce that t is interpreted with a transitive relation that includes the transitive closure of the interpretation of r, but one cannot prevent that t is interpreted with a larger transitive relation.

Another instance of the same phenomenon is that the connectedness of a finite directed graph, i.e., the existence of a path of any possible finite length between two nodes, cannot be expressed.

6. Restrictions to specific classes of domains or to specific classes of interpretations. In general, applications are not concerned with all imaginable domains

and interpretations, but with limited classes of domains and interpretations. Such restrictions have to be expressed by appropriate formulas – if such formulas exist.

As discussed earlier, the restriction to domains with a given cardinality cannot be expressed in first-order predicate logic without equality. Nor can interpretations be restricted to normal ones. In these cases a way out is to use the version of first-order predicate logic with built-in equality.

But the same kind of problems may arise with application-specific concepts that cannot be expressed. For instance, an application about boards of trustees might require that they consist of an odd number of members in order to avoid inconclusive votes. In first-order predicate logic with or without built-in equality one cannot express the restriction to domains with odd cardinality. In this case there is no way out, a version with suitable built-ins is not available.

Several approaches aim at overcoming some of these problems 1 to 6. For example, considering only *Herbrand interpretations* and *Herbrand models* instead of general interpretations [88], addresses points 1 and 2. Considering only *minimal Herbrand models* addresses point 3. Considering only *finite* interpretations and models, the realm of *finite model theory* [61], addresses point 4. Unfortunately, all such approaches raise new problems themselves. Such approaches and problems are discussed below.

4.2 Herbrand Model Theory

Most of the information in this subsection is based on a technical report by Hinrichs and Genesereth [88]. Herbrand model theory restricts interpretations to Herbrand interpretations (without equality, except if otherwise stated).

Definition 85. *For formulas or sets of formulas φ and ψ:*
φ is Herbrand valid *iff it is satisfied in each Herbrand interpretation.*
φ is Herbrand satisfiable *iff it is satisfied in some Herbrand interpretation.*
φ is Herbrand unsatisfiable *iff it is falsified in each Herbrand interpretation.*
$\mathcal{I} \models_{Hb} \varphi$ iff $\mathcal{I}$ is an Herbrand interpretation and $\mathcal{I} \models \varphi$.
$\varphi \models_{Hb} \psi$ iff for each Herbrand interpretation $\mathcal{I}$: if $\mathcal{I} \models_{Hb} \varphi$ then $\mathcal{I} \models_{Hb} \psi$.

4.2.1 Herbrand Model Theory vs. Tarski Model Theory. Obviously, each Herbrand satisfiable formula or set of formulas is Tarski satisfiable. The converse does not hold. Assume a signature with a unary relation symbol p and a constant a and no other symbol, such that the Herbrand universe is $HU = \{a\}$. The set $S = \{p(a), \exists x \neg p(x)\}$ is Tarski satisfiable, but Herbrand unsatisfiable. However, S is Herbrand satisfiable with respect to a larger signature containing an additional constant b. Thus, Herbrand satisfiability depends on the signature, whereas Tarski satisfiability does not.

There are two conventions for establishing the signature of a set of formulas: (1) *explicitly* by an *a priori* specification; (2) *implicitly* by gathering the symbols from the formulas considered. Convention (1) is common in mathematical

logic and also underlies the definitions in this survey. Convention (2) is widespread in computational logic. With Tarski model theory, either convention is fine. With Herbrand model theory, convention (2) would not be reasonable: the Herbrand satisfiable set $\{p(a), \neg p(b), \exists x \neg p(x)\}$ would have an Herbrand unsatisfiable subset $\{p(a), \exists x \neg p(x)\}$. In contrast, convention (1) ensures that Herbrand (un)satisfiability translates to subsets/supersets like Tarski (un)satisfiability.

With Tarski model theory, there is no strong correspondence between individuals in the semantic domain and names, i.e., terms as syntactic representations of semantic individuals. Semantic individuals need not have a name (see Examples 58 and 59), and different names may refer to the same semantic individual. With Herbrand model theory, every semantic individual has a name. Moreover, with normal Herbrand interpretations different ground terms represent different individuals, thus incorporating the unique name assumption.

Some momentous results do not copy from Tarski to Herbrand model theory.

Theorem 86. *The following results do* not *hold for Herbrand model theory: The model extension theorem 73. The Löwenheim-Skolem-Tarski theorem 84. The finiteness/compactness theorem 40.*

Proof. The first two are rather immediate. For the last one assume a signature with a unary relation symbol p, a unary function symbol f, a constant a, and no other symbol. Then the Herbrand base is $HB = \{p(a), p(f(a)), p(f(f(a))), \ldots\}$. Although each finite subset of $S = \{\exists x \neg p(x)\} \cup HB$ is Herbrand satisfiable, S is Herbrand unsatisfiable. □

Note that many of the properties of Tarski model theory that appear to be undesirable for query and rule languages depend on one of the results that do not hold for Herbrand model theory. Another property of Tarski model theory is that some theories are not finitely axiomatisable (see Theorem 47). This is so for Herbrand model theory as well.

Proposition 87. *There are Herbrand interpretations $\mathcal{I}$ for which the theory $Th(\{\mathcal{I}\}) = \{\varphi \mid \mathcal{I} \models_{Hb} \varphi\}$ is not finitely axiomatisable.*

Proof. Assume a signature $\mathcal{L}$ with at least one constant, one function symbol of arity > 0, and one relation symbol of arity > 0. Then the Herbrand base $HB_{\mathcal{L}}$ is infinite, thus its powerset is not enumerable. There is a bijection between this set and the set of all Herbrand interpretations for $\mathcal{L}$ (Proposition 63), which is therefore not enumerable either. On the other hand, the set of $\mathcal{L}$-formulas is enumerable, hence the set of its finite subsets is enumerable. There are more Herbrand interpretations than potential finite axiomatisations. □

The price for the simplicity of Herbrand model theory is a loss of semi-decidability of semantic properties. Let us start with an immediate consequence of the "Herbrand Theorem" (Corollary 65).

Proposition 88. *Let S be a set of universal closed formulas. Let φ be a closed formula such that $\neg\varphi$ is universal. Then S is Herbrand satisfiable iff S is Tarski satisfiable and $S \models_{Hb} \varphi$ iff $S \models \varphi$.*

Corollary 89. *Herbrand unsatisfiability and Herbrand entailment are not decidable, but they are semi-decidable for formulas meeting the conditions above.*

Theorem 90 ([88]). *For formulas that do not meet the conditions above, Herbrand unsatisfiability and Herbrand entailment are not semi-decidable.*

Herbrand Model Theory for Finite Herbrand Base. A rather natural restriction for applications is to consider only signatures with a finite number of relation symbols and of constants and without function symbols of arity ≥ 1, as in datalog. This restriction ensures that the Herbrand universe HU and the Herbrand base HB are finite, hence there are only finitely many Herbrand interpretations.

The following results are immediate [88]. See also Theorem 57 and the discussion on propositional vs. ground on page 9.

Proposition 91. *If the Herbrand base HB is finite*

- *Quantification can be transformed into conjunction and disjunction.*
- *The expressive power of the logic is the same as for propositional logic with finitely many propositional relation symbols.*
- *Herbrand satisfiability and Herbrand entailment are decidable.*
- *Every theory is finitely axiomatisable.*

Remark 92. Sets that are Tarski satisfiable with infinite domains only, such as $S = \{\ \forall x \neg(x < x),\quad \forall x \forall y \forall z (x < y \wedge y < z \Rightarrow x < z),\quad \forall x \exists y\ x < y\ \}$, are Herbrand unsatisfiable if the Herbrand universe HU is finite.

4.3 Finite Model Theory

Model theory investigates how syntactic properties of formulas and structural properties of their models relate to each other. Finiteness of the domain is certainly an interesting structural property of interpretations, but it does not have a syntactic counterpart, because finiteness is not expressible in first-order predicate logic (Corollaries 78 and 82). Because of that, finite models were for a long time not a major concern in model theory.

Since the late 1970s, this has changed. Finite model theory gained interest in computer science because of its connections to discrete mathematics, complexity theory, database theory, and to computation in general. Finite interpretations can be encoded as words, i.e., finite sequences of symbols, making computation applicable to model theory. Finite interpretations can describe terminating computations. Databases can be formalised as finite interpretations, such that queries correspond to formulas evaluated against finite interpretations. Complexity classes can be represented as queries (in some logic) that are evaluated against finite interpretations. Because of all these connections, many issues in finite model theory are motivated by computer science.

The purpose of this subsection is to outline salient elementary aspects of "entailment in the finite" and to list results in finite model theory that may be useful for query languages or query evaluation and optimisation methods.

Definition 93. *A* finite interpretation *is an interpretation with finite domain. For formulas or sets of formulas φ and ψ:*

φ is finitely valid *iff it is satisfied in each finite interpretation.*
φ is finitely satisfiable *iff it is satisfied in some finite interpretation.*
φ is finitely unsatisfiable *iff it is falsified in each finite interpretation.*
$\mathcal{I} \models_{fin} \varphi$ iff $\mathcal{I}$ is a finite interpretation and $\mathcal{I} \models \varphi$.
$\varphi \models_{fin} \psi$ iff for each finite interpretation $\mathcal{I}$: if $\mathcal{I} \models_{fin} \varphi$ then $\mathcal{I} \models_{fin} \psi$.

Finite interpretations are special Tarski interpretations, hence there are obvious inclusions between corresponding notions: each valid formula is finitely valid, each finitely satisfiable formula is satisfiable, etc. These inclusions are proper.

For example, $\{\, \forall x\, \neg(x < x),\quad \forall x \forall y \forall z (x < y \,\wedge\, y < z \Rightarrow x < z),\quad \forall x \exists y\, x < y \,\}$ is a satisfiable, but finitely unsatisfiable set of formulas, and the single formula $[\forall x\, \neg(x < x) \wedge \forall x \forall y \forall z (x < y \wedge y < z \Rightarrow x < z)] \Rightarrow \exists x \forall y\, \neg(x < y)$ is finitely valid, but not valid. Strict orderings in finite domains necessarily have maximal elements, but in infinite domains may not have maximal elements.

An example with equality is that any injection of a domain in itself is necessarily surjective in finite domains, but may not be surjective in infinite domains. The formula $\forall x \forall x' (f(x) \doteq f(x') \Rightarrow x \doteq x') \wedge \neg \forall y \exists x\, y \doteq f(x)$ has normal models, but no finite normal models.

The model relationship (Definition 32) is defined by a recursive algorithm for evaluating a formula in an interpretation. There are several reasons why this algorithm may not terminate for infinite interpretations, such as the quantifier cases or potentially undecidable relations and incomputable functions over the domain. None of these reasons applies to finite domains. Thus we have

Theorem 94. *The theory $Th(\{\mathcal{I}\}) = \{\varphi \mid \mathcal{I} \models_{fin} \varphi\}$ is decidable for each finite interpretation $\mathcal{I}$.*

Definition 95 (Isomorphic interpretations). *Let $\mathcal{L}$ be a signature for first-order predicate logic and let $\mathcal{I}$ and $\mathcal{J}$ be $\mathcal{L}$-interpretations. An* isomorphism of $\mathcal{I}$ into $\mathcal{J}$ *is a function $\pi : dom(\mathcal{I}) \to dom(\mathcal{J})$ such that*

- *$\pi : dom(\mathcal{I}) \to dom(\mathcal{J})$ is a bijection.*
- *$\pi(c^{\mathcal{I}}) = c^{\mathcal{J}}$ for each constant c of $\mathcal{L}$.*
- *$\pi(f^{\mathcal{I}}(d_1, \ldots, d_n)) = f^{\mathcal{J}}(\pi(d_1), \ldots, \pi(d_n))$ for each n-ary ($n \geq 1$) function symbol f of $\mathcal{L}$ and all $d_1, \ldots, d_n \in dom(\mathcal{I})$.*
- *$p^{\mathcal{I}} = p^{\mathcal{J}}$ for each propositional relation symbol of $\mathcal{L}$*
- *$(d_1, \ldots, d_n) \in p^{\mathcal{I}}$ iff $(\pi(d_1), \ldots, \pi(d_n)) \in p^{\mathcal{J}}$ for each n-ary ($n \geq 1$) relation symbol p of $\mathcal{L}$ and all $d_1, \ldots, d_n \in dom(\mathcal{I})$.*

$\mathcal{I} \cong \mathcal{J}$, pronounced $\mathcal{I}$ and $\mathcal{J}$ are isomorphic*, iff such an isomorphism exists.*

Note that the variable assignments of isomorphic interpretations need not be compatible.

Proposition 96. *If $\mathcal{I} \cong \mathcal{J}$, then for each closed formula φ: $\mathcal{I} \models \varphi$ iff $\mathcal{J} \models \varphi$.*

If π is an isomorphism of $\mathcal{I}$ into $\mathcal{J}$ and V is a variable assignment in $dom(\mathcal{I})$, then $\pi \circ V$ is a variable assignment in $dom(\mathcal{J})$ and for each formula φ:
$\mathcal{I}[V] \models \varphi$ iff $\mathcal{J}[\pi \circ V] \models \varphi$.

Definition 97. *For $k \in \mathbb{N}$ let $\mathbb{N}_k = \{0, \ldots, k\}$ denote the initial segment of $\mathbb{N}$ with cardinality $k+1$. Let V_0 be the variable assignment in $\mathbb{N}_k$ for arbitrary k with $x^{V_0} = 0$ for each variable x.*

Proposition 98. *Each finite interpretation is isomorphic to an interpretation with domain $\mathbb{N}_k$ for some $k \in \mathbb{N}$.*

Proposition 99. *For finite signatures there are for each $k \in \mathbb{N}$ only finitely many interpretations with domain $\mathbb{N}_k$ and variable assignment V_0.*

Proof. For each $k \in \mathbb{N}$ and each $n \in \mathbb{N}$ the set $\mathbb{N}_k^n$ contains $(k+1)^n$ tuples. Thus there are $2^{((k+1)^n)}$ possible relations and $(k+1)^{((k+1)^n)}$ possible functions of arity n, to which the finitely many signature symbols can be mapped by an interpretation. □

Corollary 100. *For finite signatures, the problem whether a finite set of closed formulas has a model with a given finite cardinality, is decidable.*

This makes possible a simple semi-decision procedure for finite satisfiability: iterating k from 1 upwards, check whether the given formulas have a model with cardinality k.

Corollary 101. *For finite signatures, finite satisfiability, finite falsifiability, and finite non-entailment of finite sets of closed formulas are semi-decidable.*

Theorem 102 (Trakhtenbrot). *For signatures with a non-propositional relation symbol and a relation or function symbol of arity ≥ 2, finite validity is not semi-decidable.*

Corollary 103. *For finite signatures with a non-propositional relation symbol and a relation or function symbol of arity ≥ 2, finite satisfiability, finite falsifiability, and finite non-entailment of finite sets of closed formulas are semi-decidable, but not decidable. Finite unsatisfiability, finite validity, and finite entailment are not semi-decidable,*

This is a remarkable reversal of results in the two model theories: (classical) Tarksi unsatisfiability is semi-decidable and Tarski satisfiability is not, whereas finite satisfiability is semi-decidable and finite unsatisfiability is not.

Corollary 104. *There is no complete calculus for finite entailment.*

Theorem 105. *The finiteness/compactness theorem does not hold for finite model theory.*

Proof. For each $n \in \mathbb{N}$ let φ_n be a finitely satisfiable formula all of whose models have domains with cardinality $\geq n+1$. Such formulas exist, see the examples to Theorem 75. Then each finite subset of $S = \{\varphi_n \mid n \in \mathbb{N}\}$ is finitely satisfiable, but S is not finitely satisfiable. □

Another difference to Tarski model theory is that finite interpretations can be characterised up to isomorphism by formulas.

Theorem 106. *For each finite interpretation $\mathcal{I}$ there exists a set $S_{\mathcal{I}}$ of closed formulas such that for each interpretation $\mathcal{J}$: $\mathcal{J} \cong \mathcal{I}$ iff $\mathcal{J} \models S_{\mathcal{I}}$.*

If the signature is finite, there is a single closed formula rather than a set of closed formulas with this property.

This does not extend to infinite interpretations, among other reasons, because there are "not enough" formulas. An example for an interpretation that cannot be characterised up to isomorphism is the standard model of arithmetic, whose domain is the set of natural numbers. If any would-be axiomatisation is satisfied by the standard model, it also has non-standard models that are not isomorphic to the standard model. In order to characterise the standard model up to isomorphism, an axiomatisation would have to include the induction axiom (or equivalent), which is not expressible in first-order predicate logic.

4.3.1 Finitely Controllable Formulas.

Definition 107. *A closed formula is* finitely controllable*, if it is either unsatisfiable or finitely satisfiable. A fragment of first-order predicate logic is finitely controllable, if each closed formula belonging to that fragment is.*

For finitely controllable formulas, satisfiability coincides with finite satisfiability. Note that a satisfiable finitely controllable formula may well have infinite models, but it is guaranteed to have also a finite model.

Theorem 108. *For finite signatures, satisfiability of a finitely controllable closed formula is decidable.*

Proof. By intertwining a semi-decision procedure for Tarski unsatisfiability with a semi-decision procedure for finite satisfiability. □

Because of this, finite controllability is one of the major techniques for proving that satisfiability in some fragment of first-order predicate logic is decidable. If one can show for that fragment that the existence of a model implies the existence of a finite model, decidability follows from the theorem above.

Theorem 109. *The fragments of first-order predicate logic consisting of closed formulas in prenex form with a quantifier prefix of one of the following forms are finitely controllable. The notation Q^* means that there may be zero or more consecutive occurrences of this quantifier.*

- $\exists^*\forall^*$ *possibly with equality* *(Bernays-Schönfinkel prefix class)*
- $\exists^*\forall\exists^*$ *possibly with equality* *(Ackermann prefix class)*
- $\exists^*\forall\forall\exists^*$ *without equality* *(Gödel prefix class)*

As can be seen from their names, these decidable fragments were identified long before computer science existed. A more recent fragment, which is highly relevant to query languages, is the following.

Definition 110 (FO^2). *The fragment of first-order predicate logic with equality whose formulas contain no variables other than x and y (free or bound) is called* two-variable first-order predicate logic *or FO^2.*

Theorem 111. *FO^2 is finitely controllable.*

As an illustration of the expressive power of this fragment, consider a directed graph. It is possible to express in FO^2 queries such as "does a given node a have an ingoing edge?" or "is every node with property p reachable from a via at most two edges?" More generally, FO^2 can express queries for properties that can be checked by a successive analysis of pairs of elements in the domain. It cannot express queries for global properties that require a simultaneous analysis of larger parts of the domain, such as "is there a cycle in the graph?" In particular, transitivity of a binary relation cannot be expressed.

Many logics that are of interest to knowledge representation, for instance several modal and description logics, can be seen as two-variable logics embedded in suitable extensions of FO^2. See [81] for an overview.

4.3.2 0-1 Laws. The following information is taken from an article by Ronald Fagin entitled "Finite-Model Theory – a Personal Perspective"[66].

As a motivating example, let us assume a signature with 2-ary function symbols $+$ and $\times$ and the set of formulas axiomatising a field.[13] Let φ be the conjunction of these (finitely many) axioms. The negation $\neg\varphi$ of this formula would seem to be very uninteresting, because, intuitively, if we consider arbitrary theories, it is highly unlikely that the theory happens to be a field, which means that $\neg\varphi$ is "almost surely true" in such theories.

In order to make this intuition more precise, imagine that a finite interpretation with cardinality n is randomly generated, such that the probability for a ground atom to be true in this interpretation is $\frac{1}{2}$ for each ground atom independently of the truth values of other ground atoms. For formulas φ let $P_n(\varphi)$ denote the probability that such a randomly generated interpretation satisfies φ.

Definition 112. *A formula φ is* almost surely true *iff $\lim_{n\to\infty} P_n(\varphi) = 1$, i.e., with increasing cardinality of interpretations the asymptotic probability for φ to be true is* 1*.*

Theorem 113 (0-1 law). *For each closed formula φ, either φ or $\neg\varphi$ is almost surely true.*

Quoting Fagin [66], who cites Vardi:

> There are three possibilities for a closed formula: it can be surely true (valid), it can be surely false (unsatisfiable), or it can be neither. The

[13] A field is an algebraic structure where both operations have an inverse. The set of rational numbers $\mathbb{Q}$ with addition and multiplication and the set of real numbers $\mathbb{R}$ with addition and multiplication are examples of fields.

> third possibility (where a closed formula is neither valid nor unsatisfiable) is the common case. When we consider asymptotic probabilities, there are *a priori* three possibilities: it can be almost surely true (asymptotic probability 1), it can be almost surely false (asymptotic probability 0), or it can be neither (either because there is no limit, or because the limit exists and is not 0 or 1). Again, we might expect the third possibility to be the common case. The 0-1 law says that the third possibility is not only not the common case, but it is, in fact, impossible!

This result may seem like an amusing oddity, but it is a significant tool for proving that certain properties of finite interpretations cannot be expressed in first-order predicate logic.

For instance if there were a closed formula φ_{even} that is satisfied by a finite interpretation iff the domain of this interpretation has even cardinality, then $P_n(\varphi_{even})$ would be 1 for even n and 0 for odd n, hence there would be no limit. By the 0-1 law, this is impossible, therefore such a formula φ_{even} does not exist. Hence, evenness is not expressible in first-order predicate logic.

The same kind of argument based on the 0-1 law can be used to show for many properties of finite interpretations that they are not expressible in first-order predicate logic. Often, such properties involve some form of "counting". It is part of the folklore knowledge about first-order predicate logic that it "cannot count".

There are several theorems similar to the 0-1 law. A research issue in this area is to determine (fragments of) other logics that admit the 0-1 law. For example, see [104].

5 Declarative Semantics: Adapting Classical Model Theory to Rule Languages

The declarative semantics of definite programs can be defined in model-theoretic terms using specific properties of syntactical classes of formulas that are more general than definite clauses.

Definite clauses (Definition 21) are also called *definite rules*. Some authors call them *positive definite rules* when they want to emphasise that they exclude normal clauses, i.e., clauses with negative literals in their antecedent. In line with this terminology, a definite program is also called a set of positive definite rules or simply a *positive rule set*. This terminology will also be used below.

In this section, a signature $\mathcal{L}$ for first-order predicate logic is assumed. Unless otherwise stated, semantic notions such as (un)satisfiability or logical equivalence refer to unrestricted interpretations, i.e., interpretations with domains of any kind, especially of any cardinality.

5.1 Minimal Model Semantics of Definite Rules

This subsection is inspired by [144].

As discussed in Subsection 3.2, a rule is a shorthand notation for its universal closure. Thus, a positive definite rule is on the one hand a special *universal* formula (defined in Subsection 3.1). On the other hand, it is also a special *inductive* formula (defined below).

Both classes of formulas have interesting model-theoretic properties. If a set of universal formulas is satisfiable, then it is Herbrand satisfiable, i.e., it has an Herbrand model. If a set of inductive formulas is satisfiable, then the intersection of its models is also a model, provided that the models intersected are compatible. Moreover, each set of definite inductive formulas is satisfiable.

Together this means that each set of positive definite rules has a unique minimal Herbrand model, which is the intersection of all Herbrand models of the set. This minimal model can be taken as "the meaning" of the set of positive definite rules in a model-theoretic sense.

5.1.1 Compatibility and Intersection of Interpretations.

Definition 114 (Compatible set of interpretations). *A set $\{\mathcal{I}_i \mid i \in I\}$ of interpretations with index set I is called* compatible*, iff*

- $I \neq \emptyset$.
- $D = \bigcap\{dom(\mathcal{I}_i) \mid i \in I\} \neq \emptyset$.
- *all interpretations of a function symbol coincide on the common domain:* $f^{\mathcal{I}_i}(d_1, \ldots, d_n) = f^{\mathcal{I}_j}(d_1, \ldots, d_n)$ *for each n-ary ($n \geq 0$) function symbol f, for all $i, j \in I$, and for all $d_1, \ldots, d_n \in D$.*
- *a variable is identically interpreted in all interpretations:* $x^{\mathcal{I}_i} = x^{\mathcal{I}_j}$ *for each variable x and all $i, j \in I$.*

Definition 115 (Intersection of a compatible set of interpretations). *Let $\{\mathcal{I}_i \mid i \in I\}$ be a compatible set of interpretations. Then $\bigcap\{\mathcal{I}_i \mid i \in I\}$ is defined as the interpretation $\mathcal{I}$ with*

- $dom(\mathcal{I}) = D = \bigcap\{dom(\mathcal{I}_i) \mid i \in I\}$.
- *a function symbol is interpreted as the intersection of its interpretations:* $f^{\mathcal{I}}(d_1, \ldots, d_n) = f^{\mathcal{I}_i}(d_1, \ldots, d_n)$ *for each n-ary ($n \geq 0$) function symbol f, for an arbitrary $i \in I$, and for all $d_1, \ldots, d_n \in D$.*
- *a relation symbol is interpreted as the intersection of its interpretations:* $p^{\mathcal{I}} = \bigcap_{i \in I} p^{\mathcal{I}_i}$ *for each relation symbol p.*
- *a variable is interpreted like in all given interpretations:* $x^{\mathcal{I}} = x^{\mathcal{I}_i}$ *for each variable x and an arbitrary $i \in I$.*

5.1.2 Universal Formulas and Theories. Let us recall some notions and results from previous sections.

The *polarity* of a subformula within a formula (Definition 13) is positive, if the subformula occurs within the scope of an even number of explicit or implicit negations, and negative, if this number is odd. A formula is *universal* (Definition 14), if all its occurrences of $\forall$ have positive and of $\exists$ have negative polarity.

This is the case iff the formula is equivalent to a formula in prenex form containing no existential quantifier (Theorem 37). A *universal theory* is a theory axiomatised by a set of universal formulas.

The Herbrand universe HU is the set of ground terms, and the Herbrand base HB is the set of ground atoms of the given signature (Definition 55). An Herbrand interpretation interprets each ground term with itself (Definition 56). There is a bijection between the set of all Herbrand interpretations and the set of all subsets of HB (Proposition 63). Each subset $B \subseteq HB$ of ground atoms induces the Herbrand interpretation $HI(B)$ that satisfies all ground atoms in B and falsifies all ground atoms not in B.

If a set S of universal formulas has a model $\mathcal{I}$, then the Herbrand interpretation $HI(\mathcal{I})$ induced by $\mathcal{I}$ is also a model of S (Theorem 64). Thus, S is satisfiable iff it has an Herbrand model.

Note that this is easily disproved for non-universal formulas. For example, if the signature consists of a unary relation symbol p and a constant a and no other symbols, then $HB = \{p(a)\}$ and there are exactly two Herbrand interpretations: $HI(\emptyset)$ and $HI(\{p(a)\})$. The formula $p(a) \wedge \exists x \neg p(x)$ is satisfiable, but neither of the two Herbrand interpretations is a model of it. See also Examples 58 and 59 in Section 4.1.

For sets of universal formulas in general, and for sets of positive definite rules in particular, satisfiability coincides with Herbrand satisfiability. This is interesting for two reasons. First, the domain of Herbrand interpretations is (computably) enumerable. Second, Herbrand interpretations are syntactically defined. Being enumerable and syntactically defined, Herbrand interpretations are amenable to computing.

Returning to the notions introduced above, the following result is obtained by straightforward application of the definitions:

Lemma 116. *Let $\{B_i \mid i \in I\}$ be a set of sets of ground atoms, i.e., $B_i \subseteq HB$ for each $i \in I$. If this set is nonempty, then*

- *$\{HI(B_i) \mid i \in I\}$ is a compatible set of interpretations, i.e., its intersection is defined.*
- *$\bigcap\{HI(B_i) \mid i \in I\} = HI\big(\bigcap\{B_i \mid i \in I\}\big)$ i.e., its intersection is the Herbrand interpretation induced by the intersection of the sets of ground atoms.*

Definition 117 (Set of inducers of Herbrand models of a set of formulas). *For a set S of formulas, the set of inducers of its Herbrand models is $Mod_{HB}(S) = \{B \subseteq HB \mid HI(B) \models S\}$.*

Note that although the class $Mod(S)$ of all models of S is not in general a set (Definition 42), the class $Mod_{HB}(S)$ of all inducers of Herbrand models of S is a set: a subset of the powerset of the Herbrand base HB.

Obviously, $Mod_{HB}(S) = \emptyset$ for unsatisfiable S. If S is satisfiable and non-universal, $Mod_{HB}(S)$ may or may not be empty. If S is satisfiable and universal, $Mod_{HB}(S) \neq \emptyset$ and the intersection of the set of its Herbrand models is defined. We introduce a notation for the intersection that is always defined:

Notation 118. *For a set S of formulas:*

$$Mod_{\cap}(S) = \begin{cases} \bigcap Mod_{HB}(S) & \text{if } Mod_{HB}(S) \neq \emptyset \\ HB & \text{if } Mod_{HB}(S) = \emptyset \end{cases}$$

Theorem 119. *If S is universal, then $Mod_{\cap}(S) = \{A \in HB \mid S \models A\}$.*

Proof. If S is unsatisfiable, both sides are equal to HB. If S is satisfiable:

"$\subseteq$": Let $A \in Mod_{\cap}(S)$, thus $A \in B$ for each $B \subseteq HB$ with $HI(B) \models S$. To be shown: $S \models A$. Let $\mathcal{I}$ be an arbitrary model of S. By Theorem 64, $HI(B') \models S$ where $B' = \{A' \in HB \mid \mathcal{I} \models A'\}$. Hence by the first sentence, $A \in B'$, therefore $\mathcal{I} \models A$. Since $\mathcal{I}$ was arbitrary, we have shown $S \models A$.

"$\supseteq$": Let $A \in HB$ with $S \models A$, i.e., each model of S satisfies A. Then for each $B \subseteq HB$ with $HI(B) \models S$ holds $HI(B) \models A$ and therefore $A \in B$. Hence $A \in Mod_{\cap}(S)$. □

The definition guarantees that $HI(Mod_{\cap}(S))$ is always an Herbrand interpretation. If S is universal and unsatisfiable, then $HI(Mod_{\cap}(S))$ satisfies all ground atoms, but is obviously not a model of S. If S is universal and satisfiable, then $HI(Mod_{\cap}(S))$ is the intersection of all Herbrand models of S. It is worth noting that in this case $HI(Mod_{\cap}(S))$ is not necessarily a model of S. The reason is that some formulas in S may be "indefinite":

Example 120. Assume a signature consisting of a unary relation symbol p and constants a and b and no other symbols. Let $S = \{p(a) \vee p(b)\}$. Then $Mod_{HB}(S) = \{\ \{p(a)\},\ \{p(b)\},\ \{p(a), p(b)\}\ \}$. But $HI(Mod_{\cap}(S)) = HI(\emptyset)$ is not a model of S.

Non-closedness of Herbrand models under intersection is possible for sets of general universal formulas, but, as we shall see, not for sets of positive definite rules. Before coming to that, let us take a look at another property of sets of universal formulas, which is one if their most significant characteristics.

Definition 121 (Subinterpretation). *An interpretation $\mathcal{I}_1$ is a* subinterpretation *of an interpretation $\mathcal{I}_2$, denoted $\mathcal{I}_1 \subseteq \mathcal{I}_2$, if*

- *$dom(\mathcal{I}_1) \subseteq dom(\mathcal{I}_2)$.*
- *the interpretations of a function symbol coincide on the common domain: $f^{\mathcal{I}_1}(d_1, \ldots, d_n) = f^{\mathcal{I}_2}(d_1, \ldots, d_n)$ for each n-ary ($n \geq 0$) function symbol f and all $d_1, \ldots, d_n \in dom(\mathcal{I}_1)$.*
- *the interpretations of a relation symbol coincide on the common domain: $p^{\mathcal{I}_1} = p^{\mathcal{I}_2} \cap dom(\mathcal{I}_1)^n$ for each n-ary ($n \geq 0$) relation symbol p.*
- *a variable is identically interpreted in the interpretations: $x^{\mathcal{I}_1} = x^{\mathcal{I}_2}$ for each variable x.*

If in addition $dom(\mathcal{I}_1) \neq dom(\mathcal{I}_2)$, then $\mathcal{I}_1$ is a proper subinterpretation *of $\mathcal{I}_2$.*

The subinterpretation relationship is a partial ordering on interpretations. Given a set of compatible interpretations where all interpretations of a relation symbol coincide on the common domain, its intersection is a subinterpretation of each interpretation in the set.

Lemma 122. *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be interpretations with $\mathcal{I}_1 \subseteq \mathcal{I}_2$. Let V be an arbitrary variable assignment in $dom(\mathcal{I}_1)$. Let φ be a quantifier-free formula. Then $\mathcal{I}_1[V] \models \varphi$ iff $\mathcal{I}_2[V] \models \varphi$.*

Proof. By structural induction on φ. □

Theorem 123 (Subinterpretation property of universal formulas). *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be interpretations with $\mathcal{I}_1 \subseteq \mathcal{I}_2$. For each universal closed formula φ, if $\mathcal{I}_2 \models \varphi$ then $\mathcal{I}_1 \models \varphi$.*

Proof. By considering a prenex form of φ and applying the previous lemma. □

As an illustration, consider a signature with 2-ary function symbols + and × and the equality relation symbol $\doteq$. Let $dom(\mathcal{I}_1) = \mathbb{Q}$, the set of rational numbers, and $dom(\mathcal{I}_2) = \mathbb{R}$, the set of real numbers, and let + and × be interpreted as addition and multiplication on the respective domain. Then $\mathcal{I}_1 \subseteq \mathcal{I}_2$. The formula $\forall x \forall y\ (x+x) \times y \doteq x \times (y+y)$ is true in $\mathcal{I}_2$ (the reals) and, being universal, it is also true in $\mathcal{I}_1$ (the rationals). The non-universal formula $\forall y \exists x\ y \doteq x \times x \times x$ is true in $\mathcal{I}_2$ (the reals), but not true in $\mathcal{I}_1$ (the rationals).

An immediate consequence of the theorem is that if a set of closed formulas is universal, then all subinterpretations of its models are models. A famous result by Łos and Tarski establishes the converse: If a set of closed formulas is satisfied by all subinterpretations of its models, then it is equivalent to a set of universal closed formulas. Thus, the subinterpretation property is a semantic characterisation of the syntactic class of universal formulas.

5.1.3 Inductive Formulas and Theories.

Definition 124 (Positive and negative formulas). *A formula φ is called* positive *(or* negative*, respectively) iff every atom occurring in φ has positive (or negative, respectively) polarity in φ.*

Definition 125 (Inductive formula). *A* generalised definite rule *is a formula of the form $\forall^*((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi)$ where φ is positive and the A_i are atoms for $1 \leq i \leq n$. It is also called a* definite inductive formula*.*

A generalised definite goal *is a formula of the form $\forall^* \varphi$ where φ is negative. It is also called an* integrity constraint*.*

An inductive formula *is either a generalised definite rule or a generalised definite goal. A* (definite) inductive theory *is a theory axiomatised by a set of (definite) inductive formulas.*

Recall that $\forall^*$ denotes the universal closure. The point of a generalised definite rule is that its only positive atoms are conjunctively connected and that all variables occurring in this conjunction are universally quantified. A generalised definite goal is logically equivalent to a formula $\forall^* \neg \varphi$ and thus to $\forall^*(\bot \leftarrow \varphi)$ with positive φ, which shows the similarity to a generalised definite rule.

Each inductive formula is equivalent to a formula in prenex form whose matrix is a conjunction of Horn clauses (Definition 21) and whose quantifier prefix

starts with universal quantifiers for all variables in the consequents followed by arbitrary quantifiers for the remaining variables. For a generalised definite rule, all Horn clauses in the matrix are definite clauses. For a generalised definite goal, all Horn clauses in the matrix are definite goals. It would make sense to call inductive formulas "generalised Horn clauses" (compare Definition 21).

Let us now introduce a partial ordering on interpretations that differs slightly from the subinterpretation relationship:

Definition 126. *$\mathcal{I}_1 \leq \mathcal{I}_2$ for interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ if*

- *$dom(\mathcal{I}_1) = dom(\mathcal{I}_2)$.*
- *the interpretations of a function symbol coincide on the common domain: $f^{\mathcal{I}_1}(d_1, \ldots, d_n) = f^{\mathcal{I}_2}(d_1, \ldots, d_n)$ for each n-ary ($n \geq 0$) function symbol f and all $d_1, \ldots, d_n \in dom(\mathcal{I}_1)$.*
- *the "smaller" interpretation of a relation symbol is a restriction of the other: $p^{\mathcal{I}_1} \subseteq p^{\mathcal{I}_2}$ for each n-ary ($n \geq 0$) relation symbol p.*
- *a variable is identically interpreted in the interpretations: $x^{\mathcal{I}_1} = x^{\mathcal{I}_2}$ for each variable x*

If in addition $p^{\mathcal{I}_1} \neq p^{\mathcal{I}_2}$ for at least one p, then $\mathcal{I}_1 < \mathcal{I}_2$.

In contrast to the subinterpretation relationship, here the domains of the interpretations are the same. For subinterpretations this would imply that the interpretations of a relation symbol coincide, here the "smaller" one may be a restriction of the other. Given a set of compatible interpretations with the same domain, its intersection is $\leq$ each interpretation in the set.

Lemma 127. *Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be interpretations with $\mathcal{I}_1 \leq \mathcal{I}_2$. Let V be an arbitrary variable assignment in $dom(\mathcal{I}_1)$.*

If φ is a positive formula: if $\mathcal{I}_1[V] \models \varphi$ then $\mathcal{I}_2[V] \models \varphi$
If φ is a negative formula: if $\mathcal{I}_2[V] \models \varphi$ then $\mathcal{I}_1[V] \models \varphi$

Proof. By structural induction on the matrix of a prenex form of φ. □

An interesting property of sets of generalised definite rules is that they are satisfiable. In particular, the $\leq$-largest Herbrand interpretation, which satisfies all ground atoms, is always a model.

Theorem 128. *For each set S of generalised definite rules, $HI(HB) \models S$.*

Proof. Let S be a set of generalised definite rules, thus its members have the form $\forall^*[(A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi]$ where φ is positive and the A_i are atoms.

Each member of S is logically equivalent to $\forall \boldsymbol{x}[(A_1 \wedge \ldots \wedge A_n) \leftarrow \exists \boldsymbol{y}\varphi]$ where $\boldsymbol{x}$ are the variables occurring in $A_1 \ldots A_n$ and $\boldsymbol{y}$ are the other free variables of φ. It suffices to show that $HI(HB)$ satisfies each instance $[(A_1 \wedge \ldots \wedge A_n) \leftarrow \exists \boldsymbol{y}\varphi]\sigma$ where σ is a ground substitution with domain $\boldsymbol{x}$ (Theorem 57). Each of these instances is $[(A_1\sigma \wedge \ldots \wedge A_n\sigma) \leftarrow (\exists \boldsymbol{y}\varphi)\sigma]$ where the $A_i\sigma$ are ground atoms.

Since $HI(HB)$ satisfies all ground atoms, it satisfies each of these instances and thus each member of S. □

The main result about inductive formulas is that their (compatible) models are closed under intersection.

Theorem 129. *Let S be a set of inductive formulas. If $\{\mathcal{I}_i \mid i \in I\}$ is a set of compatible models of S with the same domain D, then $\mathcal{I} = \bigcap\{\mathcal{I}_i \mid i \in I\}$ is also a model of S.*

Proof. Let V be an arbitrary variable assignment in D. By definition of the partial ordering $\leq$ on interpretations, $\mathcal{I}[V] \leq \mathcal{I}_i[V]$ for each $i \in I$.

Let $\forall^*((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi)$ with positive φ be a generalised definite rule in S. If $\mathcal{I}[V] \not\models \varphi$, then $\mathcal{I}[V] \models ((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi)$. If $\mathcal{I}[V] \models \varphi$, then $\mathcal{I}_i[V] \models \varphi$ for each $i \in I$ by Lemma 127. Therefore, since each $\mathcal{I}_i$ satisfies each member of S, $\mathcal{I}_i[V] \models A_j$ for each $i \in I$ and $1 \leq j \leq n$. By Definition 115, $\mathcal{I}[V] \models A_j$ for each j with $1 \leq j \leq n$. Thus $\mathcal{I}[V] \models (A_1 \wedge \ldots \wedge A_n)$ and $\mathcal{I}[V] \models ((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi)$. In both cases, since V is arbitrary, $\mathcal{I}$ satisfies the considered member of S.

Let $\forall^*\varphi$ with negative φ be a generalised definite goal in S. Then $\mathcal{I}_i[V] \models \varphi$ for each $i \in I$, because each $\mathcal{I}_i$ satisfies each member of S. By Lemma 127, $\mathcal{I}[V] \models \varphi$. Since V is arbitrary, $\mathcal{I}$ satisfies the considered member of S. □

Corollary 130. *If S is a set of inductive formulas and $\{B_i \subseteq HB \mid i \in I\}$ is a nonempty set with $HI(B_i) \models S$ for each $i \in I$, then $HI\big(\bigcap\{B_i \mid i \in I\}\big) \models S$.*

5.1.4 Minimal Models.

Definition 131 (Minimal model). *A* minimal model *of a set of formulas is a $\leq$-minimal member $\mathcal{I}$ of the set of all its models with domain $dom(\mathcal{I})$.*

The partial ordering $\leq$ on interpretations corresponds to the subset relationship $\subseteq$ on sets of n-tuples of the domain with which relation symbols are interpreted. A model is minimal, if there is no other model with the same domain that interprets some relation symbol with a proper subset of n-tuples of the domain. Note that the subset relationship refers to sets that are not syntactic.

For Herbrand interpretations the partial ordering $\leq$ on interpretations corresponds to the subset relationship on their inducers, which are syntactic sets.

Lemma 132. *Let S be a set of formulas.*

- *An Herbrand model of S is minimal iff it is induced by a $\subseteq$-minimal member of $Mod_{HB}(S)$.*
- *If $HI(Mod_\cap(S))$ is a model of S, it is a minimal Herbrand model of S and it is the only minimal Herbrand model of S.*

An Herbrand model $HI(B)$ of S is minimal iff there is no proper subset $B' \subset B$ such that $HI(B')$ is also a model of S. Reconsidering $S = \{p(a) \vee p(b)\}$ from Example 120 above: both $HI(\{p(a)\})$ and $HI(\{p(b)\})$ are minimal Herbrand models of S, and $HI(\{p(a), p(b)\})$ is a non-minimal Herbrand model of S.

Theorem 133. *Let S be a set of inductive formulas. If either each member of S is definite, or S is satisfiable and each member of S is universal, then $HI(Mod_\cap(S))$ is the unique minimal Herbrand model of S.*

Proof. $Mod_{HB}(S) \neq \emptyset$ in the first case by Theorem 128, in the second case by Theorem 64. By Corollary 130, $HI(Mod_{\cap}(S))$ is a model of S, and by the previous lemma it is the unique minimal Herbrand model of S. □

Noting that positive definite rules (i.e., definite clauses) are both universal and definite inductive formulas, and taking into account Theorem 119, we obtain:

Corollary 134 (Minimal Herbrand Model of a Definite Program). *Each set S of positive definite rules (i.e., each definite program) has a unique minimal Herbrand model. This model is the intersection of all Herbrand models of S. It satisfies precisely those ground atoms that are logical consequences of S.*

This unique minimal model of a set of positive definite rules can be regarded as its natural "meaning".

The notion of minimal model is also defined for non-Herbrand interpretations and therefore also applies to more general classes of formulas than inductive formulas. Typically, for such more general classes of formulas both the uniqueness and the closedness under intersection of minimal models are lost.

Let us now consider a generalisation of inductive formulas for which the notion of minimal models nevertheless retains a useful characterisation.

Definition 135 (Generalised rule). *A* generalised rule *is a formula of the form $\forall^*(\psi \leftarrow \varphi)$ where φ is positive and ψ is positive and quantifier-free.*

Among others, disjunctive clauses (Definition 21) are generalised rules. The generalised rule $(p(a) \vee p(b) \leftarrow \top)$ is equivalent to the formula from Example 120, which has two minimal Herbrand models.

Note that generalised rules, like generalised definite rules, are not necessarily universal, because their antecedent may contain quantifiers of both kinds.

Definition 136 (Implicant of a positive quantifier-free formula). *Let ψ be a positive quantifier-free formula. The set $primps(\psi)$ of* pre-implicants *of ψ is defined as follows depending on the form of ψ:*

- $primps(\psi) = \{\, \{\psi\} \,\}$ *if ψ is an atom or $\top$ or $\bot$.*
- $primps(\neg\psi_1) = primps(\psi_1)$.
- $primps(\psi_1 \wedge \psi_2) = \{\, C_1 \cup C_2 \mid C_1 \in primps(\psi_1),\ C_2 \in primps(\psi_2) \,\}$.
- $primps(\psi_1 \vee \psi_2) = primps(\psi_1 \Rightarrow \psi_2) = primps(\psi_1) \cup primps(\psi_2)$.

The set of implicants of ψ *is obtained from $primps(\psi)$ by removing all sets containing $\bot$ and by removing $\top$ from the remaining sets.*

Note that each implicant of a positive quantifier-free formula is a finite set of atoms and that the set of implicants is finite. Forming a conjunction of the atoms in an implicant and a disjunction of all of these conjunctions, results in a *disjunctive normal form*, which is equivalent to the original formula. If ψ is a conjunction of atoms (like the consequent of a generalised definite rule), then it has exactly one implicant, which consists of all of these atoms. If ψ is a disjunction of atoms, then each of its implicants is a singleton set consisting of one of these atoms. Taking into account that the definition of implicants applies to positive and quantifier-free formulas only, the following result is straightforward.

Lemma 137.

1. *If C is an implicant of ψ, then $C \models \psi$.*
2. *For any interpretation $\mathcal{I}$, if $\mathcal{I} \models \psi$ then there exists an implicant C of ψ with $\mathcal{I} \models C$.*

Definition 138 (Supported atom). *Let $\mathcal{I}$ be an interpretation, V a variable assignment in $dom(\mathcal{I})$ and $A = p(t_1, \ldots, t_n)$ an atom, $n \geq 0$.*

- *an atom B* supports *A in $I[V]$ iff*
  $I[V] \models B$ and $B = p(s_1, \ldots, s_n)$ and $s_i^{I[V]} = t_i^{I[V]}$ for $1 \leq i \leq n$.
- *a set C of atoms* supports *A in $I[V]$ iff*
  $I[V] \models C$ and there is an atom in C that supports A in $I[V]$.
- *a generalised rule $\forall^*(\psi \leftarrow \varphi)$* supports *$A$ in I iff for each variable assignment V with $I[V] \models \varphi$ there is an implicant C of ψ that supports A in $I[V]$.*

The idea of an atom being supported is that some atom with the same relation symbol and identically interpreted term list occurs in one of the parts of the consequent of the generalised rule that have to be true when the antecedent is true. It turns out that in minimal models only those atoms are true that have to be true in this sense.

Theorem 139 (Minimal models satisfy only supported ground atoms). *Let S be a set of generalised rules. Let $\mathcal{I}$ be an interpretation with domain D. If $\mathcal{I}$ is a minimal model of S, then: For each ground atom A with $\mathcal{I} \models A$ there is a generalised rule in S that supports A in $\mathcal{I}$.*

Proof. Assume that $\mathcal{I}$ is a minimal model of S and there is a ground atom A with $\mathcal{I} \models A$, such that A is not supported in $\mathcal{I}$ by any generalised rule in S.

Let $\mathcal{I}'$ be identical to $\mathcal{I}$ except that $\mathcal{I}' \not\models A$ (by removing just one tuple from the relation $p^{\mathcal{I}}$ for the relation symbol p of A). Then $\mathcal{I}' < \mathcal{I}$.

Consider an arbitrary member $\forall^*(\psi \leftarrow \varphi)$ of S. By assumption it does not support A. Let V be an arbitrary variable assignment in D. We show that $\mathcal{I}'[V] \models (\psi \leftarrow \varphi)$.

If $\mathcal{I}[V] \not\models \varphi$, by Lemma 127 also $\mathcal{I}'[V] \not\models \varphi$, hence $\mathcal{I}'[V] \models (\psi \leftarrow \varphi)$.

If $\mathcal{I}[V] \models \varphi$, then $\mathcal{I}[V] \models \psi$ because $\mathcal{I}$ is a model of S. Furthermore, by assumption for each implicant C of ψ either $\mathcal{I}[V] \not\models C$ or A is not supported in $\mathcal{I}[V]$ by any atom in C.

- If for each implicant C of ψ holds $\mathcal{I}[V] \not\models C$, then $\mathcal{I}[V] \not\models \psi$ by part (2) of Lemma 137, making this case impossible by contradiction.
- If there exists an implicant C of ψ with $\mathcal{I}[V] \models C$, then by assumption A is not supported in $\mathcal{I}[V]$ by any atom in C. By construction $\mathcal{I}'[V]$ agrees with $\mathcal{I}[V]$ on all atoms except those supporting A in $\mathcal{I}[V]$, thus $\mathcal{I}'[V] \models C$. By Lemma 137 (1), $\mathcal{I}'[V] \models \psi$. Hence $\mathcal{I}'[V] \models (\psi \leftarrow \varphi)$.

In all possible cases $\mathcal{I}'$ satisfies the generalised rule under consideration, thus $\mathcal{I}'$ is a model of S, contradicting the minimality of $\mathcal{I}$. □

This result means that minimal models satisfy only such ground atoms as are supported by appropriate atoms in the consequents of the generalised rules. But the relationship between the supporting atom and the supported ground atom is of a semantic nature. The only guaranteed syntactic relationship between the two is that they share the same relation symbol.

Example 140. Consider a signature containing a unary relation symbol p and constants a and b. Let $S = \{ (p(b) \leftarrow \top) \}$.

The interpretation $\mathcal{I}$ with $dom(\mathcal{I}) = \{1\}$ and $a^{\mathcal{I}} = b^{\mathcal{I}} = 1$ and $p^{\mathcal{I}} = \{(1)\}$ is a minimal model of S. (Note that the only smaller interpretation interprets p with the empty relation and does not satisfy the rule.)

Moreover, $\mathcal{I} \models p(a)$. By the theorem, $p(a)$ is supported in $\mathcal{I}$ by $p(b)$, which can be confirmed by applying the definition.

Definition 141. *An interpretation $\mathcal{I}$ has the* unique name property*, if for each term s, ground term t, and variable assignment V in $dom(\mathcal{I})$ with $s^{I[V]} = t^{I[V]}$ there exists a substitution σ with $s\sigma = t$.*

Obviously, Herbrand interpretations have the unique name property. For minimal interpretations with the unique name property the relationship between the supporting atom and the supported ground atom specialises to the ground instance relationship, which is syntactic and decidable.

The converse of Theorem 139 does not hold for sets with indefinite rules such as $\{ (p(a) \vee p(b) \leftarrow \top) \}$, because the definition of *supported* cannot distinguish between implicants of rule consequent. Both atoms are supported in the Herbrand model $HI(\{p(a), p(b)\})$ of this set, although the model is not minimal.

Regarding definite rules, there was for some time a tacit conviction that satisfying only supported ground atoms was a sufficient criterion for the minimality of models. In the case of Herbrand interpretations the criterion would even be syntactic. But in fact, the converse of Theorem 139 is refuted by rather trivial counter-examples with definite rules.

Example 142. Consider $S = \{ (p \leftarrow p) \}$ and its Herbrand model $HI(\{p\})$. The only ground atom satisfied by $HI(\{p\})$ is p, which is supported in $HI(\{p\})$ by the rule. But $HI(\{p\})$ is not minimal because $HI(\emptyset)$ is also a model of S.

5.2 Fixpoint Semantics of Positive Definite Rules

This subsection first summarises some general results on operators on an arbitrary set and fixpoints of such operators. The arbitrary set will afterwards be specialised to the Herbrand base.

5.2.1 Operators.

Definition 143 (Operator). *Let X be a set. Let $\mathcal{P}(X)$ denote its powerset, the set of subsets of X. An* operator on X *is a mapping $\Gamma : \mathcal{P}(X) \to \mathcal{P}(X)$.*

Definition 144 (Monotonic operator). *Let X be a set. An operator Γ on X is* monotonic*, iff for all subset $M \subseteq M' \subseteq X$ holds: $\Gamma(M) \subseteq \Gamma(M')$.*

Definition 145 (Continuous operator). *Let X be a nonempty set.*

A set $Y \subseteq \mathcal{P}(X)$ of subsets of X is directed*, if every finite subset of Y has an* upper bound *in Y, i.e., for each finite $Y_{fin} \subseteq Y$, there is a set $M \in Y$ such that $\bigcup Y_{fin} \subseteq M$.*

An operator Γ on X is continuous*, iff for each directed set $Y \subseteq \mathcal{P}(X)$ of subsets of X holds: $\Gamma(\bigcup Y) = \bigcup\{\Gamma(M) \mid M \in Y\}$.*

Lemma 146. *Each continuous operator on a nonempty set is monotonic.*

Proof. Let Γ be a continuous operator on $X \neq \emptyset$. Let $M \subseteq M' \subseteq X$. Since Γ is continuous, $\Gamma(M') = \Gamma(M \cup M') = \Gamma(M) \cup \Gamma(M')$, thus $\Gamma(M) \subseteq \Gamma(M')$. □

The converse of this lemma does not hold. Being continuous is a stronger property of operators than being monotonic.

Note that the main purpose of the definition of continuous is to ensure that the operator commutes with set union. But there is no need to require this for all unions of sets, it suffices for unions of directed sets.

5.2.2 Fixpoints of Monotonic and Continuous Operators.

Definition 147 (Fixpoint). *Let Γ be an operator on a set X. A subset $M \subseteq X$ is a* fixpoint *of Γ iff $\Gamma(M) = M$.*

Theorem 148 (Knaster-Tarski, existence of least and greatest fixpoint). *Let Γ be a monotonic operator on a nonempty set X. Then Γ has a least fixpoint $lfp(\Gamma)$ and a greatest fixpoint $gfp(\Gamma)$ with*

$$lfp(\Gamma) = \bigcap\{M \subseteq X \mid \Gamma(M) = M\} = \bigcap\{M \subseteq X \mid \Gamma(M) \subseteq M\}.$$
$$gfp(\Gamma) = \bigcup\{M \subseteq X \mid \Gamma(M) = M\} = \bigcup\{M \subseteq X \mid \Gamma(M) \subseteq M\}.$$

Proof. For the least fixpoint let $L = \bigcap\{M \subseteq X \mid \Gamma(M) \subseteq M\}$.

Consider an arbitrary $M \subseteq X$ with $\Gamma(M) \subseteq M$. By definition of L we have $L \subseteq M$. Since Γ is monotonic, $\Gamma(L) \subseteq \Gamma(M)$. With the assumption $\Gamma(M) \subseteq M$ follows $\Gamma(L) \subseteq M$. Therefore (1) $\Gamma(L) \subseteq \bigcap\{M \subseteq X \mid \Gamma(M) \subseteq M\} = L$.

For the opposite inclusion, from (1) and since Γ is monotonic it follows that $\Gamma(\Gamma(L)) \subseteq \Gamma(L)$. By definition of L therefore (2) $L \subseteq \Gamma(L)$. From (1) and (2) follows that L is a fixpoint of Γ.

Now let $L' = \bigcap\{M \subseteq X \mid \Gamma(M) = M\}$. Then $L' \subseteq L$. because L is a fixpoint of Γ. The opposite inclusion $L \subseteq L'$ holds, since all sets involved in the intersection defining L', are also involved in the intersection defining L.

The proof for the greatest fixpoint is similar. □

Definition 149 (Ordinal powers of a monotonic operator). *Let Γ be a monotonic operator on a nonempty set X. For each finite or transfinite ordinal the* upward and downward power *of Γ is defined as*

$\Gamma \uparrow 0$	$= \emptyset$	*(base case)*	$\Gamma \downarrow 0$	$= X$	
$\Gamma \uparrow \alpha + 1$	$= \Gamma(\Gamma \uparrow \alpha)$	*(successor case)*	$\Gamma \downarrow \alpha + 1$	$= \Gamma(\Gamma \downarrow \alpha)$	
$\Gamma \uparrow \lambda$	$= \bigcup\{\Gamma \uparrow \beta \mid \beta < \lambda\}$	*(limit case)*	$\Gamma \downarrow \lambda$	$= \bigcap\{\Gamma \downarrow \beta \mid \beta < \lambda\}$	

Lemma 150. *Let Γ be a monotonic operator on a nonempty set X. For each ordinal α holds:*

1. $\Gamma \uparrow \alpha \subseteq \Gamma \uparrow \alpha + 1$
2. $\Gamma \uparrow \alpha \subseteq \mathit{lfp}(\Gamma)$.
3. *If $\Gamma \uparrow \alpha = \Gamma \uparrow \alpha + 1$, then $\mathit{lfp}(\Gamma) = \Gamma \uparrow \alpha$.*

Proof. 1. and 2. are shown by transfinite induction on α and 3. as follows:

If $\Gamma \uparrow \alpha = \Gamma \uparrow \alpha + 1$, then $\Gamma \uparrow \alpha = \Gamma(\Gamma \uparrow \alpha)$, i.e., $\Gamma \uparrow \alpha$ is a fixpoint of Γ, therefore $\Gamma \uparrow \alpha \subseteq \mathit{lfp}(\Gamma)$ by 2., and $\mathit{lfp}(\Gamma) \subseteq \Gamma \uparrow \alpha$ by definition. □

Theorem 151. *Let Γ be a monotonic operator on a nonempty set X. There exists an ordinal α such that $\Gamma \uparrow \alpha = \mathit{lfp}(\Gamma)$.*

Proof. Otherwise, for all ordinals α by the previous lemma $\Gamma \uparrow \alpha \subseteq \Gamma \uparrow \alpha + 1$ and $\Gamma \uparrow \alpha \neq \Gamma \uparrow \alpha + 1$. Thus $\Gamma \uparrow$ injectively maps the ordinals onto $\mathcal{P}(X)$, a contradiction as there are "more" ordinals than any set can have elements. □

Theorem 152 (Kleene). *Let Γ be a continuous operator on a nonempty set X. Then $\mathit{lfp}(\Gamma) = \Gamma \uparrow \omega$. ($\omega$ is the first limit ordinal, the one corresponding to $\mathbb{N}$)*

Proof. By 1. from the previous lemma, it suffices to show that $\Gamma \uparrow \omega + 1 = \Gamma \uparrow \omega$.

$$
\begin{aligned}
\Gamma \uparrow \omega + 1 &= \Gamma(\Gamma \uparrow \omega) && \text{by definition, successor case} \\
&= \Gamma\big(\bigcup\{\Gamma \uparrow n \mid n \in \mathbb{N}\}\big) && \text{by definition, limit case} \\
&= \bigcup\big\{\Gamma(\Gamma \uparrow n) \mid n \in \mathbb{N}\big\} && \text{because } \Gamma \text{ is continuous} \\
&= \bigcup\big\{\Gamma \uparrow n + 1 \mid n \in \mathbb{N}\big\} && \text{by definition, successor case} \\
&= \Gamma \uparrow \omega && \text{by definition, base case}
\end{aligned}
$$

□

An analogous result for the greatest fixpoint does not hold: it may well be that $\mathit{gfp}(\Gamma) \neq \Gamma \downarrow \omega$. Note that the decisive step in the proof depends on the operator being continuous. Being monotonic would not be sufficient. The theory of well-founded semantics uses operators that are monotonic, but not necessarily continuous, and therefore not covered by this theorem.

5.2.3 Immediate Consequence Operator for a Set of Positive Definite Rules.

Generalised definite rules according to Definition 125 need not be universal, because so far the results on this class of formulas did not depend on universality. In this subsection, however, we consider only universal formulas.

Recall that an Herbrand interpretation satisfies a set of universal closed formulas iff it satisfies the set of its ground instances (Corollary 60). Assuming a signature with at least one constant, the Herbrand base HB is nonempty.

Let us now apply the results on operators to the case where $X = HB$ and a subset M is a set $B \subseteq HB$ of ground atoms, which induces an Herbrand interpretation. The following generalises a definition first given in [153].

Definition 153 (Immediate consequence operator). *Let S be a set of universal generalised definite rules. Let $B \subseteq HB$ be a set of ground atoms. The* immediate consequence operator $\mathbf{T}_S$ for S *is:*

$$\begin{array}{lll}\mathbf{T}_S : \mathcal{P}(HB) \to \mathcal{P}(HB) & & \\ \quad B & \mapsto \{A \in HB \mid & \textit{there is a ground instance } ((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi) \\ & & \textit{of a member of } S \textit{ with } HI(B) \models \varphi \textit{ and } A = A_i \\ & & \textit{for some } i \textit{ with } 1 \leq i \leq n \qquad \}\end{array}$$

Lemma 154 ($\mathbf{T}_S$ is continuous). *Let S be a set of universal generalised definite rules. The immediate consequence operator $\mathbf{T}_S$ is continuous.*

Lemma 155 ($\mathbf{T}_S$ is monotonic). *Let S be a set of universal generalised definite rules. The immediate consequence operator $\mathbf{T}_S$ is monotonic, that is, if $B \subseteq B' \subseteq HB$ then $\mathbf{T}_S(B) \subseteq \mathbf{T}_S(B')$.*

Recall that for Herbrand interpretations $HI(B) \leq HI(B')$ iff $B \subseteq B'$. Thus, the immediate consequence operator $\mathbf{T}_S$ is also monotonic with respect to $\leq$ on Herbrand interpretations.

Theorem 156. *Let S be a set of universal generalised definite rules. Let $B \subseteq HB$ be a set of ground atoms. Then $HI(B) \models S$ iff $\mathbf{T}_S(B) \subseteq B$.*

Proof. "only if:" Assume $HI(B) \models S$. Let $A \in \mathbf{T}_S(B)$, i.e., $A = A_i$ for some ground instance $((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi)$ of a member of S with $HI(B) \models \varphi$. By assumption $HI(B) \models (A_1 \wedge \ldots \wedge A_n)$, hence $HI(B) \models A$, hence $A \in B$ because A is a ground atom.

"if:" Assume $\mathbf{T}_S(B) \subseteq B$. Let $((A_1 \wedge \ldots \wedge A_n) \leftarrow \varphi)$ be a ground instance of a member of S. It suffices to show that $HI(B)$ satisfies this ground instance. If $HI(B) \not\models \varphi$, it does. If $HI(B) \models \varphi$, then $A_1 \in \mathbf{T}_S(B), \ldots, A_n \in \mathbf{T}_S(B)$ by definition of $\mathbf{T}_S$. By assumption $A_1 \in B, \ldots, A_n \in B$. As these are ground atoms, $HI(B) \models A_1, \ldots, HI(B) \models A_n$. Thus $HI(B)$ satisfies the ground instance. □

5.2.4 Least Fixpoint of a Set of Positive Definite Rules.

Corollary 157. *Let S be a set of universal generalised definite rules. Then $lfp(\mathbf{T}_S) = \mathbf{T}_S \uparrow \omega = Mod_\cap(S) = \{A \in HB \mid S \models A\}$ and $HI(lfp(\mathbf{T}_S))$ is the unique minimal Herbrand model of S.*

Proof. By Lemma 155, $\mathbf{T}_S$ is a monotonic operator on HB, and by Theorem 152, $lfp(\mathbf{T}_S) = \mathbf{T}_S \uparrow \omega$.

Note that $Mod_{HB}(S) \neq \emptyset$ by Theorem 128. Now,

$$\begin{array}{lll}lfp(\mathbf{T}_S) & = \bigcap\{B \subseteq HB \mid \mathbf{T}_S(B) \subseteq B\} & \text{by the Knaster-Tarski Theorem 148} \\ & = \bigcap\{B \subseteq HB \mid HI(B) \models S\} & \text{by Theorem 156} \\ & = \bigcap Mod_{HB}(S) & \text{by Definition 117} \\ & = Mod_\cap(S) & \text{by Definition 118} \\ & = \{A \in HB \mid S \models A\} & \text{by Theorem 119}\end{array}$$

By Theorem 133, $HI(lfp(\mathbf{T}_S))$ is the unique minimal Herbrand model of S. □

The immediate consequence operator for a set of universal generalised definite rules also has a *greatest fixpoint* (Knaster-Tarski Theorem 148). Using similar proof techniques as above one can show [111] that this greatest fixpoint

is $gfp(\mathbf{T}_S) = \mathbf{T}_S \downarrow \omega + 1$, but in general $gfp(\mathbf{T}_S) \neq \mathbf{T}_S \downarrow \omega$. As an example, let $S = \{\, \forall x\big(q \leftarrow p(x)\big),\ \forall x\big(p(f(x)) \leftarrow p(x)\big) \,\}$. Then $\mathbf{T}_S \downarrow \omega = \{q\}$ and $\mathbf{T}_S \downarrow \omega + 1 = \emptyset = gfp(\mathbf{T}_S)$, which in this example is the only fixpoint.

A fixpoint requiring more than ω steps is not in general computably enumerable. The result on the least fixpoint means that $lfp(\mathbf{T}_S)$ is computably enumerable.

The "natural meaning" of a set S of universal generalised definite rules, the unique minimal Herbrand model of S, has several equivalent characterisations. It is the intersection of all Herbrand models of S and satisfies precisely the ground atoms entailed by S. These characterisations allow a declarative understanding of S: each of its rules represents a statement about the application at hand, and a query asks whether something is a logical consequence of these statements, or, equivalently, whether it is true in all their Herbrand models.

The corollary above allows in addition an operational understanding of S based on forward chaining, even though forward chaining is not necessarily the intended operational semantics (backward chaining is in many cases preferable, see Section 6). The unique minimal Herbrand model of S is induced by the smallest fixpoint of $\mathbf{T}_S$. This operator models one step in a forward chaining process: applied to a set of ground atoms, it adds the atoms from the consequents of those ground instances of rules whose antecedent is satisfied in the current set. Being satisfied here means satisfied by the induced Herbrand interpretation, but this can by checked algorithmically using unification. The iteration starts from the empty set. Reaching a fixpoint means that the operator does not generate any new atom. The result guarantees that this happens after finitely many steps or at most as many steps as there are natural numbers.

By the corollary the declarative and this operational semantics coincide: a ground atom is a logical consequence of S if and only if it can be derived by this forward chaining process.

Another procedural semantics based on a backward chaining process called SLD resolution (Subsection 6.4) is also equivalent to the ones above. Backward chaining with SLD resolution succeeds on S with some ground atom as a query if and only if the ground atom is a logical consequence of S.

Authors of rules may thus freely switch between different understandings of their rules, because all of these understandings amount to the same.

Notation 158 (Least fixpoint of a definite program). *For a set S of universal generalised definite rules, the* least fixpoint of S *is $lfp(S) = lfp(\mathbf{T}_S)$.*

5.3 Declarative Semantics of Rules with Negation

The nice results above heavily depend on the rules being positive: their antecedents have to be positive formulas. If the antecedents may contain atoms with negative polarity, as in normal clauses and normal goals (Definition 21), things turn out to be much more difficult. The starting point is to clarify what negative antecedents of rules are supposed to mean.

When working with a set of positive definite rules, more generally, a set of universal generalised definite rules, it is often intuitive to consider everything to be false that does not follow from the set.

This is the common understanding of inductive definitions in mathematics (the inductive definitions of terms and formulas in Section 3 are examples of that). This understanding of inductive definitions is sometimes stressed by concluding the definition, say, of a formula, with "nothing else is a formula".

This is also the common understanding of many specifications one encounters in real life. The time table of a railway company can be seen as a set of ground atoms, each specifying a direct railway connection. The common understanding of such a time table is that any direct connection not explicitly mentioned does not exist. This understanding is naturally extended to connections with changes as follows: if no connections with changes between two places and under certain time constraints can be derived from a time table, then it may be concluded that there are no such connections.

In databases, the common understanding is similar. If a database of students does not list "Mary", then it may be concluded that "Mary" is not a student.

The principle underlying this common understanding has been formalised under the name of "closed world assumption" [137].

It might seem obvious to formalise the closed world assumption by adding additional axioms to the set of formulas so as, so to speak, to complete – or "close" – it. But this is undesirable for two reasons. First, it would blow up the axiomatisation and as a consequence make deduction more time and space consuming. Second, it is in many, if not most, practical cases infeasible – how could one list, or otherwise specify, non-existing train connections or non-students? Note that, in contrast to such applications, mathematics, except in a few cases like inductive definitions, does not rely on closed world assumptions of any kind.

One approach to handling negation was to find deduction methods that employ the closed world assumption without relying on additional, application-dependent, axioms. This approach has two sides: one is to find convenient declarative semantics, i.e., declarative semantics that are both easy to understand and convenient to use in establishing properties; the other is to find efficient automated deduction methods that take the closed world assumption into account.

Another, related approach was to find application independent rewritings of a rule set, which, possibly together with additional application independent axioms, would correspond to the common understanding under the closed world assumption of the original set. This led to the so-called "completion semantics".

Both approaches have yielded unsatisfactory results and are therefore not addressed in the following. Indeed, the intended semantics could not be expressed with Tarski model theory, but required drastic changes to the notion of a model.

The "common understanding" or "intuitive meaning" of a set of rules with negation is what people reading or writing the rules are likely to think they mean. Good candidates for making this common sense notion more precise are the minimal Herbrand models defined earlier. However, not all of them convey the intuitive meaning under the closed world assumption.

Example 159. $S_1 = \{ (q \leftarrow r \wedge \neg p),\ (r \leftarrow s \wedge \neg t),\ (s \leftarrow \top) \}$ has the following three minimal Herbrand models: $HI(\{s, r, q\})$, $HI(\{s, r, p\})$, and $HI(\{s, t\})$.

Intuitively, neither p nor t are "justified" by the rules in S_1. Under the closed world assumption only the first of the minimal models above should be seen as conveying the intuitive meaning of S_1.

The example illustrates that some minimal Herbrand models do not convey the intuitive meaning of rules with negation, because classical model theory treats negated atoms in rule antecedents like (positive) atoms in rule consequents. Indeed, for minimal or non-minimal Herbrand or other models, the rule $(r \leftarrow s \wedge \neg t)$ is equivalent to $(r \vee t \leftarrow s)$ and to $(t \leftarrow s \wedge \neg r)$, hence no interpretation can distinguish these formulas.

Example 160. $S_2 = \{ (p \leftarrow \neg q),\ (q \leftarrow \neg p) \}$ has the following two minimal Herbrand models: $HI(\{p\})$, $HI(\{q\})$.

In this example both minimal Herbrand models of S_2 well convey the intuitive meaning of S_2 under the closed world assumption. Intuitively, the example specifies that exactly one of p and q is true, but it does not specify which.

Example 161. $S_3 = \{ (p \leftarrow \neg p) \}$ has only one minimal Herbrand model: $HI(\{p\})$.

In examples like these, intuition turns out to be somewhat indeterminate and subject to personal preference.

People tending towards a "justification postulate" request dependable justifications for derived truths. The only rule of S_3 does not in this sense "justify" p, because it requires $\neg p$ to hold as a condition for p to hold, thus its outcome violates its own precondition. The "justification postulate" arrives at the conclusion that no model at all conveys the intuitive meaning of S_3, that S_3 should be regarded as inconsistent.

Adherents of a "consistency postulate", on the other hand, insist that every syntactically correct set of normal clauses *is* consistent and must therefore have a model. As there is only one candidate model of S_3, this has to be it, like it or not. Note that the "consistency postulate" is closer to classical model theory, which, for minimal or non-minimal Herbrand or other models, treats the rule in S_3 like the formula p, to which it is logically equivalent.

Example 162. $S_4 = \{ (p \leftarrow \neg p),\ (p \leftarrow \top) \}$ has only one minimal Herbrand model: $HI(\{p\})$.

S_4 extends S_3 with a rule enforcing p. Its only minimal Herbrand model, in which p is true, is perfectly intuitive under the closed world assumption – even with the "justification postulate" that considers S_3 to be inconsistent. The new rule in S_4 clearly "justifies" p. Since intuitively p follows from S_4, the antecedent of the rule $(p \leftarrow \neg p)$ is not satisfied, i.e., the rule is satisfied.

As these examples suggest, only some of the minimal Herbrand models of a set of rules with negation should be retained in order to specify the declarative semantics of the set under the closed world assumption. In the literature the

retained minimal Herbrand models are often called *canonical models* or *preferred models*. With the "justification postulate" the set of retained minimal Herbrand models may be empty, with the "consistency postulate" it always contains at least one model. Some of the formal approaches to the declarative semantics support the "justification postulate" some the "consistency postulate".

Once the notion of canonical models has been formalised, the notion of entailment (Definition 34) and of a theory (Definition 43) can be adapted such that they consider only canonical models. In contrast to the operator Th for closure under classical entailment, which is monotonic (Proposition 45), the appropriately adapted operator $Th_{canonical}$ is not.

Example 163 (Non-monotonicity). $S_5 = \{ (q \leftarrow \neg p) \}$ has two minimal Herbrand models: $HI(\{p\})$ and $HI(\{q\})$. Only the latter conveys the intuitive meaning under the closed world assumption and should be retained as (the only) canonical model. Therefore, $q \in Th_{canonical}(S_5)$.

$S'_5 = S_5 \cup \{ (p \leftarrow \top) \}$ has only one minimal Herbrand model: $HI(\{p\})$, which also conveys the intuitive meaning under the closed world assumption and should be retained as a canonical model. Therefore, $q \notin Th_{canonical}(S'_5)$.

Thus, $S_5 \subseteq S'_5$, but $Th_{canonical}(S_5) \not\subseteq Th_{canonical}(S'_5)$.

Note that non-monotonicity is independent of the choice between the two "postulates". However, any semantics *not* complying with the "consistency postulate" (i.e., all or most semantics complying with the "justification postulate"), is non-monotonic in an even stronger sense. With such a semantics, consistency – defined as usual as the existence of models – is not inherited by subsets: S_4 above is consistent, but S_3 is not, although $S_3 \subseteq S_4$.

As the non-monotonicity of the operator is caused by the non-classical treatment of negation, this kind of negation is also called *non-monotonic negation*.

5.3.1 Stratifiable Rule Sets. Some approaches to formalise the semantics of rules with negation make use of a weak syntactic property of sets of rules that ensures stronger results. The idea is to avoid cases like $(p \leftarrow \neg p)$ and more generally recursion through negative literals. For that purpose the set of rules is partitioned into "strata", and negative literals in the antecedents are required to belong to a lower "stratum".

Definition 164 (Stratification). *Let S be a set of normal clauses (Definition 21). A* stratification of S *is a partition $S_0, \ldots, S_k$ of S such that*

- *For each relation symbol p there is a stratum S_i, such that all clauses of S containing p in their consequent are members of S_i.*
  In this case one says that the relation symbol p is defined in stratum S_i.
- *For each stratum S_j and for each positive literal A in the antecedents of members of S_j, the relation symbol of A is defined in a stratum S_i with $i \leq j$.*
- *For each stratum S_j and for each negative literal $\neg A$ in the antecedents of members of S_j, the relation symbol of A is defined in a stratum S_i with $i < j$.*

A set of normal clauses is called stratifiable*, if there exists a stratification of it.*

Obviously, each definite program is stratifiable by making it its only stratum. The set of normal clauses $S = \{ (r \leftarrow \top),\ (q \leftarrow r),\ (p \leftarrow q \wedge \neg r) \}$ is stratifiable in different ways: the stratum S_0 contains the first clause and the stratum S_1 the last one, while the middle clause may belong to either of the strata. If the middle clause is replaced by $(q \leftarrow p \wedge r)$, the set remains stratifiable, but now there is only one stratification, the middle clause belonging to S_1. The set $S = \{ (p \leftarrow \neg p) \}$ is not stratifiable. More generally, any set of normal clauses with a "cycle of recursion through negation" [9] is not stratifiable.

By definition the stratum S_0 always consists of definite clauses (positive definite rules). Hence the truth values of all atoms of stratum S_0 can be determined without negation being involved. After that the clauses of stratum S_1 refer only to such negative literals whose truth values have already been determined. And so on. The principle simply is to work stratum by stratum, see Subsection 7.1.

All results about stratifiable rule sets depend only on the existence of some stratification and are independent of the concrete specification of the strata.

Note that stratifiability evades any commitment as to the "justification" or "consistency postulate", because rule sets where the postulates make a difference are not stratifiable. Even though stratifiable sets of normal clauses seem to be sufficient for many, if not all, practical programming examples, a semantics is desirable that covers all syntactically correct programs. Indeed, one of the purposes of a semantics of programs is to uncover the "meaning" of unintended, syntactically correct programs. The next three subsections describe attempts to define such semantics, which, alas, do not perfectly meet the objective.

5.3.2 Stable Model Semantics. The stable model semantics [77] is defined in terms of a criterion for retaining a minimal Herbrand model of a set of normal clauses. This criterion is expressed in terms of the following transformation named after the authors of [77].

Definition 165 (Gelfond-Lifschitz transformation). *Let S be a (possibly infinite) set of* ground *normal clauses, i.e., of formulas of the form*

$$A \leftarrow L_1 \wedge \ldots \wedge L_n$$

where $n \geq 0$ and A is a ground atom and the L_i for $1 \leq i \leq n$ are ground literals. Let $B \subseteq HB$. The Gelfond-Lifschitz transform $GL_B(S)$ of S with respect to B *is obtained from S as follows:*

1. *remove each clause whose antecedent contains a literal $\neg A$ with $A \in B$.*
2. *remove from the antecedents of the remaining clauses all negative literals.*

The transformation corresponds to a partial evaluation of S in the interpretation $HI(B)$. The clauses removed in the first step are true in $HI(B)$ because their antecedent is false in $HI(B)$. The literals removed from the antecedents of the remaining clauses in the second step, are also true in $HI(B)$.

Note that the Gelfond-Lifschitz transform $GL_B(S)$ of a set S of ground normal clauses is a (possibly infinite) set of definite clauses, i.e., a set of universal generalised definite rules. Therefore $GL_B(S)$ has a unique minimal Herbrand model with all characterisations according to Corollary 157.

Definition 166 (Stable Model). *Let S be a (possibly infinite) set of ground normal clauses. An Herbrand interpretation $HI(B)$ is a* stable model of S*, iff it is the unique minimal Herbrand model of $GL_B(S)$.*

A stable model *of a set S of normal clauses is a stable model of the (possibly infinite) set of ground instances of S.*

For this notion to be well-defined, we have to ensure that the unique minimal Herbrand model of $GL_B(S)$ is indeed also a model of S.

Lemma 167. *Let S be a set of ground normal clauses and $HI(B)$ an Herbrand interpretation. $HI(B) \models S$ iff $HI(B) \models GL_B(S)$.*

Proof. Let S_1 be the set of clauses obtained from S by applying the first step of the transformation. Each clause in $S \setminus S_1$ is satisfied by $HI(B)$, because its antecedent is falsified by $HI(B)$. Thus, $HI(B) \models S$ iff $HI(B) \models S_1$.

Let S_2 be the set of clauses obtained from S_1 by applying the second step of the transformation. For each clause $C_2 \in S_2$ there is a clause $C_1 \in S_1$ such that C_2 is obtained from C_1 by removing the negative literals from its antecedent. Since $C_1 \in S_1$, for any such negative literal $\neg A$ in its antecedent, $A \notin B$, i.e., $HI(B) \models \neg A$. Therefore, $HI(B) \models C_1$ iff $HI(B) \models C_2$. □

Theorem 168. *Let S be a set of normal clauses. Each stable model of S is a minimal Herbrand model of S.*

Proof. By definition of a stable model, it suffices to show the result for a set S of *ground* normal clauses.

Let $B' \subseteq B \subseteq HB$ such that $HI(B)$ is a stable model of S and $HI(B')$ is also a model of S, i.e., $HI(B') \models S$. If we establish that $HI(B') \models GL_B(S)$, then $B' = B$ because, by definition of a stable model, $HI(B)$ is the unique minimal Herbrand model of $GL_B(S)$.

Let $C \in GL_B(S)$. By definition of $GL_B(S)$ there exists a clause $D \in S$, such that C is obtained from D by removing the negative literals from its antecedent. If $\neg A$ is such a literal, then $A \notin B$, and, since $B' \subseteq B$, also $A \notin B'$. Therefore, $C \in GL_{B'}(S)$, and by the previous lemma $HI(B') \models C$. □

It is easy to verify that the stable models of the examples above are as follows:

Example 169.
$S_1 = \{ (q \leftarrow r \wedge \neg p),\ (r \leftarrow s \wedge \neg t),\ (s \leftarrow \top) \}$ has one stable model: $HI(\{s, r, q\})$.
$S_2 = \{ (p \leftarrow \neg q),\ (q \leftarrow \neg p) \}$ has two stable models: $HI(\{p\})$ and $HI(\{q\})$.
$S_3 = \{ (p \leftarrow \neg p) \}$ has no stable model.
$S_4 = \{ (p \leftarrow \neg p),\ (p \leftarrow \top) \}$ has one stable model: $HI(\{p\})$.

Thus, the stable model semantics coincides with the intuitive understanding based on the "justification postulate". The unintuitive minimal models of the examples turn out not to be stable, and the stability criterion retains only those minimal models that are intuitive. A set may have several stable models or exactly one or none. Each stratifiable set has exactly one stable model.

The remarks about non-monotonicity on page 59 apply also to stable models, including non-inheritance of consistency by subsets. To give up such a fundamental principle can be seen as a serious drawback of the stable model semantics.

5.3.3 Well-Founded Semantics. The well-founded semantics [155] of a set of normal clauses is defined as the least fixpoint of a monotonic operator that explicitly specifies derivations both of positive and of negative ground literals. Recall that the immediate consequence operator (Definition 153) for a set of definite clauses explicitly specifies derivations only of positive ground literals.

In contrast to the stable model semantics, the well-founded semantics specifies for each set of normal clauses a *single* model, a so-called *well-founded model*.

A well-founded model can be either *total*, in which case it makes each ground atom true or false like a standard model, or *partial*, in which case it makes some ground atoms neither true nor false, but undefined.

Recall that $\overline{L}$ denotes the complement of a literal L with $\overline{A} = \neg A$ and $\overline{\neg A} = A$ for an atom A (Definition 19). *HB* denotes the Herbrand base, the set of all ground atoms for the given signature.

Notation 170. *For a set I of ground literals:*
$\overline{I} = \{\, \overline{L} \mid L \in I \,\}$ *and* $pos(I) = I \cap HB$ *and* $neg(I) = \overline{I} \cap HB$.
Thus, $I = pos(I) \cup \overline{neg(I)}$.

Definition 171. *A set I of ground literals is* consistent, *iff* $pos(I) \cap neg(I) = \emptyset$. *Otherwise, I is* inconsistent.

Two sets I_1 and I_2 of ground literals are (in)consistent *iff $I_1 \cup I_2$ is.*

A literal L and a set I of ground literals are (in)consistent *iff $\{L\} \cup I$ is.*

Definition 172 (Partial interpretation). *A* partial interpretation *is a consistent set of ground literals.*

A partial interpretation I is called total, *iff* $pos(I) \cup neg(I) = HB$, *that is, for each ground atom A either $A \in I$ or $\neg A \in I$.*

For a total interpretation I, the Herbrand interpretation induced by I is defined as $HI(I) = HI(pos(I))$.

Definition 173 (Model relationship for partial interpretations). *Let I be a partial interpretation. Then $\top$ is* satisfied *in I and $\bot$ is* falsified *in I.*

A ground literal L is

satisfied *or true in I*	*iff $L \in I$.*
falsified *or false in I*	*iff $\overline{L} \in I$.*
undefined *in I*	*iff $L \notin I$ and $\overline{L} \notin I$.*

A conjunction $L_1 \wedge \ldots \wedge L_n$ of ground literals, $n \geq 0$, is

satisfied *or true in I*	*iff each L_i for $1 \leq i \leq n$ is satisfied in I.*
falsified *or false in I*	*iff at least one L_i for $1 \leq i \leq n$ is falsified in I.*
undefined *in I*	*iff each L_i for $1 \leq i \leq n$ is satisfied or undefined in I and at least one of them is undefined in I.*

A ground normal clause $A \leftarrow \varphi$ is

satisfied *or true in I*	*iff A is satisfied in I or φ is falsified in I.*
falsified *or false in I*	*iff A is falsified in I and φ is satisfied in I.*
weakly falsified *in I*	*iff A is falsified in I and φ is satisfied or undefined in I.*

A normal clause is
satisfied *or true in* I *iff each of its ground instances is.*
falsified *or false in* I *iff at least one of its ground instances is.*
weakly falsified *in* I *iff at least one of its ground instances is.*
A set of normal clauses is
satisfied *or true in* I *iff each of its members is.*
falsified *or false in* I *iff at least one of its members is.*
weakly falsified *in* I *iff at least one of its members is.*

For a total interpretation I the cases "undefined" and "weakly falsified" are impossible, and obviously the notion of satisfied (or falsified, respectively) in I in the sense above coincides with the notion of satisfied (or falsified, respectively) in $HI(I)$ in the classical sense.

Definition 174 (Total and partial model). *Let S be a set of normal clauses.*
A total interpretation I is a total model of S*, iff S is satisfied in I.*
A partial interpretation I is a partial model of S*, iff there exists a total model I' of S with $I \subseteq I'$.*

Note that if a ground normal clause is weakly falsified, but not falsified in a partial interpretation I, then its consequent *is* falsified in I and some literals in its antecedent are undefined in I. No extension of I with additional literals can satisfy the consequent. The only way to satisfy the normal clause is to extend I by the complement of one of the undefined antecedent literals, thus falsifying the clause's antecedent. Any extension of I that satisfies all of those antecedent literals, falsifies the normal clause.

Lemma 175. *Let S be a set of normal clauses and I a partial interpretation. If no clause in S is weakly falsified in I, then I is a partial model of S.*

Proof. Let no clause in S be weakly falsified in I. Let $I' = I \cup (HB \setminus neg(I))$, that is, I extended by all ground atoms consistent with I. Then I' is total.

Consider an arbitrary ground instance $A \leftarrow \varphi$ of a member of S. Since it is not weakly falsified in I, it is by definition not falsified in I either and therefore satisfied in I, i.e., $A \in I$ or $\overline{L} \in I$ for some literal L in the antecedent φ. This membership cannot be affected by adding literals to I that preserve its consistency. Thus, $A \in I'$ or $\overline{L} \in I'$, and $A \leftarrow \varphi$ is satisfied also in I'.

Hence I' is a total model of S and I is a partial model of S. □

The basis for drawing negative conclusions and the notion most central to the well-founded semantics is that of an *unfounded set* of ground atoms. Given a partial interpretation I, i.e., a set of ground literals that are already known (or assumed) to be true, a set U of ground atoms is unfounded, if the normal clauses at hand give no reason to consider any member of U to be true. More precisely, each atom $A \in U$ occurs in the consequent only of such ground instances of clauses that do not justify A. This can happen for two reasons: because the antecedent of the ground instance is falsified in I or because the antecedent of the ground instance contains an unfounded atom, i.e., a member of U.

Definition 176 (Unfounded set of ground atoms). *Let S be a set of normal clauses, I a partial interpretation, and $U \subseteq HB$ a set of ground atoms.*

U is an unfounded set *with respect to S and I, if for each $A \in U$ and for each ground instance $A \leftarrow L_1 \wedge \ldots \wedge L_n$, $n \geq 1$, of a member of S having A as its consequent, at least one of the following holds:*

1. *$L_i \in \overline{I}$ for some positive or negative L_i with $1 \leq i \leq n$. (L_i is falsified in I)*
2. *$L_i \in U$ for some positive L_i with $1 \leq i \leq n$. (L_i is unfounded)*

A literal fulfilling one of these conditions is called a witness of unusability *for the ground instance of a clause.*

U is a maximal unfounded set *with respect to S and I, iff U is an* unfounded set *with respect to S and I and no proper superset of U is.*

Example 177. Let $S = \{ (q \leftarrow p),\ (r \leftarrow s),\ (s \leftarrow r) \}$ and $I = \{\neg p, \neg q\}$. The set $U = \{q, r, s\}$ is unfounded with respect to S and I. The atom q is unfounded by condition 1, the atoms r and s by condition 2.

U is not maximal, because $U' = \{p, q, r, s\}$ is also unfounded w.r.t. S and I. Note that p is unfounded because there is no rule with consequent p. Furthermore, p would even be unfounded if it were satisfied in I.

Note that the empty set is unfounded with respect to every set of normal clauses and every partial interpretation. Note also that the union of sets that are unfounded with respect to the same S and I is also unfounded with respect to the same S and I. As a consequence, the following lemma holds:

Lemma 178. *Let S be a set of normal clauses and I a partial interpretation. There exists a unique maximal unfounded set with respect to S and I, which is the union of all unfounded sets with respect to S and I.*

Starting from "knowing" I in the example above, the atoms r and s depend on each other, but none of them has to be true for other reasons. Thus, if we choose to consider them or one of them to be false, we will not be forced to undo this decision. Making one of them false preserves the other's being unfounded. Generalising this observation, we get:

Lemma 179. *Let S be a set of normal clauses, I a partial interpretation, and U' an unfounded set with respect to S and I, such that $pos(I) \cap U' = \emptyset$.*

For each $U \subseteq U'$, its remainder $U' \setminus U$ is unfounded w.r.t. S and $I \cup \overline{U}$.

Proof. The condition $pos(I) \cap U' = \emptyset$ ensures that $I \cup \overline{U}$ is consistent.

Any atom falsified in I remains falsified in $I \cup \overline{U}$, thus any witness of unusability w.r.t. S and I by condition 1, is also a witness of unusability w.r.t. S and $I \cup \overline{U}$ by condition 1.

Any $A \in U$ that is a witness in U' of unusability w.r.t. S and I by condition 2, is not in $U' \setminus U$ and can no longer satisfy condition 2. But $\overline{A} \in \overline{U} \subseteq I \cup \overline{U}$. Hence, A is a witness of unusability w.r.t. S and $I \cup \overline{U}$ by condition 1. □

In a sense the lemma allows to make unfounded atoms false without affecting the unfoundedness of others. The next lemma is a kind of opposite direction, in a sense it allows to make falsified atoms unfounded. Recall that $I = pos(I) \cup \overline{neg(I)}$.

Lemma 180. *Let S be a set of normal clauses and I a partial interpretation. If no clause in S is weakly falsified in I, then $neg(I)$ is unfounded with respect to S and $pos(I)$.*

Proof. Let $A \in neg(I)$ and $A \leftarrow \varphi$ an arbitrary ground instance of a member of S. Since A is falsified in I and $A \leftarrow \varphi$ is not weakly falsified in I, some literal L in φ is falsified in I. If L is positive, then $L \in neg(I)$ and L is a witness of unusability for $A \leftarrow \varphi$ by condition 2. If L is negative, then $L \in \overline{pos(I)}$ and L is a witness of unusability for $A \leftarrow \varphi$ by condition 1. □

Definition 181. *Let $\mathcal{PI} = \{\, I \subseteq HB \cup \overline{HB} \mid I \text{ is consistent} \,\}$, and note that $\mathcal{P}(HB) \subseteq \mathcal{PI}$. Let S be a set of normal clauses. We define three operators:*

$$\begin{aligned}
\mathbf{T}_S :\ & \mathcal{PI} \to \mathcal{P}(HB) \\
& I \ \mapsto \{\, A \in HB \mid \text{there is a ground instance } (A \leftarrow \varphi) \\
& \qquad\qquad \text{of a member of } S \text{ such that } \varphi \text{ is satisfied in } I \,\} \\
\mathbf{U}_S :\ & \mathcal{PI} \to \mathcal{P}(HB) \\
& I \ \mapsto \text{the maximal subset of } HB \text{ that is unfounded with respect to } S \text{ and } I \\
\mathbf{W}_S :\ & \mathcal{PI} \to \mathcal{PI} \\
& I \ \mapsto \mathbf{T}_S(I) \cup \overline{\mathbf{U}_S(I)}
\end{aligned}$$

If S is a set of definite clauses, that is, if all antecedents are positive, the operator $\mathbf{T}_S$ coincides with the immediate consequence operator $\mathbf{T}_S$ from Definition 153. Whether or not S is definite, $\mathbf{T}_S(I)$ is a set of ground atoms. The operator $\mathbf{U}_S$ is well-defined by Lemma 178 and also produces a set of ground atoms. Starting from "knowing" I, the ground atoms in $\mathbf{T}_S(I)$ are those that have to be true, whereas those in $\mathbf{U}_S(I)$ are unfounded. Note that the definition of unfounded implies $\mathbf{T}_S(I) \cap \mathbf{U}_S(I) = \emptyset$. This ensures the consistency of $\mathbf{W}_S(I)$, which satisfies what has to be true and falsifies all unfounded ground atoms.

Example 182. Assume a signature with $HB = \{p, q, r, s, t\}$, and let $I_0 = \emptyset$ and $S = \{\, (q \leftarrow r \wedge \neg p),\ (r \leftarrow s \wedge \neg t),\ (s \leftarrow \top) \,\}$.

$\mathbf{T}_S(I_0) = \{s\}$	$\mathbf{T}_S(I_1) = \{s, r\}$	$\mathbf{T}_S(I_2) = \{s, r, q\}$
$\mathbf{U}_S(I_0) = \{p, t\}$	$\mathbf{U}_S(I_1) = \{p, t\}$	$\mathbf{U}_S(I_2) = \{p, t\}$
$\mathbf{W}_S(I_0) = \{s, \neg p, \neg t\} = I_1$	$\mathbf{W}_S(I_1) = \{s, r, \neg p, \neg t\} = I_2$	$\mathbf{W}_S(I_2) = \{s, r, q, \neg p, \neg t\}$

$\mathbf{T}_S(\emptyset)$ is nonempty only if S contains a clause with antecedent $\top$ or empty.

Atoms such as p and t, which do not appear in any consequents, are always unfounded. $\mathbf{U}_S(I)$ can never contain the consequent of a satisfied clause instance whose antecedent is satisfied, too. This explains why $\mathbf{U}_S(I_2)$ is maximal. For the maximality of $\mathbf{U}_S(I_1)$ note that for q to be unfounded either r would have to be unfounded, which is impossible by $r \in \mathbf{T}_S(I_1)$, or one of the antecedent literals r and $\neg p$ would have to be falsified in I_1, but r is undefined and $\neg p$ is satisfied in I_1. The maximality of $\mathbf{U}_S(I_0)$ can be confirmed by similar arguments.

Lemma 183. $\mathbf{T}_S$, $\mathbf{U}_S$, *and* $\mathbf{W}_S$ *are monotonic.*[14]

Proof. Immediate from the definition of the operators. □

Theorem 184 (Existence of least fixpoint). *Let S be a set of normal clauses. The operator $\mathbf{W}_S$ has a least fixpoint $lfp(\mathbf{W}_S)$ with*

$$lfp(\mathbf{W}_S) \;=\; \bigcap\{I \in \mathcal{PI} \mid \mathbf{W}_S(I) = I\} \;=\; \bigcap\{I \in \mathcal{PI} \mid \mathbf{W}_S(I) \subseteq I\}.$$

Moreover, $lfp(\mathbf{W}_S)$ is a partial interpretation and a partial model of S.

Proof. The first part follows from the Knaster-Tarski Theorem 148. For the second part, both consistency and that no clause in S is weakly falsified, are shown by transfinite induction. Lemma 175 ensures the model property. □

Definition 185 (Well-founded model). *Let S be a set of normal clauses. The well-founded model of S is its partial model $lfp(\mathbf{W}_S)$.*

The well-founded model may be total, in which case it specifies a truth value for each ground atom, or partial, in which case it leaves some atoms undefined.

The examples considered earlier for the stable model semantics have the following well-founded models. Note that S_1 is the set used for illustrating the operators in the previous example. One more step will reproduce the fixpoint.

Example 186.

$S_1 = \{\, (q \leftarrow r \wedge \neg p),\ (r \leftarrow s \wedge \neg t),\ (s \leftarrow \top) \,\}$ has the well-founded model $\{s, r, q, \neg p, \neg t\}$. It is total.

$S_2 = \{\, (p \leftarrow \neg q),\ (q \leftarrow \neg p) \,\}$ has the well-founded model $\emptyset$. It is partial and leaves the truth values of p and of q undefined.

$S_3 = \{\, (p \leftarrow \neg p) \,\}$ has the well-founded model $\emptyset$. It is partial and leaves the truth value of p undefined.

$S_4 = \{\, (p \leftarrow \neg p),\ (p \leftarrow \top) \,\}$ has the well-founded model $\{p\}$. It is total.

Thus, the well-founded semantics coincides with the intuitive understanding based on the "consistency postulate". Each set of normal clauses has a unique model, but this model does not necessarily commit to truth values for all atoms. Each stratifiable set of normal clauses has a total well-founded model.

Note that the operators are monotonic, but not necessarily continuous, thus the Kleene Theorem 152 ensuring that a fixpoint is reached with at most ω steps, is not applicable. Indeed there are examples for which $lfp(\mathbf{W}_S) \neq \mathbf{W}_S \uparrow \omega$.

Example 187. By definition $\mathbf{W}_S \uparrow 0 = \emptyset$. Assume a signature containing no other symbols than those occurring in the following set of normal clauses. Let $S =$

$\{\, p(a) \leftarrow \top,\quad p(f(x)) \leftarrow p(x),\quad q(y) \leftarrow p(y),\quad s \leftarrow p(z) \wedge \neg q(z),\quad r \leftarrow \neg s \,\}$

$\mathbf{T}_S \uparrow 1$	$= \{\, p(a) \,\}$	
$\mathbf{U}_S \uparrow 1$	$= \emptyset$	
$\mathbf{W}_S \uparrow 1$	$= \{\, p(a) \,\}$	
$\mathbf{T}_S \uparrow 2$	$= \{\, p(a),\ p(f(a)) \,\}$	$\cup\ \{\, q(a) \,\}$
$\mathbf{U}_S \uparrow 2$	$= \emptyset$	
$\mathbf{W}_S \uparrow 2$	$= \{\, p(a),\ p(f(a)) \,\}$	$\cup\ \{\, q(a) \,\}$

[14] but not in general continuous!

$$
\begin{array}{lll}
\mathbf{T}_S\uparrow n+1 & = & \{\, p(a),\ldots,p(f^n(a))\,\} \cup \{\, q(a),\ldots,q(f^{n-1}(a))\,\} \\
\mathbf{U}_S\uparrow n+1 & = & \emptyset \\
\mathbf{W}_S\uparrow n+1 & = & \{\, p(a),\ldots,p(f^n(a))\,\} \cup \{\, q(a),\ldots,q(f^{n-1}(a))\,\} \\
\hline
\mathbf{T}_S\uparrow \omega & = & \{\, p(a),\ldots,p(f^n(a)),\ldots\,\} \cup \{\, q(a),\ldots,q(f^n(a)),\ldots\,\} \\
\mathbf{U}_S\uparrow \omega & = & \emptyset \\
\mathbf{W}_S\uparrow \omega & = & \{\, p(a),\ldots,p(f^n(a)),\ldots\,\} \cup \{\, q(a),\ldots,q(f^n(a)),\ldots\,\} \\
\hline
\mathbf{T}_S\uparrow \omega+1 & = & \{\, p(a),\ldots,p(f^n(a)),\ldots\,\} \cup \{\, q(a),\ldots,q(f^n(a)),\ldots\,\} \\
\mathbf{U}_S\uparrow \omega+1 & = & \{\, s\,\} \\
\mathbf{W}_S\uparrow \omega+1 & = & \{\, p(a),\ldots,p(f^n(a)),\ldots\,\} \cup \{\, q(a),\ldots,q(f^n(a)),\ldots\,\} \cup \{\, \neg s\,\} \\
\hline
\mathbf{T}_S\uparrow \omega+2 & = & \{\, p(a),\ldots,p(f^n(a)),\ldots\,\} \cup \{\, q(a),\ldots,q(f^n(a)),\ldots\,\} \cup \{\, r\,\} \\
\mathbf{U}_S\uparrow \omega+2 & = & \{\, s\,\} \\
\mathbf{W}_S\uparrow \omega+2 & = & \{\, p(a),\ldots,p(f^n(a)),\ldots\,\} \cup \{\, q(a),\ldots,q(f^n(a)),\ldots\,\} \cup \{\, \neg s,\ r\,\}
\end{array}
$$

It is debatable whether sets of normal clauses like that are likely to be needed in practice. In [155] doubts are expressed that such cases are common. But this position is questionable in view of the fact that the set S above is the (standard) translation into normal clauses of the following set of generalised rules:

$$\{\, p(a) \leftarrow \top,\quad p(f(x)) \leftarrow p(x),\quad q(y) \leftarrow p(y),\quad r \leftarrow \forall z\big(p(z) \Rightarrow q(z)\big)\,\}$$

Admittedly, range restricted universal quantification with ranges returning infinitely many bindings for the quantified variables will, in general, hardly be evaluable in finite time.

However, more advanced evaluation methods might well be devised that could recognise, like in the example above, the necessary truth of a universally quantified formula. Furthermore, computable semantics are needed not only for those programs considered acceptable, but also for those considered buggy – as long as they are syntactically correct. From this point of view it is a serious drawback of the well-founded semantics that it is not always computable.

5.3.4 Stable and Well-Founded Semantics Compared. The well-founded semantics and the stable model semantics relate to each other as follows. For space reasons, the proofs are not included in this survey (they can be found in [155], for instance).

If a rule set is stratifiable (which holds in particular if it is definite), then it has a unique minimal model, which is its only stable model and is also its well-founded model and total.

If a rule set S has a total well-founded model, then this model is also the single stable model of S. Conversely, if a rule set S has a single stable model, then this model is also the well-founded model of S and it is total.

If a rule set S has a partial well-founded model I that is not total, i.e., in which some ground atoms have the truth value undefined, then S has either no stable model or more than one stable model.

In the latter case, a ground atom is true (or false, respectively) in all stable models of S if and only if it is true (or false, respectively) in I.

In other words, if a rule set S has a partial (non-total) well-founded model I and at least one stable model, then a ground atom A is undefined in I if and only if it has different truth values in different stable models of S.

Furthermore, if a rule set has no stable model, then some ground atoms are undefined in its well-founded model.

Roughly speaking, the stable model semantics and the well-founded semantics tend to agree with each other and with the intuition, when there is a unique minimal Herbrand model that conveys the intuitive meaning.

When there are several minimal Herbrand models that convey the intuitive meaning, such as for $S_2 = \{ (p \leftarrow \neg q),\ (q \leftarrow \neg p) \}$ with minimal models $HI(\{p\})$ and $HI(\{q\})$, then these tend to be exactly the stable models, and the well-founded model tends to be partial, because it represents their "merge" (being defined where they agree and undefined where they disagree).

When no minimal Herbrand model clearly conveys the intuitive meaning, such as for $S_3 = \{ (p \leftarrow \neg p) \}$ with minimal model $HI(\{p\})$, then there tends to exist no stable model (corresponding to the "justification postulate") whereas the well-founded model exists (corresponding to the "consistency postulate"), but tends to leave the truth values of all atoms undefined.

Thus, the well-founded semantics cannot differentiate between the two critical cases, although they are quite different.

5.3.5 Inflationary Semantics. In this subsection, we restrict our attention to *datalog*$^\neg$ programs. Thus, in this case, the Herbrand universe is always a finite universe denoted as **dom** and the Herbrand base HB is finite, too. A *datalog*$^\neg$ program P is a finite set of normal clauses. Recall from Definition 21 that a *normal clause* is a rule r of the form

$$A \leftarrow L_1, \ldots, L_m$$

where $m \geq 0$ and A is an atom $R_0(\boldsymbol{x}_0)$. Each L_i is an atom $R_i(\boldsymbol{x}_i)$ or a negated atom $\neg R_i(\boldsymbol{x}_i)$. The arguments $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_m$ are vectors of variables or constants (from **dom**). Every variable in $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_m$ must occur in some unnegated atom $L_i = R_i(\boldsymbol{x}_i)$, i.e., the clause must be range restricted (Definition 23).

We denote the head (consequent) of a rule r as $H(r)$, and the body (antecedent) of r as $B(r)$. We further distinguish the positive and negative literals in the body as follows:

$$B^+(r) = \{R(\boldsymbol{x}) \mid \exists i\, L_i = R(\boldsymbol{x})\}, \qquad B^-(r) = \{R(\boldsymbol{x}) \mid \exists i\, L_i = \neg R(\boldsymbol{x})\}$$

Let us now extend the definition of the *immediate consequence operator*

$$\mathbf{T}_P(\mathbf{I}) : \mathcal{P}(HB) \rightarrow \mathcal{P}(HB)$$

(cf. Definition 153) to rules containing negated atoms. Note that since HB is finite, so is $\mathcal{P}(HB)$.

Definition 188 (Immediate consequence operator $\mathbf{T}_P(\mathbf{I})$ for datalog$^\neg$). *Given a datalog$^\neg$ program P and an instance $\mathbf{I}$ over its schema $sch(P)$, a fact $R(\boldsymbol{t})$ is an* immediate consequence *for $\mathbf{I}$ and P (denoted as $\mathbf{T}_P(\mathbf{I})$), if either R is an extensional predicate symbol of P and $R(\boldsymbol{t}) \in \mathbf{I}$, or there exists some ground instance r of a rule in P such that*

- $H(r) = R(\boldsymbol{t})$,
- $B^+(r) \subseteq \mathbf{I}$, *and*
- $B^-(r) \cap \mathbf{I} = \emptyset$.

The *inflationary semantics* [5, 2] is inspired by inflationary fixpoint logic [86]. In place of the immediate consequence operator $\mathbf{T}_P$, it uses the *inflationary operator* $\widetilde{\mathbf{T}}_P$, which (for any datalog$^\neg$ program P) is defined as follows:

$$\widetilde{\mathbf{T}}_P(\mathbf{I}) = \mathbf{I} \cup \mathbf{T}_P(\mathbf{I})$$

Definition 189 (Inflationary semantics of datalog$^\neg$). *Given a datalog$^\neg$ program P and an instance $\mathbf{I}$ over the extensional predicate symbols of P, the inflationary semantics of P w.r.t. $\mathbf{I}$, denoted as $P_{inf}(\mathbf{I})$, is the limit of the sequence $\{\widetilde{\mathbf{T}}_P^i(\mathbf{I})\}_{i\geq 0}$, where $\widetilde{\mathbf{T}}_P^0(\mathbf{I}) = \mathbf{I}$ and $\widetilde{\mathbf{T}}_P^{i+1}(\mathbf{I}) = \widetilde{\mathbf{T}}_P(\widetilde{\mathbf{T}}_P^i(\mathbf{I}))$.*

By the definition of $\widetilde{\mathbf{T}}_P$, the following sequence of inclusions holds:

$$\widetilde{\mathbf{T}}_P^0(\mathbf{I}) \subseteq \widetilde{\mathbf{T}}_P^1(\mathbf{I}) \subseteq \widetilde{\mathbf{T}}_P^2(\mathbf{I}) \subseteq \ldots$$

Furthermore, each set in this sequence is a subset of the finite set HB, such that the sequence clearly reaches a fixpoint $P_{inf}(\mathbf{I})$ after a finite number of steps. However, $HI(P_{inf}(\mathbf{I}))$ is a model of P containing $\mathbf{I}$, but not necessarily a *minimal* model containing $\mathbf{I}$.

Example 190. Let P be the program: $\{\, (p \leftarrow s \wedge \neg q),\ (q \leftarrow s \wedge \neg p)\,\}$. Let $\mathbf{I} = \{s\}$. Then $P_{inf}(\mathbf{I}) = \{s, p, q\}$. Although $HI(P_{inf}(\mathbf{I}))$ is a model of P, it is not minimal. The minimal models containing $\mathbf{I}$ are $HI(\{s, p\})$ and $HI(\{s, q\})$.

The above example shows that given an instance $\mathbf{I}$, the inflationary semantics $P_{inf}(\mathbf{I})$ of P w.r.t. $\mathbf{I}$ may not yield any of the minimal models that convey the intuitive meaning of P. Moreover, $P_{inf}(\mathbf{I})$ is not necessarily the least fixpoint of $\widetilde{\mathbf{T}}_P$ containing $\mathbf{I}$, either. In fact, the inflationary operator $\widetilde{\mathbf{T}}_P$ is not monotonic. With the above example, let I_1 be $\{s\}$, and I_2 be $\{s, p\}$. Then $\widetilde{\mathbf{T}}_P(I_1) = \{s, p, q\}$, and $\widetilde{\mathbf{T}}_P(I_2) = \{s, p\}$.

Thus the existence of a least fixpoint cannot be guaranteed by the Knaster-Tarski Theorem 148, but then it might be guaranteed by the finiteness of $\mathcal{P}(HB)$. However, the problem is[15] that there may be different minimal fixpoints. In the example above, both $\{s, p\}$ and $\{s, q\}$ are fixpoints of $\widetilde{\mathbf{T}}_P$ containing $\{s\}$, but none of their proper subsets is.

Let us now see how the inflationary semantics behaves with the examples used earlier to illustrate and compare the other approaches.

[15] Another problem is that even if a least fixpoint exists, Lemma 150 is not applicable. If $\widetilde{\mathbf{T}}_P^n(\mathbf{I}) = \widetilde{\mathbf{T}}_P^{n+1}(\mathbf{I})$, this might not be the least fixpoint.

Example 191. $S_1 = \{ (q \leftarrow r \wedge \neg p),\ (r \leftarrow s \wedge \neg t),\ (s \leftarrow \top) \}$
$\widetilde{\mathbf{T}}^1_{S_1}(\emptyset) = \{s\}$,
$\widetilde{\mathbf{T}}^2_{S_1}(\emptyset) = \{s, r\}$,
$\widetilde{\mathbf{T}}^3_{S_1}(\emptyset) = \{s, r, q\} = \widetilde{\mathbf{T}}^4_{S_1}(\emptyset)$.
Thus the inflationary fixpoint of S_1 is $\{s, r, q\}$, which agrees with the stable and the well-founded semantics and with the intuitive meaning.

Example 192. $S_2 = \{ (p \leftarrow \neg q),\ (q \leftarrow \neg p) \}$
$\widetilde{\mathbf{T}}^1_{S_2}(\emptyset) = \{p, q\} = \widetilde{\mathbf{T}}^2_{S_2}(\emptyset)$.
Thus the inflationary fixpoint of S_2 is $\{p, q\}$, which is not minimal and disagrees with the stable and the well-founded semantics and with the intuitive meaning. It represents the union of all minimal models.

Example 193. $S_3 = \{ (p \leftarrow \neg p) \}$
$\widetilde{\mathbf{T}}^1_{S_3}(\emptyset) = \{p\} = \widetilde{\mathbf{T}}^2_{S_3}(\emptyset)$.
The inflationary fixpoint of S_3 is $\{p\}$, which corresponds to the intuitive understanding based on the "consistency postulate", but differs from both the stable model semantics (there is no stable model) and the well-founded semantics (the well-founded model leaves the truth value of p undefined).

Example 194. $S_4 = \{ (p \leftarrow \neg p),\ (p \leftarrow \top) \}$
$\widetilde{\mathbf{T}}^1_{S_4}(\emptyset) = \{p\} = \widetilde{\mathbf{T}}^2_{S_4}(\emptyset)$.
The inflationary fixpoint of S_4 is $\{p\}$, which agrees with the stable and the well-founded semantics and with the intuitive meaning.

These examples may give the impression that the inflationary semantics just handles the critical cases differently from the other approaches, but agrees with them in uncritical cases. However, this is not so.

Example 195. $S_5 = \{ (r \leftarrow \neg q),\ (q \leftarrow \neg p) \}$ has two minimal models $HI(\{q\})$ and $HI(\{p, r\})$, of which only the first conveys the intuitive meaning under the closed world assumption. This model coincides with the only stable model and with the well-founded model, which is total. Note that the set is stratifiable.
The inflationary fixpoint of S_5 is $\widetilde{\mathbf{T}}^1_{S_5}(\emptyset) = \{q, r\}$, which is neither intuitive nor related to the minimal models in any systematic way.

The inflationary semantics gives up a fundamental principle, which the other approaches do keep: that models are preserved when adding logical consequences, that is, if a formula φ is true in all "canonical" models of S, then each "canonical" model of S is also a model of $S \cup \{\varphi\}$. In the previous example, q is true in the only inflationary model $HI(\{q, r\})$ of S_5, but $HI(\{q, r\})$ is not an inflationary model of $S_5 \cup \{q\}$.

This may be the deeper reason why in spite of its attractive complexity properties (Section 8) the inflationary semantics is not very much being used in practice.[16]

[16] But it does integrate imperative constructs into logic, and there are indications that it may be useful for querying relational databases, see Section 14.5 in [2].

5.4 RDF Model Theory

5.4.1 Introduction to RDF. "The *Resource Description Framework* (*RDF*) is a language for representing information about 'resources' on the world wide web" [114].

Resources. How is the concept of "resource" to be understood? While RDF data representing information *about* resources is supposed to be accessible on the Web, the resources themselves do not necessarily have to be accessible. Thus, a "resource" is not necessarily a Web site or service. A resource is any (tangible or intangible) entity one represents information about.

Each resource is assumed to be uniquely identified by a *uniform resource identifier (URI)*, and everything identified by a URI is a resource.[17] A URI only plays the role of a unique identifier comparable to a bar code. In contrast to a *uniform resource locator (URL)*, a URI identifying a resource is not assumed to point to a Web page representing this resource – even though in practice this is often the case.

Triples and graphs. RDF data is represented in the form of *triples* consisting of a *subject*, a *predicate* and an *object* with the predicate representing a binary relation between the subject and the object. Instead of triples, one might just as well express RDF data in the form of atoms where the predicate is written as a binary relation symbol with the subject and the object as arguments.

Definition 196 (RDF syntax).

- *There are two classes of RDF symbols:*
  - *An* RDF Vocabulary $V = U \cup L$ *consists of two disjoint subsets: a set* U *of so-called* URI references *and a set* L *of so-called* literals*. Both URI references and literals are also called* names*.*
  - B *is a set disjoint from* V *containing so-called* blank nodes *or* b-nodes*.*
- *From these symbols the following complex structures may be formed:*
  - *An* RDF triple *or* RDF statement *in* V *and with blank nodes in* B *is an expression of the form* (s, p, o) *where*
    $s \in U \cup B$ *is called the* subject *of the triple,*
    $p \in U$ *is called the* predicate *or* property *of the triple,*
    $o \in U \cup B \cup L$ *is called the* object *of the triple.*
    An RDF triple is ground *if it contains no blank node.*
  - *An* RDF graph *in* V *and with blank nodes in* B *is a finite or infinite or empty subset of* $(U \cup B) \times U \times (U \cup B \cup L)$*, i.e., a set of RDF triples. An RDF graph is* ground *if the triples it contains are all ground.*

In the context of RDF (and throughout this subsection) the word *literal* is not used as in logic (see Definition 19), but as in programming languages, where *literal* means a textual representation of a value.

[17] Note the circularity of this definition.

In order to make the intention of the RDF notions easier to grasp, let us draw some rough analogies to first-order predicate logic.

An RDF vocabulary $V = U \cup L$ is analogous to a signature (Definition 3) and the blank nodes in B are analogous to variables, which belong to the logical symbols (Definition 2). The literals in L are analogous to constants, but with the intention that they should not be arbitrarily interpreted, but as strings or numbers or similar values. The URI references in U are also analogous to constants, but, being permitted in predicate position, also to relation symbols. In the terminology of Section 3, the signature is overloaded.

A triple is analogous to an atomic formula, but there are restrictions how it may be constructed: URI references from U may occur in all three positions, literals from L may only occur in object position, blank nodes from B may occur in subject or object position, but not in predicate position.

An RDF graph is analogous to a formula or set of formulas. A finite RDF graph corresponds to the existential closure of the conjunction of its triples. An infinite RDF graph corresponds to the appropriate generalisation, which does not have a direct counterpart in first-order predicate logic.

A slightly different analogy explains the terminology "RDF graph". Given a set of triples, the set of members of $U \cup L \cup B$ occurring in subject or object position can be regarded as "nodes", and each triple (s, p, o) can be regarded as a directed edge from s to o that is labelled with p. Thus an RDF graph corresponds to a graph in the mathematical sense.

Moreover, the URI references occurring in subject or object position of triples can also be regarded as pointers to the actual data representing the resource, which can be used in efficient implementations for a fast data access. This view illustrates the advantage of distinguishing between identifiers and resources in RDF (compare also Section 9.2.4) in spite of the slightly more involved formalism. The same distinction is made in object-oriented databases, and as pointed out in [23], storing RDF data as graphs in object-oriented databases may be preferable over storing them as relational tuples, as far as data retrieval and query answering are concerned.

5.4.2 Formal Semantics of RDF. The *formal semantics of RDF* [87] is specified similarly to that of first-order predicate logic by a Tarski style model theory. The concepts of interpretation and model serve to define a notion of logical consequence, or entailment, of an RDF graph.

However, there is not a single notion of an RDF interpretation, but several ones, each imposing additional constraints to the previous one:

- *Simple RDF Interpretations*
- *RDF Interpretations*
- *RDFS Interpretations*

Simple RDF interpretation is the basic notion: RDF interpretations and RDFS interpretations are simple RDF interpretations satisfying further conditions.

RDF interpretations give special meaning to predefined URI references such as `rdf:type`, `rdf:Property` and `rdf:XMLLiteral`: the URI reference `rdf:type` expresses a notion of instance relationship between the instance of a type and the type class itself (named in analogy to object-oriented type systems), `rdf:Property` expresses the type class of all predicates and `rdf:XMLLiteral` expresses the type class of all XML literals.[18]

RDFS interpretations are RDF interpretations that give special meaning to some additional URI references related to domain and range of properties or to subclass relationship (which, remarkably, may be cyclic).

Simple RDF Interpretations. It is convenient to distinguish between "untyped" (or "plain") and "typed literals". The former are syntactic representations of values that cannot be represented by other literals, such as the boolean value `true`; such literals are to be interpreted as "themselves", i.e., they are element of the domain. The latter are syntactic representations of values that may also be represented by other literals, such as the floating point number 0.11, which according to XML Schema, the reference for RDF scalar datatypes, can be represented by the literals `".11"^^xsd:float` and `"0.110"^^xsd:float`, among others.[19]

Definition 197 (Simple RDF interpretation of an RDF vocabulary). *A* simple RDF interpretation $I = (IR, IP, IEXT, IS, IL, LV)$ *of an RDF Vocabulary* $V = U \cup L^T \cup L^U$ *where* U *is a set of URIs,* L^T *a set of typed literals, and* L^U *a set of untyped literals (with* $L^T \cap L^U = \emptyset$*), is defined as follows:*

- IR *is a set of resources, called the* domain *of* I
- IP *is a set, called the* set of properties[20] *of* I
- $IEXT : IP \to \mathcal{P}(IR \times IR)$ *is a total function*
- $IS : U \to IR \cup IP$ *is a total function*
- $IL : L^T \to IR$ *is a total function*
- $LV \subseteq IR$*, called the* set of literal values *of* I*. Recall, that the untyped literals are interpreted as "themselves", i.e.,* $L^U \subseteq LV$*.*

The main difference between the above definition and the corresponding part of the definition of an interpretation for first-order predicate logic are as follows:

- In simple RDF interpretations, a predicate symbol in U is not directly associated with a binary relation, but with an arbitrary domain element that refers, by means of $IEXT$ to the actual binary relation. Note that no conditions are put on how such a "relation representative" may occur in relations. Thus, in a simple RDF interpretation, a domain element d may well occur in the relation it "represents". As a consequence, this additional level of "relation representatives" is no stringent deviation from Tarski model theory for first-order predicate logic.

[18] Following a widespread practice, `rdf` is the namespace prefix for the XML namespace of RDF and used here for conciseness instead of the actual namespace URI.
[19] Following a widespread practice, `xsd` stands for the namespace of XML Schema.
[20] The denomination "set of property representatives" would be more accurate.

Furthermore, this tie between the "relation representative" and the actual relation in a simple RDF interpretation is used in none of the other concepts of RDF semantics – RDF semantics keeps an account of all overloaded symbols, but does not make use of this book-keeping. Indeed, following [50], the classical model theory for RDF introduced below does not maintain this tie between "relation representative" and actual relation.
- Simple RDF interpretations of RDF *vocabularies* do not interpret blank nodes, since they are not part of an RDF vocabulary. Only the extension of simple RDF interpretations to RDF *graphs* (Definition 199) needs to consider blank nodes, and does so in the spirit of Tarski model theory for first-order predicate logic in the case of existentially quantified variables.

Thus, simple RDF interpretations of RDF vocabularies are, in spite of "stylistic" peculiarities, very much in line with Tarski model theory for first-order predicate logic.[21]

Simple RDF Interpretations of Ground RDF Graphs. The notion of simple RDF interpretation of RDF vocabularies introduced above is extended to RDF graphs such that a ground RDF graph is interpreted as the conjunction of its (ground) RDF triples, which are interpreted like ground atoms. Note, however, that this intuitive view may lead to infinite formulas (not considered in first-order predicate logic) as an RDF graph may contain infinitely many triples.

As a minor detail, RDF makes it possible to specify the language of an untyped literal `lit` using the widespread ISO 639 and IETF 1766 standardised language codes: `lit@en` means that `lit` is in English, `lit@de` means that `lit` is in German, etc.

Definition 198 (Simple RDF interpretations of ground RDF graphs). *Let $I = (IR, IP, IEXT, IS, IL, LV)$ be a simple RDF interpretation of an RDF vocabulary $V = U \cup L^T \cup L^U$ with set of URIs U, set of typed literals L^T and set of untyped literals L^U, where $L^T \cap L^U = \emptyset$.*

I is extended to ground RDF graphs as follows:

- $I(\texttt{lit}) = \texttt{lit}$ *(*untyped literal *in V)*
- $I(\texttt{lit@lang}) = \texttt{(lit, lang)}$ *(*untyped literal with language *in V)*
- $I(\texttt{lit\^\^type}) = IL(\texttt{lit})$ *(*typed literal *in V)*
- $I(\texttt{uri}) = IS(\texttt{uri})$ *(*URI *in V)*
- $I((s,p,o)) = \texttt{true}$ *(*ground RDF triple*)*
  iff $s, p, o \in V, \quad I(p) \in IP, \quad (I(s), I(o)) \in IEXT(I(p))$
- $I(G) = \texttt{true}$ *(*ground RDF graph*)*
  iff $I((s,p,o)) = \texttt{true}$ for all triples (s,p,o) in G.

[21] In the document introducing RDF Model theory, the notions "interpretation" and "denotation" do not seem to be clearly distinguished. Whereas the table of contents and the definitions suggest that denotations are defined on graphs and interpretations on vocabularies only, in later sections the parlance changes to "denotations of names" and to interpretations that assign truth values to graphs. In this tutorial, only the name "interpretation" is used both for vocabularies and for RDF graphs.

Note that the empty graph is true in all simple RDF interpretations.

Note furthermore that a ground RDF graph G is false in a simple RDF interpretation of a vocabulary V as soon as the subject, predicate or object of a triple in G does not belong to the vocabulary V. This is a slight (though entirely benign) deviation from the model theory of first-order predicate logic, which does not assign truth values to formulas composed of symbols from another vocabulary, or signature, than that of the interpretation considered.

Extended Simple RDF Interpretations of RDF Graphs. Simple RDF interpretation only apply to *ground* RDF graphs. This notion is extended to RDF graphs containing blank nodes such that an RDF graph is interpreted as the existential closure (the blank nodes representing variables) of the conjunction of its triples. As pointed out above, this intuition may lead to infinite formulas with, in presence of blank nodes, possibly even infinitely many variables, a case not considered in first-order predicate logic. The technique of the extension is to add to an interpretation I a mapping A that corresponds to a variable assignment.

Definition 199 (Extended simple RDF interpretation). *Let V be an RDF vocabulary and B a set of blank nodes with $V \cap B = \emptyset$. Furthermore, let $I = (IR, IP, IEXT, IS, IL, LV)$ be a simple RDF interpretation of V.*

A blank node assignment *for B in I is a total function $A : B \to IR$ mapping each blank node to a member of the domain of I.*

For any blank node assignment A the extended simple RDF interpretation *$[I + A]$ is defined as follows:*

- $[I + A](\texttt{bnode}) = A(\texttt{bnode})$ *(*bnode*)*
- $[I + A](\texttt{lit}) = I(\texttt{lit})$ *(*untyped literal *in V)*
- $[I + A](\texttt{lit@lang}) = I(\texttt{lit@lang})$ *(*untyped literal with language *in V)*
- $[I + A](\texttt{lit\^\^type}) = I(\texttt{lit\^\^type})$ *(*typed literal *in V)*
- $[I + A](\texttt{uri}) = I(\texttt{uri})$ *(*URI *in V)*
- $[I + A]((s, p, o)) = \texttt{true}$ *(*ground or non-ground RDF triple*)*
  iff $s, o \in V \cup B$, $p \in V$, $I(p) \in IP$ and $([I + A](s), [I + A](o)) \in IEXT(I(p))$
- $[I + A](G) = \texttt{true}$ *(*ground or non-ground RDF graph*)*
  iff $[I + A]((s, p, o)) = \texttt{true}$ for all triples (s, p, o) in G

The simple RDF interpretation I satisfies *a ground or non-ground RDF graph G, iff there exists a blank node assignment A with $[I + A](G) = \texttt{true}$.*

Definition 200 (Simple RDF entailment). *Let G_1 and G_2 be two arbitrary RDF graphs. The simple RDF entailment relation $\models_{simply}$ is defined as follows: $G_1 \models_{simply} G_2$, read G_1 simply entails G_2, if and only if all simple RDF interpretations satisfying G_1 also satisfy G_2.*

A Classical Model Theory for Simple RDF Entailment. The RDF model theory has been criticised for its non-standardness and several problems have been identified in the context of layering more expressive ontology or rule languages on top of RDF. In this section, we propose a model theory which is closer to classical

logics to characterise simple RDF entailment.[22] This characterisation shows that simple RDF entailment on finite RDF graphs is the same as entailment for first-order predicate logical formulas that

- are existential[23],
- are conjunctive[24],
- contain only binary relation symbols, and
- contain no function symbols of arity ≥ 1

For infinite RDF graphs it is an obvious generalisation of entailment for first-order predicate logic.

First, *classical RDF interpretations* and models are introduced, and the entailment relation $\models_{classical}$ is defined based on the notion of classical RDF models. Furthermore a one-to-one mapping between classical RDF interpretations and extended simple RDF interpretations is established. Finally we show that for two arbitrary RDF graphs G_1 and G_2 holds $G_1 \models_{simply} G_2$ if and only if $G_1 \models_{classical} G_2$.

Definition 201 (Classical RDF interpretation). *Let $V = U \cup B \cup L$ be an RDF Vocabulary where U is a set of URI references, B a set of blank node identifiers and Li a set of literals. We call elements in U also in analogy to first-order predicate logic constant symbols and distinguish the set of predicate symbols $U_P \subset U$. Note, that here a predicate symbol is* necessarily *also a constant symbol (in contrast to standard first-order predicate logic, cf. Section 3.1, where predicate and constant symbols are allowed but not required to overlap). A classical RDF interpretation $I = (D, M^c, M^p, M^l, A)$ consists of*

- *a domain of discourse $D \subset L^U$ where $L^U \subset L$ is the set of untyped literals in L,*
- *a total function M^c from URI references (constant symbols) in U to elements of the domain D,*
- *a total function M^p from predicate symbols in U_P to elements in $\mathcal{P}(D \times D)$*
- *a total function M^l from literals in L to elements of the domain D, plain literals are mapped to themselves,*
- *and a blank node assignment function A from B to $U \cup L$.*

The main deviation from standard first-order predicate logic interpretations are the specific treatment of literals and that predicate symbols are required to be a subset of the constant symbols. The latter is required for equivalence with the notion of extended simple RDF interpretation from [87] as explained in the previous section and could otherwise be dropped.

Definition 202 (Classical RDF model). *Let G be an RDF graph and $I = (D, M^c, M^p, M^l, A)$ a classical RDF interpretation of the vocabulary $V = U \cup$*

[22] Note that this theory does not characterise the other forms of RDF entailment such as non-simple RDF entailment or RDFS entailment.

[23] I.e. a formula the negation of which is universal

[24] I.e. the matrix of their prenex normal form is a conjunction of atoms

$B \cup L$ where U is a set of URIs, B is a set of blank node identifiers, and L is a set of literals. I is a model of G, if for every triple $(s, p, o) \in G$, s is in $U \cup B$, p is in U, o is in $U \cup B \cup L$, and the tuple $(M^c(s), M^c(o))$ is in the relation $M^p(p)$ interpreting the predicate symbol p.

We define the *classical RDF entailment* relation $\models_{classical}$ in the expected way: A graph G_1 entails a graph G_2 if and only if every classical model of G_1 is also a classical model of G_2.

Definition 203 (Classical interpretation corresponding to an extended simple RDF interpretation). *Let $V = U \cup B \cup L^T \cup L^U$ be an RDF vocabulary where U is a set of URI references, B a set of blank node identifiers, L^T a set of typed, and L^U a set of untyped literals. Let $I = [(IR, IP, IEXT, IS, IL, LV) + A]$ be an extended simple RDF interpretation of V. The* corresponding classical interpretation *$class(I) = (D, M^c, M^p, M^l, A')$ of the same vocabulary V is defined as follows:*

- *$D := IR$*
- *$M^c(c) := IS(c)$ for all $c \in U$*
- *$M^p(p) := IEXT(IS(p))$ for all $p \in P$ such that $IEXT(IS(p))$ is defined.*
- *$M^l(l) := IL(l)$ for all typed literals in L^T*
- *$A' := A$*

Lemma 204. *Let G be an RDF graph, and I an extended simple RDF interpretation of the vocabulary V. Let $class(I)$ be the corresponding classical interpretation of the vocabulary V. If $I \models_{simply} G$ then $class(I) \models_{classical} G$.*

Proof. Let $I = [(IR, IP, IEXT, IS, IL, LV) + A]$ and $class(I) = (D, M^c, M^p, M^l, A)$. Let $V = U \cup B \cup L^T \cup L^U$ where U is the set of URIs, B the set of blank node identifiers, L^T the set of typed literals, and L^U the set of untyped literals of V. It suffices to show that for any triple (s, p, o) in G the following conditions hold:

- *$s \in U \cup B$: (s, p, o) is in G, I is a model of G, hence $s \in U \cup B$: by definition*
- *$p \in U$: by definition*
- *$o \in U \cup B \cup L$: by definition*
- *$(M^c(s), M^c(o)) \in M^p(p)$: (s, p, o) is in G, I is a model of G, therefore the tuple $(IS(s), IS(o))$ is in $IEXT(IS(p))$, hence $(M(s), M(o)) \in M^p(p)$.* □

In the following definition $dom(f)$ denotes the domain of a function f.

Definition 205 (Extended simple RDF interpretation corresponding to a classical RDF interpretation). *Let $V = U \cup B \cup L^T \cup L^U$ be an RDF vocabulary where U is a set of URI references, B a set of blank node identifiers, L^T a set of typed literals, and L^U a set of untyped literals. Let $I_c = (D, M^c, M^p, M^l, A)$ be a classical RDF interpretation over V. The* extended simple RDF interpretation *$RDF(I_c) = [(IR, IP, IEXT, IS, IL, LV) + A']$* corresponding to *$I_c$ is defined as follows:*

- *$IR := D$*

- $IP := \{M^c(p) \mid p \in dom(M^p)$ *and* $p \in dom(M^c)\ \}$
- $IEXT : IP \to (IR \times IR), IEXT(M^c(p)) := M^p(p)$ *for all* p *in* U *such that both* M^c *and* M^p *are defined on* p.
- $IS : U \to IR \cup IP$, $IS(u) := M^c(u)$ *for all* c *in* U.
- $IL := M^l$
- LV *is the set of untyped literals* L^U
- $A'(x) := A(x)$ *for all* $x \in B$

Lemma 206. *Let* G *be an RDF graph,* I_c *a classical interpretation of the vocabulary* V*, and* $RDF(I_c)$ *its corresponding RDF interpretation. If* $I_c \models_{classical} G$ *then* $RDF(I_c)$ *is a simple RDF model of* G.

Proof. We have to show that $RDF(I_c) \models_{simply} G$. *Hence* $RDF(I_c) \models_{simply} t$ *must be true for every triple* t *in* G. *Let* $t := (s, p, o)$ *be a triple in* G. *Then* $I_c \models_{classical} t$ *is true by assumption. Therefore* s *is in* $C \cup B$, p *is in* U *and* o *is in* $U \cup B \cup L$. *Moreover* $(M^c(s), M^c(o)) \in M^p(p)$. *Hence* $(IS(s), IS(o)) \in IEXT(IS(p))$, *and thus* $RDF(I_c) \models_{simply} G$. □

Lemma 207. *Let* I_s *be an extended simple RDF interpretation and* I_c *a classical RDF interpretation of the same vocabulary. Then* $RDF(class(I_s)) = I_s$ *and* $class(RDF(I_c)) = I_c$.

Proof. This lemma is a direct consequence of the Definitions 203 and 205. □

From Lemmata 204, 206 and 207 we can immediately conclude the following corollary:

Corollary 208 (Equivalence of classical *RDF entailment* and simple RDF entailment). *Let* G_1 *and* G_2 *be two RDF graphs.* $G_1 \models_{simply} G_2$ *if and only if* $G_1 \models_{classical} G_2$.

In [50], a more involved reformulation of RDF interpretation and entailment is presented that also extends to RDFS interpretations and entailment. However, for consistency with the rest of this article, we have chosen the above presentation.

6 Operational Semantics: Positive Rule Sets

6.1 Semi-naive Evaluation of Datalog Programs

The fixpoint semantics of a positive logic program P directly yields an operational semantics based on canonical forward chaining of the immediate consequence operator $\mathbf{T}_P$ introduced in Section 5 (Definition 153) until the least fixpoint is reached.

Let us quickly recapitulate the definition of the fixpoint of a datalog program P given the example program in Listing 6_1.

Listing 6_1. An example program for fixpoint calculation

```
feeds_milk(betty).
lays_eggs(betty).
has_spines(betty).

monotreme(X) ← lays_eggs(X), feeds_milk(X).
echidna(X) ← monotreme(X), has_spines(X).
```

The *intensional* predicate symbols of a datalog program P are all those predicate symbols that appear within the head of a rule, as opposed to the *extensional* predicate symbols which appear only in the bodies of rules of a program. With this definition `feeds_milk`, `lays_eggs` and `has_spines` are extensional predicate symbols, whereas `monotreme` and `echidna` are intensional predicate symbols. The set of all extensional and intensional predicate symbols of a datalog program P (denoted $ext(P)$ and $int(P)$ respectively) is called the schema of P. An *instance* over a schema of a logic program is a set of sets of tuples $s_1, \ldots, s_k$, where each set of tuples $s_i, 1 \leq i \leq k$ is associated with a predicate symbol p in the schema and s_i is the extension of p. The set of base facts of the program 6_1 corresponds to an instance over the extensional predicate symbols, where the set $\{betty\}$ is the set associated with each of the symbols `feeds_milk`, `lays_eggs` and `has_spines`.

Based on these definitions the semantics of a logic program P is defined as a mapping from extensions over $ext(P)$ to extensions over $int(P)$. There are several possibilities to define this function. The fixpoint semantics uses the *immediate consequence operator* $\mathbf{T}_P$ for this aim.

Given a datalog program P and an instance I over its schema $sch(P)$, an atom A is an *immediate consequence* of P and I if it is either already contained in I or if there is a rule $A \leftarrow cond_1, \ldots, cond_n$ in P where $cond_i \in I \ \forall 1 \leq i \leq n$. The immediate consequence operator $\mathbf{T}_P(I)$ maps an instance over the schema $sch(P)$ to the set of immediate consequences of P and I.

A fixpoint over the operator $\mathbf{T}_P$ is defined as an instance I such that $\mathbf{T}_P(I) = I$. It turns out that any fixpoint for a datalog program is a model of the (conjunction of clauses of the) program. Furthermore, the model-theoretic semantics $P(I)$ of a logic program P on an input instance I, which is defined as the minimum model of P that also contains I, is the minimum fixpoint of $\mathbf{T}_P$.

As mentioned above, this fixpoint semantics for datalog programs, which may be extended to non-datalog rules[69], gives directly rise to a constructive algorithm to compute the minimum model of a program.

Consider again Listing 6_1. The set of immediate consequences of this program with the initial instance[25] $I_0 = \{\{\}_{f\_m}, \{\}_{l\_e}, \{\}_{h\_s}, \{\}_{mon}, \{\}_{ech}\}$ is $I_1 := \mathbf{T}_P(I_0) = \{\{betty\}_{f\_m}, \{betty\}_{l\_e}, \{betty\}_{h\_s}, \{\}_{mon}, \{\}_{ech}\}$. The second application of the fixpoint operator yields $I_2 := \mathbf{T}_P^2(I_0) = \{\{betty\}_{f\_m}, \{betty\}_{l\_e}, \{betty\}_{h\_s}, \{betty\}_{mon}, \{\}_{ech}\}$.

[25] the predicate symbols in subscript position indicate that the first set is the extension of `feeds_milk`, the second one the one of `lays_eggs`, and so on.

I_3 is defined analogously and the extension of `echidna` is set to $\{betty\}$. Finally the application of $\mathbf{T}_P$ to I_3 does not yield any additional facts such that the condition $I_3 = I_4$ is fulfilled, and the fixpoint is reached.

The above procedure can be implemented with the pseudo-algorithm in Listing 6_2, which is called *naive evaluation* of datalog programs, because for the computation of I_i all elements of I_{i-1} are recomputed. As suggested by its name, the function `ground_facts` returns all the ground facts of the program which is to be evaluated. The function `instantiations` takes as a first argument a rule R, which may contain variables, and as a second argument the set of facts I_{i-1} which have been derived in the previous iteration. It finds all instantiations of the rule R which can be satisfied with the elements of I_{i-1}.

Listing 6_2. Naive evaluation of a datalog program P

```
I_0 := ∅
I_1 := ground_facts(P)
i := 1
while I_i ≠ I_{i-1} do
  i := i + 1
  I_i := I_{i-1}
  while (R = Rules.next())
    Insts := instantiations(R, I_{i-1})
    while (inst = Insts.next())
      I_i := I_i ∪ head(inst)
return I_i
```

The central idea underlying the so-called *semi-naive* evaluation of datalog programs is that all facts that can be newly derived in iteration i must use one of the facts that were newly derived in iteration $i-1$ – otherwise they have already been derived earlier. To be more precise, the rule instantiations that justify the derivation of a new fact in iteration i must have a literal in their rule body which was derived in iteration $i-1$. In order to realize this idea one must keep track of the set of newly derived facts in each iteration. This method is also called *incremental forward chaining* and is specified by Listing 6_3. In line 2 the increment `Ink` is initialized with all facts of the datalog program. In line 4 the set `Insts` of instantiations of rules that make use of at least one atom of the increment Ink is computed at the aid of the function `instantiations`. The function `instantiations` does not yield ground rules that are justified by the set `KnownFacts` only, such that the call `instantiations({ p(a), q(a) }, { })` for a program consisting of the rule `r(x)` ← `p(X), q(X)` would *not* yield the instantiation $i_1 :=$ `r(a)` ← `p(a), q(a)`. In contrast, i_1 would be returned by the call `instantiations({ p(a) }, { q(a) })` with respect to the same program.

Once these fresh rule instantiations have been determined, the distinction between facts in the increment and older facts is no longer necessary, and the two sets are unified (line 5). The new increment of each iteration is given by the heads of the rule instantiations in `Insts`.

Listing 6_3. Semi-naive evaluation of a datalog program P

```
1 KnownFacts := ∅
2 Ink := { Fact | (Fact ← true) ∈ P }
3 while (Ink ≠ ∅)
4   Insts := instantiations(KnownFacts, Ink)
5   KnownFacts := KnownFacts ∪ Ink
6   Ink := heads(Insts)
7 return KnownFacts
```

Although the semi-naive evaluation of datalog programs avoids a lot of redundant computations that the naive evaluation performs, there are still several ways of optimizing it.

- In the case that besides a program P also a query q is given, it becomes apparent that a lot of computations, which are completely unrelated to q, are carried out. This is a general problem of forward chaining algorithms when compared to backward chaining. However, it is possible to write logic programs that, also when executed in a forward-chaining manner, are in a certain sense goal-directed. In fact it is possible to transform any datalog program P and query q into a logic program P' such that the forward chaining evaluation of P' only performs computations that are necessary for the evaluation of q. In Section 6.2 these so-called *magic templates transformations* are presented.
- A second source of inefficiency is that in each iteration i, it is tested from scratch whether the body of a rule is satisfied. It is often the case that a rule body completely satisfied in iteration i was almost completely satisfied in iteration $i-1$, but the information about which facts contributed to the satisfaction of rule premises in iteration $i-1$ must be recomputed in iteration i. It is therefore helpful to store complete and partial instantiations of rules during the entire evaluation of the program.
- Storing partial instantiations of rule bodies gives rise to another optimization if the rules of the program share some of their premises. In this case, the partial rule instantiations are shared among the rules. Both this and the previous optimization are realized by the Rete algorithm, which is introduced in Section 6.3.

6.2 The Magic Templates Transformation Algorithm

The magic templates algorithm [51] is a method of introducing a goal directed search into a forward chaining program, thereby benefiting both from the termination of forward chaining programs and from the efficiency of goal directed search. It is important to emphasize that the evaluation of the transformed program is performed in the ordinary forward chaining way or can be combined with the semi-naive algorithm as described above.

The magic templates rewriting transforms a program P and a query q in two steps: a transformation of the program into an adorned version, and a rewriting of the adorned program into a set of rules that can be efficiently evaluated with a bottom up strategy.

6.2.1 Adornment of Datalog Programs. In the first step, the program is rewritten into an *adorned* version according to a *sideways information passing strategy*, often abbreviated sip.

A sideways information passing strategy determines how variable bindings gained from the unification of a rule head with a goal or sub-goal are passed to the body of the rule, and how they are passed from a set of literals in the body to another literal. The ordinary evaluation of a Prolog program implements a special sideways information passing strategy, in which variable bindings are passed from the rule head and all previously occurring literals in the body to the body literal in question. There are, however, many other sips which may be more convenient in the evaluation of a datalog or Prolog program. In this survey, only the standard Prolog sip is considered, and the interested reader is referred to [18] for a more elaborate discussion of sideways information passing strategies.

The construction of an adorned program is exemplified by the transformation of the transitive closure program in Listing 6_4 together with the query `t(a,Answer)` into its adorned version in Listing 6_5. In order to better distinguish the different occurrences of the predicate `t` in the second rule, they are labeled `t-1`, `t-2` and `t-3`, but they still denote the same predicate.

Listing 6_4. Transitive closure computation

```
t(X,Y) ← r(X, Y).
t-3(X,Z) ← t-1(X, Y), t-2(Y, Z).

r(a, b).
r(b, c).
r(c, d).
```

When evaluated in a backward chaining manner, the query Q :=`t(a,Answer)` is first unified with the head of the first rule, generating the binding `X=a` which is passed to the rule body. This sideways information passing can be briefly expressed by `t` $\hookrightarrow_X$ `r`. The query$Q$ is also unified with the head of the second rule, generating once more the binding `X=a`, which would be used to evaluate the literal `t-1(X,Y)` by a Prolog interpreter. In the remaining evaluation of the second rule, the binding for `Y` computed by the evaluation of `t-1(X,Y)` is passed over to the predicate `t-2`. This can be briefly expressed by the sips `t-3` $\hookrightarrow_X$ `t-1` and `t-1` $\hookrightarrow_Y$ `t-2`.

From this information passing strategy an adorned version of Listing 6_4 can be derived. Note that all occurrences of the predicate `t` (and its numbered versions) are evaluated with the first argument bound and the second argument free when Q is to be answered. In the magic templates transformation it is important to differentiate between different call-patterns for a predicate. This is where adornments for predicates come into play. An adornment a for a predicate p of arity n is a word consisting of n characters which are either 'b' (for bound) or 'f' (for free). Since the first argument of `t` is always bound in the program and the second argument is always free, the only adornment for `t` is bf. Since

the evaluation of literals of *extensional* predicates amounts to simply looking up the appropriate values, adornments are only introduced for *intensional* predicate symbols.

It is interesting to note that the choice of the information passing strategy strongly influences the resulting adorned program. In the case that one chooses to evaluate the literal `t-2` before `t-1`, both arguments of `t-2` would be unbound yielding the sub-query $\texttt{t-2}^{ff}$, and thus an additional adorned version of the second rule would have to be introduced for this sub-query. This additional adorned rule would read $\texttt{t-3}^{ff}\texttt{(X,Z)} \leftarrow \texttt{t-1}^{fb}\texttt{(X, Y), t-2}^{ff}\texttt{(Y, Z).}$. For the sake of simplicity, the following discussion refers to the shorter version depicted in Listing 6_5 only.

Listing 6_5. The adorned version of the program in Listing 6_4

```
t^bf(X,Y) ← r(X, Y).
t-3^bf(X,Z) ← t-1^bf(X, Y), t-2^bf(Y, Z).

r(a, b).
r(b, c).
r(c, d).
```

6.2.2 Goal-Directed Rewriting of the Adorned Program. Given an adorned datalog program P^{ad} and a query q, the general idea of the magic templates rewriting is to transform P^{ad} into a program P^{ad}_m in a way such that all sub-goals relevant for answering q can be computed from additional rules in P^{ad}_m. Slightly alternated versions of the original rules in P^{ad} are included in P^{ad}_m, the bodies of which ensure that the rule is only fulfilled if the head of the rule belongs to the set of relevant sub-goals.

Hence the magic template transformation generates two kinds of rules: The first set of rules controls the evaluation of the program by computing all relevant sub-goals from the query, and the second set of rules is an adapted version of the original program with additional premises in the bodies of the rules, which ensure that the rules are only evaluated if the result of the evaluation contributes to the answer of q.

The functioning of the magic templates transformation and the evaluation of the transformed program is again exemplified by the transitive closure computation in Listings 6_4 and 6_5.

1. In a first step, a predicate $\texttt{magic\_p}^a$ of arity $nb(p^a)$ is created for each adorned predicate $\texttt{p}^a$ that occurs in the adorned program, where $nb(p^a)$ denotes the number of bound arguments in p^a – in other words the number of 'b's in the adornment a. Thus for the running example, the predicate $\texttt{magic\_t}^{bf}$ with arity one is introduced. The intuition behind magic predicates is that their extensions during bottom-up evaluation of the program, often referred to as *magic sets*, contain all those sub-goals that need to be computed for p^a. In the transitive closure example, the only initial instance of $\texttt{magic\_t}^{bf}$ is

`magic_t`^{bf}`(a)`, which is directly derived from the query `t(a,Answer)`. This initial magic term is added as a seed[26] to the transformed program in Listing 6_6.

2. In a second step, rules for computing sub-goals are introduced reflecting the sideways information passing within the rules. Let r be a rule of the adorned program P^{ad}, let h^a be the head of r, and $l_1 \ldots l_k$ the literals in the body of r. If there is a query that unifies with the head of the rule, if queries for $l_1 \ldots l_i$ $(i < k)$ have been issued and if they have been successful, the next step in a backward chaining evaluation of P^{ad} would be to pursue the sub-goal l_{i+1}. Thus a control rule $l_{i+1} \leftarrow$ `magic_h`^{a}`,` $l_1 \ldots l_i$ is included in P^{ad}_m. For the running example the rule `magic_t`^{bf}`(Y)` $\leftarrow$ `magic_t`^{bf}`(X), t(X,Y)` is added.
3. In a third step, the original rules of P^{ad} are adapted by adding some extra conditions to their bodies in order to evaluate them only if appropriate sub-goals have already been generated by the set of control rules. Let r be a rule in P^{ad} with head h^a and with literals $l_1, \ldots, l_n$. r shall only be evaluated if there is a sub-goal $magic\_h^a$ for the head, and if there are sub-goals for each of the derived predicates of the body. For the adorned version of the transitive closure program (Listing 6_5) both the first and the second rule must be rewritten. Since there is no derived predicate in the first rule, the only literal which must be added to the rule body is `magic_t`$^{bf}(X)$, yielding the transformed rule `t`^{bf}`(X,Y)` $\leftarrow$ `magic_t`$^{bf}(X)$`, r(X,Y)`. With the second rule having two derived predicates in the rule body, one might expect that three additional magic literals would have to be introduced in the rule body. But since `t-1` and `t-3` have the same adornment and the same variables for their bound arguments, they share the same magic predicate.

The evaluation of the magic transitive closure program is presented in Listing 6_7 for the goal `t(a,Answer)`. Note that in contrast to the naive and semi-naive bottom-up algorithms, only those facts are derived, which are potentially useful for answering the query. In particular, the facts `r(1, 2)`, `r(2, 3)`, and `r(3, 1)` are never used. Moreover the sub-goal `r(d,X)` corresponding to the magic predicate `magic_t`^{bf}`(d)` is never considered.

Listing 6_6. The magic templates transformation of the program in Listing 6_5 for the query `t(a,X)`

```
1  magic_t^bf(Y) ← magic_t^bf(X), t(X,Y).
2  t^bf(X,Y) ← magic_t^bf(X), r(X,Y).
3  t-3^bf(X,Z) ← magic_t^bf(X), t-1^bf(X,Y), magic_t^bf(Y), t-2^bf(Y,Z).
4  magic_t^bf(a). // the seed
5
6  r(a, b).  r(b, c).  r(c, d).
7  r(1, 2).  r(2, 3).  r(3, 1).
```

[26] it is called the *seed*, because all other magic terms are directly or indirectly derived from it.

Listing 6_7. Evaluation of program 6_6

```
t(a,b)       // derived by the seed and rule 2
magic_t^bf(b) // derived by the seed, t(a,b) and rule 1
t(b,c)       // derived by magic_t^bf(b), and rule 2
magic_t^bf(c) // derived by magic_t^bf(b), t(b,c) and rule 1
t(c,d)       // derived by magic_t^bf(c) and rule 2
t(a,c)       // derived by rule 3
t(a,d)       // derived by rule 3
t(b,d)       // derived by rule 3
```

6.3 The Rete Algorithm

The Rete algorithm [72, 56] was originally conceived by Charles L. Forgy in 1974 as an optimized algorithm for inference engines of rule based expert systems. Since then several optimizations of Rete have been proposed, and it has been implemented in various popular expert systems such as Drools, Soar, Clips, JRules and OPS5.

The Rete algorithm is used to process rules with a conjunction of conditions in the body and one or more actions in the head, that are to be carried out when the rule fires. These rules are stored in a so-called *production memory*. The other type of memory that is used by the Rete algorithm is the *working memory*, which holds all the facts that make up the current configuration the rule system is in. A possible action induced by a rule may be the addition of a new fact to the working memory, which may itself be an instance of a condition of a rule, therefore triggering further actions to be carried out in the system.

Avoiding redundant derivations of facts and instances of rule precedents, the Rete algorithm processes production rules in a so-called *Rete network* consisting of *alpha-nodes*, *beta-nodes*, *join-nodes* and *production-nodes*.

Figure 1 illustrates the way a Rete network is built and operates. It serves as an animal classification system relying on characteristics such as `has wings`, `has spikes`, `is poisonous`, etc. The example rules exhibit overlapping rule bodies (several atomic conditions such as `X lays eggs` are shared among the rules).

For each atomic condition in the body of a rule, the Rete network features one alpha-node containing all the elements of the working memory that make this atomic condition true. Alpha-nodes are distinguished by shaded rectangles with round corners in Figure 1. Although the same atomic condition may occur multiple times distributed over different production rules, only one single alpha node is created in the Rete network to represent it. Therefore the condition `X lays eggs`, which is present in the conditions of all rules except for `p2`, is represented by a single alpha-node in Figure 1.

While alpha-nodes represent single atomic conditions of rule bodies, beta-nodes stand for conjunctions of such conditions, and hold sets of tuples of working memory elements that satisfy them. Beta-nodes are depicted as ovals with white background in Figure 1.

In contrast to alpha and beta nodes, join nodes do not hold tuples of or single working memory elements, but serve computation purposes only. For each rule in the rule system, there is one production node (depicted as rectangles with grey background in Figure 1) holding all the tuples of working memory elements that satisfy all the atomic conditions in its body.

Alpha- and beta-nodes are a distinguishing feature of Rete in that they remember the state of the rule system in a fine grained manner. With beta-nodes storing instantiations of (partial) rule bodies, there is no need of reevaluating the bodies of all rules within the network in the case that the working memory is changed.

Besides *storing* derived facts and instantiations of (partial) rule premises, the Rete network also allows information *sharing* to a large extent. There are two ways that information is shared among rules in the network. The first way concerns the alpha-nodes and has already been mentioned above. If an atomic condition (such as `X feeds milk`) appears within more than one rule, this alpha node is shared among both rules. Needless to say, this is also the case if both conditions are variants (equivalent modulo variable renaming) of each other. The second way that information is shared within the Rete network is by sharing partial rule instantiations between different rules. In Figure 1, the conjunction of atomic conditions `(X lays eggs), (X feeds milk)` is common to the rules `p3`, `p4` and `p5`. In a Rete network, instantiations of these partial rule bodies are computed only once and saved within a beta node which is connected (possibly via other beta nodes) to the production nodes of the affected rules.

6.4 Basic Backward Chaining: SLD-Resolution

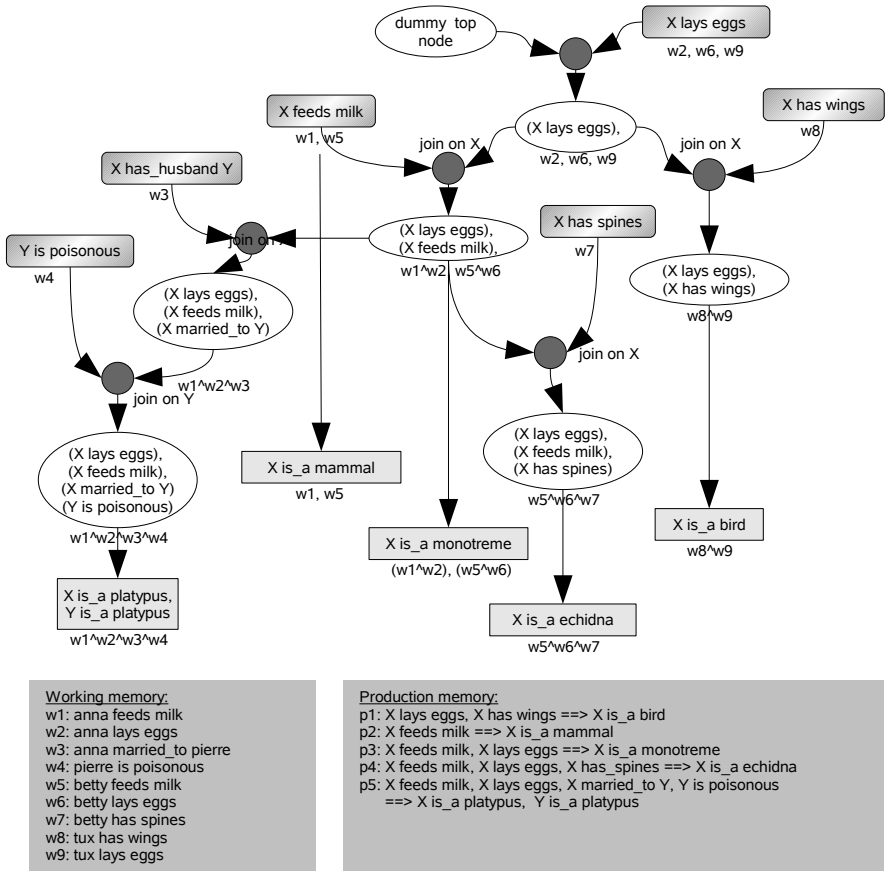
Resolution proofs are refutation proofs, i.e. they show the unsatisfiability of a set of formulas. As it holds that the set of formulas $P \cup \{\neg\varphi\}$ is unsatisfiable iff $P \models \varphi$, resolution may be used to determine entailment (compare Theorem 35). Observe that a goal $\leftarrow a_1, \ldots, a_n$ is a syntactical variant of the first order sentence $\forall x_1 \ldots x_m(\bot \leftarrow a_1 \wedge \ldots \wedge a_n)$ where $x_1, \ldots, x_m$ are all variables occurring in $a_1, \ldots a_n$. This is equivalent to $\neg\exists x_1 \ldots x_m(a_1 \wedge \ldots \wedge a_n)$. If we use SLD-resolution[27] to show that a logic program P and a goal $\leftarrow a_1, \ldots, a_n$ are unsatisfiable we can conclude that $P \models \exists x_1 \ldots x_m(a_1 \wedge \ldots \wedge a_n)$.

Definition 209 (SLD Resolvent). *Let C be the clause $b \leftarrow b_1, \ldots, b_k$, G a goal of the form $\leftarrow a_1, \ldots, a_m, \ldots, a_n$, and let θ be the mgu of a_m and b. We assume that G and C have no variables in common (otherwise we rename the variables of C). Then G' is an* SLD resolvent of G and C using θ *if G' is the goal $\leftarrow (a_1, \ldots a_{m-1}, b_1, \ldots b_k, a_{m+1}, \ldots a_n)\theta$.*

Definition 210 (SLD Derivation). *A* SLD derivation *of $P \cup \{G\}$ consists of a sequence $G_0, G_1, \ldots$ of goals where $G = G_0$, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that G_{i+1} is*

[27] SLD is an acronym for Selected Literal Definite Clause

Fig. 1 A Rete Network for Animal Classification



a resolvent from G_i and C_{i+1} using θ_{i+1}. An SLD-refutation *is a finite SLD-derivation which has the empty goal as its last goal.*

Definition 211 (SLD Tree). *An* SLD tree T w.r.t. a program P and a goal G *is a labeled tree where every node of T is a goal and the root of T is G and if G is a node in T then G has a child G' connected to G by an edge labeled (C, θ) iff G' is an SLD-resolvent of G and C using θ.*

Let P be a definite program and G a definite goal. A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition of $\theta_1, \ldots \theta_n$ to the variables occurring in G, where $\theta_1, \ldots \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$.

Observe that in each resolution step the selected literal a_m and the clause C are chosen non-deterministically. We call a function that maps to each goal

one of its atoms a *computation rule*. The following proposition shows that the result of the refutation is independent of the literal selected in each step of the refutation.

Proposition 212 (Independence of the Computation Rule). *[111] Let P be a definite Program and G be a definite goal. Suppose there is an SLD-refutation of $P \cup \{G\}$ with computed answer θ. Then, for any computation rule R, there exists an SLD-refutation of $P \cup \{G\}$ using the atom selected by R as selected atom in each step with computed answer θ' such that $G\theta$ is a variant of $G\theta'$.*

The independence of the computation rule allows us to restrict the search space: As a refutation corresponds to a branch of in an SLD-tree, to find all computed answers we need to search all branches of the SLD-tree. The independence of the computation rule allows us to restrict our search to branches constructed using some (arbitrary) computation rule.

Example 213. Consider the logic program 6_8 with query $q = \leftarrow$ `t(1,2)`:

Listing 6_8. Transitive Closure

```
t(x,y) ← e(x,y).
t(x,y) ← t(x,z), e(z,y).
e(1,2) ← .
e(2,1) ← .
e(2,3) ← .
← t(1,2) .
```

An SLD-tree for program 6_8 and q is shown in the following figure. We label the edges of an SLD tree with the number of a rule instead of a rule. We denote by (n') the rule number n where each variable x occurring in rule n is replaced by x'.

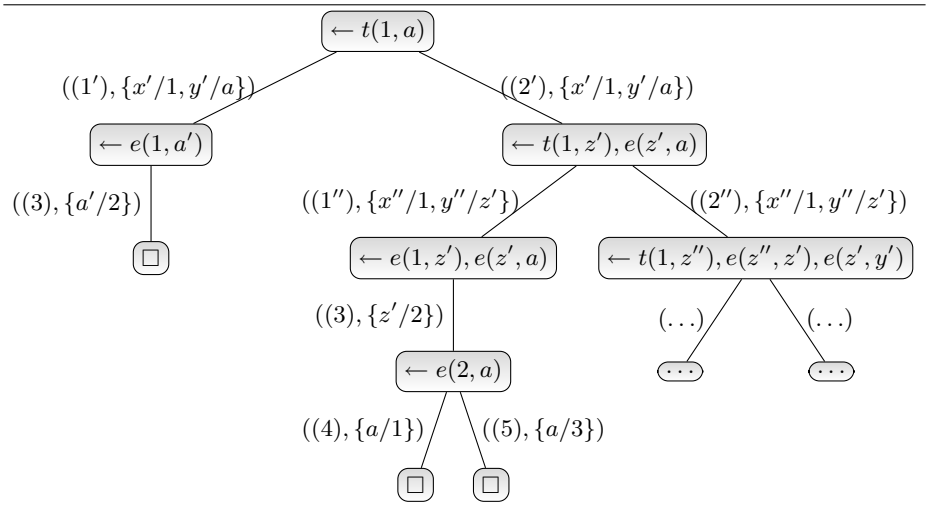
If we want to compare the operational semantics of a program P to its declarative semantics we need a declarative notion of an answer of program P. A *correct answer* for a program P and goal G is a substitution θ such that $P \models G\theta$. Using this notion we can define the soundness and completeness of logic programming.

Proposition 214 (Soundness and Completeness of Logic Programming). *[111] Let P be a program and let Q be a query. Then it holds that*

- *every computed answer of P and G is a correct answer and*
- *for every correct answer σ of P and G there exists a computed answer θ such that θ is more general that σ.*

Observe that to find a computed answers of a program P and goal G operationally one has to visit the leaf of a finite branch in the SLD-tree w.r.t. P and G. The order in which we visit these nodes is not determined by the definition of an SLD-refutation. We call such an order a *search strategy*. An *SLD-procedure*

Fig. 2 An SLD tree for program 6_8



is a deterministic algorithm which is an SLD-resolution constrained by a computation rule and a search strategy.

As SLD-trees are infinite in general, the completeness of an SLD-procedure depends on the search strategy. To be complete, an SLD-procedure must visit every leaf of a finite branch of an SLD-tree within a finite number of steps. A search strategy with this property is called *fair*. Obviously not every search strategy is fair. For example the depth first search strategy used by Prolog is not fair. An example of a fair search strategy is breath first search.

6.5 Backward Chaining with Memorization: OLDT-Resolution

As stated in the previous section not every search strategy is complete. This is due to the fact that an SLD-tree is infinite in general. As we only consider finite programs, an SLD-tree may only be infinite if it has an infinite branch. As a branch in an SLD-tree corresponds to an SLD-derivation we denote a branch as $[G_1, G_2, \ldots]$ where $G_1, G_2 \ldots$ are the goals of the corresponding derivation.

A branch $B = [G_1, G_2, \ldots]$ in an SLD-tree may be infinite if there is a subsequence $[G_{i_1}, G_{i_2}, \ldots]$ ($i_j < i_k$ if $j < k$) of B such that

- for all $j, k \in \mathbb{N}$ G_{i_j} and G_{i_k} contain an equal (up to renaming of variables) atom or
- for all $j \in \mathbb{N}$ G_{i_j} contains an atom which is a real instance of an atom in $G_{i_{j+1}}$.

Non-termination due to the first condition is addressed by a evaluation technique called *tabling* or *memorization*. The idea of tabling is the idea of dynamic programming: store intermediate results to be able to look these results

up instead of having to recompute them. In addition to the better termination properties, performance is improved with this approach.

The OLDT algorithm [150] is an extension of the SLD-resolution with a left to right computation rule. Like SLD-resolution, it is defined as a non-deterministic algorithm.

A subset of the predicate symbols occurring in a program are classified as *table predicates*. A goal is called a *table goal* if its leftmost atom has a table predicate. Solutions to table goals are the intermediate results that are stored. Table goals are classified as either *solution goals* or *look-up goals*. The intuition is that a solution goal 'produces' solutions while a look-up goal looks up the solutions produced by an appropriate solution goal.

An *OLDT-structure* (T, T_S, T_L) consists of an SLD-tree T and two tables, the solution table T_S and the look-up table T_L. The *solution table* T_S is a set of pairs $(a, T_S(a))$ where a is an atom and $T_S(a)$ is a list of instances of a called the *solutions* of a. The *look-up table* T_L is a set of pairs $(a, T_L(a))$ where a is an atom and p is a pointer pointing to an element of $T_S(a')$ where a is an instance of a'. T_L contains one pair $(a, T_L(a))$ for an atom a occurring as a leftmost atom of a goal in T.

The *extension of an OLDT structure* (T, T_S, T_L) consists of three steps:

1. a resolution step,
2. a classification step, and
3. a table update step.

In the resolution step a new goal is added to the OLDT-tree, in the classification step this new goal is classified as either non-tabled goal or solution goal or look-up goal and in the table update step the solution table and the update table are updated. While step one is equal for non-tabled and solution goals, step two and three are equal for tabled nodes while there is nothing to do in these steps for non-tabled nodes.

Let (T, T_S, T_L) be an OLDT structure and $G = \leftarrow a_1, \ldots, a_n$ a goal in T. If G is a non-tabled goal or a solution goal then in the resolution step a new goal G' is added to T which is connected to G with an edge labeled (C, θ) where G' is the SLD-resolvent of G and C using θ. If G is a look-up node then in the resolution step the new node G' is added to T with an edge labeled $(a \leftarrow, \theta)$ where a is the atom in the solution table that the pointer $T_L(a_1)$ points to and the substitution θ is the mgu of a and a_1. Finally the pointer $T_L(a_1)$ is set to point to the next element of the list it points to.

In the classification step the new goal G' is classified as a non-table goal if its leftmost atom is not a table predicate and a table goal otherwise. If G' is a table goal then G' is classified as a look-up node if there is a pair $(a, T_S(a))$ in the solution table and a is more general than the leftmost atom a' of G'. In this case a new pair (a', p) is added to the look-up table and p points to the first element of $T_S(a)$. If G' is not classified as a look-up node then it is classified as a solution node and a new pair $(a', [])$ is added to the solution table.

In the table update step new solutions are added to the solution table. Recall that the problem we want to tackle here is the recurrent evaluation of equal (up

to renaming of variables) atoms in goals. Therefore the 'solutions' we want to store in the solution table are answers to an atom in a goal.

In SLD-resolution the term answer is defined only for goals. This notion can be extended to atoms in goals in the following way. OLDT-resolution uses a left to right computation rule. If the derivation of a goal $G = \leftarrow a_1, \ldots, a_n$ is finite, then there is a finite number n of resolution steps such that the nth resolvent G_n on G is $\leftarrow a_2, \ldots, a_n$. We call the sequence $[G_1, \ldots, G_n]$ a *unit sub-refutation of* a_1 and the restriction of $\theta_1 \ldots \theta_n$ to the variables occurring in a_1 is called an *answer for* a_1.

Now if the goal G produced in the resolution step is the last goal of a unit sub-refutation of a with answer θ then the update step consists in adding θ to the list $T_S(a)$.

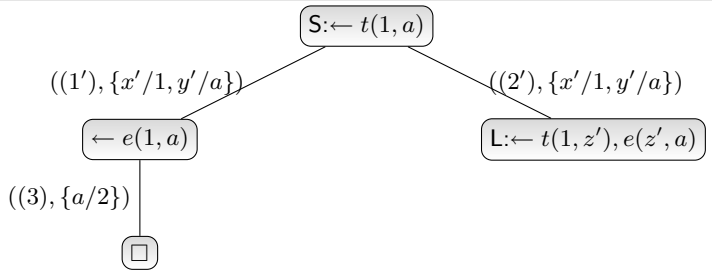
Example 215. Reconsider the program from Example 213

Listing 6_9. Transitive Closure

```
t(x,y) ← e(x,y) .
t(x,y) ← t(x,z), e(z,y) .
e(1,2) ← .
e(2,1) ← .
e(2,3) ← .
← t(1,2) .
```

After a sequence of OLDT-resolutions of solution goals or non-tabled goals the OLDT-tree in Figure 3 is constructed. To indicate which nodes are solution nodes and which are look-up nodes we prefix solution nodes with 'S:' and look-up nodes with 'L:'.

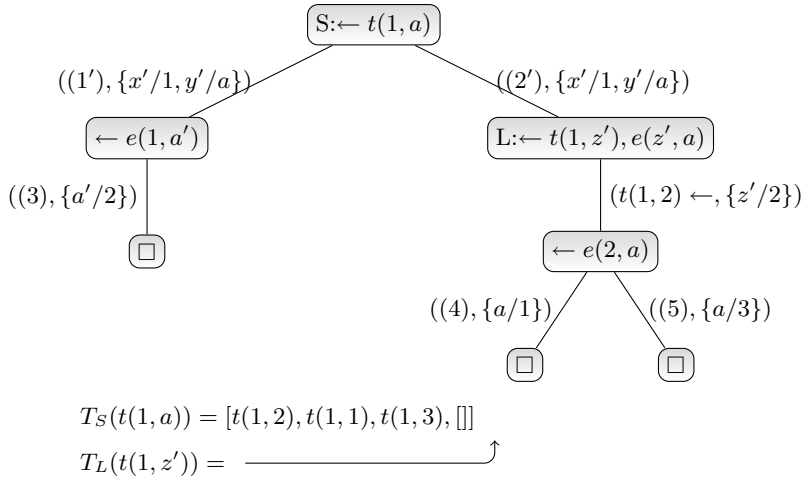
Fig. 3 An intermediary OLDT tree for program 6_9



As the left branch is a unit sub-refutation of $t(1, a)$ with solution $\{a/2\}$ the entry $t(1, 2)$ is added to the solution table. As $t(1, a)$ is more general than the leftmost atom of the goal $t(1, z'), e(z', a)$ this goal is classified as a look-up node.

Instead of using resolution to compute answers for the first atom of this goal we use the solutions stored in the solution table. The final OLDT-tree is depicted in 4:

Fig. 4 The final OLDT tree for program 6_9



Observe that the program of example 215 does not terminate with SLD-resolution while it does terminate with OLDT-resolution. The following example shows that OLDT-resolution is not complete in general.

Example 216. Consider the program 6_10 and query $q = \leftarrow$ `p(x)`

Listing 6_10. Program for which OLDT resolution is incomplete

```
p(x) ← q(x), r .
q(s(x)) ← q(x) .
q(a) ← .
r ← .
← p(x) .
```

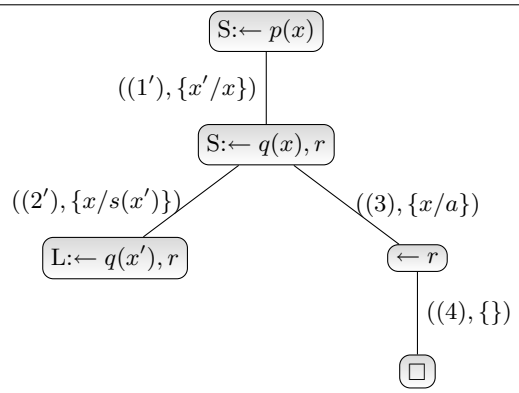
After a sequence of OLDT-resolution steps the OLDT-tree in Figure 5 is constructed

In the next step the solution $q(a)$ can be used to generate the solution $q(s(a))$ (see Figure 6).

It is easy to see that if reduction steps are only applied to the node L:← $q(x'), r$ then no solutions for $p(x)$ will be produced in finite time. Therefore OLDT is not complete in general.

This problem was addressed by the authors of OLDT. They specified a search strategy called *multistage depth-first strategy* for which they showed that OLDT

Fig. 5 An intermediary OLDT tree for program 6_10



$T_S(p(x)) = []$

$T_S(q(x)) = [q(a)]$

$T_L(q(x')) = \curvearrowright$

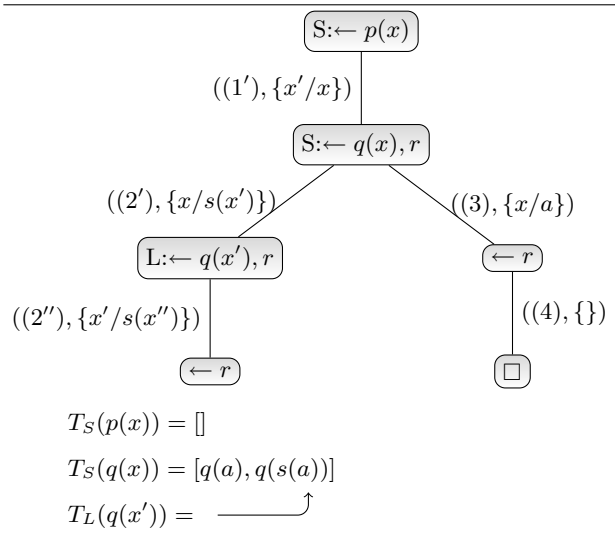
becomes complete if this search strategy is used. The idea of this search strategy is to order the nodes in the OLDT-tree and to apply OLDT-resolution-steps to the nodes in this order. If the node that is the biggest node with respect to that ordering is reduced then a stage is complete and a new stage starts where reduction is applied to the smallest node again. Therefore it is not possible to apply OLDT-steps twice in a row if there are other nodes in the tree which are resolvable.

In the above example it would therefore not be possible to repeatedly apply reductions to the node L:← $q(x'), r$ without reducing the node ← r which yields a solution for $p(x)$.

6.6 The Backward Fixpoint Procedure

The last sections have shown bottom up and top down methods for answering queries on Horn logic programs. While the naive and semi-naive bottom up methods suffer from an undirected search for answering queries, the top down methods such as SLD resolution (Section 6.4) may often not terminate although the answer can be computed in finite time by a bottom up procedure. Non-termination of top down procedures is addressed by tabling (storing the encountered sub-queries and their solutions) in OLDT resolution (Section 6.5) and other advanced top down methods such as QSQ or SLDAL-Resolution[158], the ET^* and ET_{interp}[54, 67] algorithms and the RQA/FQI[122] strategy. The problem of undirected search in forward chaining methods is solved by rewriting the rules such that special atoms representing encountered sub-goals are represented by custom-built atoms and by requiring an appropriate sub-goal to be generated before a rule of the original program is fired. Two representatives of

Fig. 6 An intermediary OLDT tree for program 6_10



this second approach are the Alexander[97] and the Magic Set methods (Section 6.2).

In [27] a sound and complete query answering method for recursive databases based on meta-interpretation called *Backward Fixpoint Procedure*, is presented, and it is shown that the Alexander and Magic Set methods can be interpreted as *specializations* of the Backward Fixpoint Procedure (BFP) and that also the efficient top down methods based on SLD resolution implement the BFP. Studying the BFP reveals the commonalities and differences between top down and bottom up processing of recursive Horn logic programs and is thus used to top of this chapter.

The backward fixpoint procedure is specified by the meta interpreter in Listing 6_11, which is intended to be evaluated by a bottom up rule engine. Facts are only generated in a bottom up evaluation of the interpreter if a query has been issued for that fact or if an appropriate sub-query has been generated by the meta-interpreter itself (Line 1). Sub-queries for rule bodies are generated if a sub-query for the corresponding rule head already exists (Line 2). Sub-queries for conjuncts are generated from sub-queries of conjunctions they appear in (Line 3 and 4). The predicate `evaluate` consults the already generated facts, and may take a single atom or a conjunction as its argument, returning true if all of the conjuncts have already been generated. It must be emphasized that using a *bottom up* rule engine for evaluating the BFP meta-interpreter for an object rule program is equivalent to evaluating this object program *top down*, thereby not generating any facts which are irrelevant for answering the query. For the correctness of the meta-interpreter approach for fixpoint computation and for an example for the evaluation of an object program with the BFP meta-interpreter see [27].

Listing 6_11. The backward fixpoint procedure meta interpreter

```
1  fact(Q) ← query_b(Q) ∧ rule(Q ← B) ∧ evaluate(B)
2  query_b(B) ← query_b(Q) ∧ rule(Q ← B)
3  query_b(Q_1) ← query_b(Q_1 ∧ Q_2)
4  query_b(Q_2) ← query_b(Q_1 ∧ Q_2) ∧ evaluate(Q_1)
```

A direct implementation of the meta-interpreter may lead to redundant computations of facts. To see this consider the application of the meta-interpreter to the object program `p ← q, r` and the query `p`. The relevant instantiated rules of the meta-interpreter contain the ground queries `evaluate(q,r)` and `evaluate(q)`, thereby accessing the fact `q` twice. Getting rid of these redundant computations is elegantly achieved by specifying a bottom up evaluation of a binary version of the predicate `evaluate` as shown in Listing 6_12. The first argument of `evaluate` contains the conjuncts which have already been proved, while the second argument contains the rest of the conjunction. Therefore a fact $\texttt{evaluate}(\emptyset, Q)$ represents a conjunction which has not yet been evaluated at all, while $\texttt{evaluate}(Q, \emptyset)$ represents a completely evaluated conjunction. With this new definition of `evaluate` the atoms $\texttt{evaluate}(B)$ and $\texttt{evaluate}(Q_1)$ in Listing 6_11 must be replaced by $\texttt{evaluate}(B, \emptyset)$ and $\texttt{evaluate}(Q_1, \emptyset)$, respectively. With this extension, the BFP meta-interpreter of Listing 6_11 becomes redundant. A non-redundant version is obtained by only considering rules (1, 5 to 11) of Listings 6_11, 6_12 and 6_13. For the proofs for the redundancy of the rules (1 to 9) on the one hand and for the equivalence of the interpreters made up of rules (1 to 4) and (1, 5 to 11) on the other hand see [27].

Listing 6_12. Implementation of the predicate `evaluate`

```
5  evaluate(∅, B) ← query_b(Q) ∧ rule(Q ← B)
6  evaluate(B_1, B_2) ← evaluate(∅, B_1 ∧ B_2) ∧ fact(B_1)
7  evaluate(B_1 ∧ B_2, B_3) ← evaluate(B_1, B_2 ∧ B_3) ∧ B_1 ≠ ∅ ∧ fact(B_2)
8  evaluate(B, ∅) ← fact(B)
9  evaluate(B_1 ∧ B_2, ∅) ← evaluate(B_1, B_2) ∧ B_1 ≠ ∅, fact(B_2)
```

Listing 6_13. Replacement rules for the rules 2 to 4 in Listing 6_11

```
10 query_b(B_2) ← evaluate(B_1, B_2) ∧ B_2 ≠ (C_1 ∧ C_2)
11 query_b(B_2) ← evaluate(B_1, B_2 ∧ B_3)
```

The implementation of the backward fixpoint procedure gives rise to two challenges exemplified by the sub-goal $\texttt{query}_b\texttt{(r(x,a))}$, which may be generated during the evaluation of a program by the BFP meta-interpreter. The sub-goal is both nested and non-ground. The main problem with non-ground terms generated by the application of the BFP meta-interpreter to an object program is that for deciding whether a newly derived fact is a logical duplicate of an already derived fact it is not sufficient to perform string matching, but full unification is needed. With *specialization*[74], a common partial evaluation technique used in logic programming, one can get rid of these problems.

Specialization is applied to the BFP meta-interpreter with respect to the rules of the object program. For each rule of the meta-interpreter that includes a premise referring to a rule of the object program, one specialized version is created for each rule of the object program. The first rule of the BFP meta-interpreter (Listing 6_11) specialized with respect to the rule `p(x)` ← `q(x)` ∧ `r(x)` results in the following partially evaluated rule:

$$\texttt{fact(p(x))} \leftarrow \texttt{query}_b\texttt{(p(x))} \wedge \texttt{evaluate(q(x)} \wedge \texttt{r(x))}$$

Similarly, the specialization of the second rule of the meta-interpreter with respect to the same object rule yields the partially evaluated rule $\texttt{query}_b\texttt{(q(x)} \wedge \texttt{r(x))} \leftarrow \texttt{query}_b\texttt{(p(x))}$.

Another specialization which can be applied to the BFP meta-interpreter is the specialization of the $\texttt{query}_b$ predicate with respect to the predicates of the object program, transforming a fact $\texttt{query}_b\texttt{(p(a))}$ into the fact $\texttt{query}_b\texttt{-p(a)}$ and eliminating some of the nested terms generated during the evaluation. With this transformation the first rule of the BFP is further simplified to:

$$\texttt{fact(p(x))} \leftarrow \texttt{query}_b\texttt{-p(x)} \wedge \texttt{evaluate(q(x)} \wedge \texttt{r(x))}$$

Getting rid of non-ground terms can also be achieved by specialization resulting in an adorned version of the program. Adornment of logic programs is described in the context of the magic set transformation in Section 6.2. [27] also discusses the *faithfulness* of representations of sub-queries as facts. Adornments of sub-queries are not completely *faithful* in the sense that the adornment of the distinct queries `p(X,Y)` and `p(X,X)` results in the same adorned predicate $\texttt{p}^{ff}$. As described in Section 6.2, multiple adorned versions for one rule of the original program may be generated.

In [27] it is shown that the above specializations of the meta-interpreter made up of the rules (1, 5 to 11) with respect to an object program P and the omittance of the meta-predicates `evaluate` and `fact` yields exactly the supplementary magic set transformation and the Alexander method applied to P. Therefore both of these methods implement the BFP.

Not only can bottom up processing methods be specialized from the BFP, but it can also be used to specify top down procedures such as ET^*, ET_{interp}, QSQ, etc. The difference between SLD resolution and the BFP is explained in [27] in terms of the employed data structures. SLD resolution uses a hierarchical data structure which relates sub-queries and proved facts to the queries they belong to. On the other hand, the BFP employs relations – i.e. a flat data structure – for memorizing answers to queries, and therefore allows to share computed answers between queries that are logically equivalent. SLD resolution and the hierarchical resolution tree can be simulated by the BFP by introducing identifiers for queries, thus allowing to relate facts and sub-queries with queries to the answers of which they contribute. See [27] for details. This simulation prevents sharing of answers between queries and thus makes the evaluation less efficient. One conclusion that can be drawn from the BFP is that it does not make sense to hierarchically structure queries according to their generation. In contrast it makes sense to rely on a static rewriting such as the Alexander or

Magic Set rewriting and process the resulting rules with a semi-naive bottom-up rule engine.

7 Operational Semantics: Rule Sets with Non-monotonic Negation

In the previous chapter evaluation methods for logic programs without negation are examined. This chapter considers a more general form of logic programs, namely ones that use negation. The rules considered in this chapter are all of the form

$$A \leftarrow L_1, \ldots, L_n$$

where the $L_i, 1 \leq, i \leq n$ are literals, and A is an atom. Thus negative literals are allowed in the bodies of rules, but not in their heads. It is important to note that augmenting logic programming with negation increases expressivity and is necessary for deriving certain information from a database.

In Section 5 two important semantics for logic programming with negation have been described: The *stable model semantics* (See Section 5.3.2) and the *well-founded model semantics* (Section 5.3.3). However, this declarative semantics does not provide an easy to implement algorithm neither for computing the entailments of a logic program with negation nor for answering queries with respect to such programs. In fact the greatest unfounded sets in the definition of the well-founded semantics and the stable models in the stable models theory must be guessed.

In this section constructive algorithms for computing the so-called *iterative fixpoint semantics*, the stable model semantics and the well-founded model semantics are described and applied to example programs to better illustrate their functioning.

The kind of negation which is generally used in logic programming is called negation as failure and can be described as follows: A negated literal $\neg A$ is true, if its positive counterpart A cannot be derived from the program. A possible application of negation as failure is given in Listing 7_14. Since `male(eduard)` is given and `married(eduard)` cannot be derived, `bachelor(eduard)` is logical consequence of the program, while `bachelor(john)` is not. Negation as failure is also known under the term *non-monotonic* negation, because the addition of new facts to a program may cause some of its entailments to be no longer true: The addition of `married(eduard)` to the program in Listing 7_14 invalidates the conclusion `bachelor(eduard)`.

Listing 7_14. An Illustration for the Use of Negation as Failure

```
married(john).
male(john).
male(eduard).
bachelor(X) ← male(X), not married(X)
```

The adoption of negation as failure brings about several interesting, not to say intricate questions. Consider the program in Listing 7_15. If the literal `q` is assumed to be false, then the literal `p` must be true according to the second rule. This, however, causes the literal `q` to be true. On the other hand there is no possibility of deriving the literal `q`. Hence the semantics of Program 7_15 is not clear. Therefore syntactical restrictions on logic programs have been proposed, to ensure that all programs satisfying these restrictions have an intuitive declarative semantics.

Listing 7_15. A program with Recursion through negation

```
q ← p.
p ← not q.
```

7.1 Computation of the Iterative Fixpoint Semantics for Stratified Logic Programs with Negation as Failure

One possibility of limiting the use of negation is by *stratification* (see Definition 164 in Subsection 5.3.1).

It is easy to see that for some programs no stratification can be found. Since in Program 7_15 `q` depends on `p` by the first rule and `p` depends on `q` by the second rule, they would have to be defined in the same stratum. Since `q` depends *negatively* on `p`, this case is precluded by the third premise above. A program for which no stratification can be found is called *non-stratifiable*, programs for which a stratification exists are called *stratifiable*. The iterative fixpoint semantics does not provide a semantics for non-stratifiable programs.

[9] defines the semantics for stratified logic programs as an iterated fixpoint semantics based on the immediate consequence operator $\mathbf{T}_P$ (Definition 153) as follows. Let $S_1, \ldots, S_n$ be a stratification for the program P. Recall that $\mathbf{T}_P^i(I)$ denotes the i-fold application of the immediate consequence operator to the database instance I. Then the semantics of the program P is the set M_n where M_0 is defined as the initial instance over the extensional predicate symbols of P, and the M_i are defined as follows:

$$M_1 := \mathbf{T}_{S_1}^{\omega}(M_0), \quad M_2 := \mathbf{T}_{S_2}^{\omega}(M_1), \quad \ldots, \quad M_n := \mathbf{T}_{S_n}^{\omega}(M_{n-1})$$

This procedure shall be illustrated at the example program in Listing 7_16. The intensional predicate symbols `has_hobbies`, `has_child`, `married`, and `bachelor` can be separated into the strata $S_1 := \{\texttt{has\_hobbies}\}$, $S_2 := \{\texttt{has\_child}, \texttt{married}\}$, $S_3 := \{\texttt{bachelor}\}$. The initial instance $M_0 = \{\{john\}_{human}, \{john\}_{plays\_the\_piano}, \{john\}_{male}\}$ is directly derived from the facts of the program. Since `has_hobbies` is the only element of S_1, only the first rule is relevant for the computation of $M_1 := M_0 \cup \{\{john\}_{has\_hobbies}\}$. The second and the third rule are relevant for the computation of M_2, which is the same instance as M_1. Finally, the fourth rule allows to add the fact `bachelor(john)` yielding the final instance $M_3 := M_2 \cup \{\{john\}_{bachelor}\}$.

Listing 7_16. A stratifiable program with negation as failure

```
human(john).
male(john).
plays_the_piano(john).

has_hobbies(X) ← plays_the_piano(X).
has_child(X) ← human(X), not has_hobbies(X).
married(X) ← human(X), has_child(X).
bachelor(X) ← male(X), not married(X).
```

7.2 Magic Set Transformation for Stratified Logic Programs

While the computation of the iterative fixpoint of a stratified logic program allows to answer an arbitrary query on the program, it is inefficient in the sense that no goal-directed search is performed. One method for introducing goal-directedness into logical query answering, that is also relatively easy to implement, is the magic set rewriting as introduced in Section 6.2. The task of transferring the magic set approach to logic programs with negation has therefore received considerable attention [41], [98], [16], [139], [15].

The main problem emerging when applying the magic set method to programs with negative literals in rule bodies is that the resulting program may not be stratified. There are two approaches to dealing with this situation. The first one is to use a preprocessing stage for the original program in order to obtain a stratified program under the magic set transformation and is pursued by [41]. The second one is to accept the unstratified outcome of the magic set transformation and to compute some other semantics which deals with unstratified programs. This second approach is employed by [19], which proposes to compute the well-founded model by Kerisit's *weak consequence operator* at the aid of a new concept called *soft stratification*.

Because of its simplicity and straightforwardness, only the first approach is presented in this article.

In [41] three causes for the unstratification of a magic-set transformed program are identified:

- both an atom `a` and its negation `not a` occur within the body of the same rule of the program to be transformed.
- a negative literal occurs multiple times within the body of a rule of the original program
- a negative literal occurs within a recursive rule

With `b` occurring both positively and negatively in the body of the first rule, Listing 7_17 is an example for the first cause of unstratification. The predicate symbol `c` is an extensional one, and therefore no magic rules are created for it, `a` and `b` are intensional ones. When the magic set transformation from the last chapter is naively applied to the program and the query `a(1)`, which means that negated literals are transformed in the very same way as positive ones, the magic

set transformed program in Listing 7_18 is not stratified, because it contains a negative dependency cycle among its predicates: `magic_b`^{b} negatively depends on `b`^{b}, which again depends on `magic_b`^{b}.

Listing 7_17. A program leading to unstratification under the naive magic set transformation

```
a(x) ← not b(x), c(x,y), b(y).
b(x) ← c(x,y), b(y).
```

Listing 7_18. The unstratified outcome of the magic set transformation applied to Program 7_17

```
magic_a^b(1).
magic_b^b(x) ← magic_a^b(x).
magic_b^b(y) ← magic_a^b(x), not b^b(x), c(x,y).
a(x) ← magic_a^b(x), not b^b(x), c(x,y), b^b(y).
magic_b^b(y) ← magic_b^b(x), c(x,y).
b(x) ← magic_b^b(x), c(x,y), b(y).
```

[41] proposes to differentiate the contexts in which a negative literal is evaluated by numbering the occurrences of the literal in the rule body before the magic set transformation is applied. The numbered version of Listing 7_17 is displayed in Listing 7_19 where the two occurrences of `b` have been numbered. Additionally, for each newly introduced numbered predicate symbol p_i its defining rules are copies from the definition of the unnumbered symbol p, with all occurrences of p replaced by p_i. In this way, the semantics of the program remains unchanged, but the program becomes stratified under the magic set transformation, as can be verified in Listing 7_20.

Listing 7_19. Program 7_17 with differentiated contexts for the literal `b`

```
a(x) ← not b_1(x), c(x,y), b_2(y).
b_1(x) ← c(x,y), b_1(y).
b_2(x) ← c(x,y), b_2(y).
```

Listing 7_20. The stratified outcome of the magic set transformation applied to program 7_19

```
magic_a^b(1).
magic_b_1^b(x) ← magic_a^b(x).
magic_b_2^b(y) ← magic_a^b(x), not b_1^b(x), c(x,y).
a^b(x) ← magic_a^b(x), not b_1^b(x), c(x,y), b_2^b(y).
magic_b_1^b(y) ← magic_b_1^b(x), c(x,y).
b_1(x) ← magic_b_1^b(x), c(x,y), b_1(y).
magic_b_2^b(y) ← magic_b_2^b(x), c(x,y).
b_2(x) ← magic_b_2^b(x), c(x,y), b_2(y).
```

Also for the second and third source of unstratification, elimination procedures can be specified that operate on the adorned rule set, but are carried out

prior to the magic set transformation. For more details and a proof, that the resulting programs are indeed stratified see [41].

7.3 Computation of the Stable Model Semantics

While the iterative fixpoint semantics provides an intuitive and canonical semantics for a subset of logic programs with negation as failure, several attempts have been made to assign a semantics to programs which are not stratifiable. One of these attempts is the *stable model semantics* (see Section 5.3.2).

An example for a program which is not stratifiable but is valid under the stable model semantics is given in Listing 7_21.

Listing 7_21. A non-stratifiable program with a stable model semantics

```
married(john, mary).
male(X) ← married(X,Y), not male(Y).
```

The stable model semantics for a program P is computed as follows: In a first step all rules containing variables are replaced by their ground instances. In a second step the program is transformed with respect to a given model M into a program $GL_M(P)$ by deleting all those rules which contain a negative literal $not(L)$ in their body where L is contained in M, and by deleting all those negative literals $not(L)$ from the rules for which L is not contained in M. Clearly, the semantics of the program P remains unchanged by both of these transformations with respect to the particular model M. Since this transformation has first been proposed by Gelfond and Lifschitz in [77], it is also known under the name *Gelfond-Lifschitz-Transformation* (See also Definition 165).

An Herbrand interpretation M is a *stable set* of P if and only if it is the unique minimal Herbrand model of the resulting negation-free program $GL_M(P)$. See Definition 166 and Lemma 167. The stable model semantics for a program P (written $S_\Pi(P)$) is defined as the stable set of P, and remains undefined if there is none or more than one of them.

The Gelfond-Lifschitz-Transformation of Listing 7_21 with respect to the set $M := \{$`married(john, mary)`, `male(John)`$\}$ results in the Program in Listing 7_22. Rule instances and negative literals of rule instances have been crossed out according to the rules mentioned above. Since the unique minimal Herbrand model of the resulting Program is also $M = \{$`married(john, mary)`, `male(John)`$\}$, M is a stable set of the original program in 7_21. Since there are no other stable sets of the program, M is its stable model.

Listing 7_22. Listing 7_21 transformed with respect to the set {`married(john, mary)`, `male(John)`}

```
married(john, mary).
male(john) ← married(john, mary), not male(mary).
male(mary) ← married(mary, john), not male(john).
male(mary) ← married(mary, mary), not male(mary).
male(john) ← married(john, john), not male(john).
```

An efficient implementation of the stable model semantics (and also the well-founded model semantics) for range-restricted and function-free normal logic programs is investigated in [123] and its performance is compared to that of SLG resolution in [40].

7.4 A More Efficient Implementation of the Stable Model Semantics

The approach of [123] recursively constructs all possible stable models by adding one after another positive and negative literals from the set of *negative antecedents* (Definition 217) to an intermediate candidate full set B (see Definition 219). The algorithm (see Listing 7_23) makes use of backtracking to retract elements from B and to find all stable models of a program. In addition to the program P itself and the intermediate candidate set B, the algorithm takes a third argument which is a formula ϕ that is tested for validity whenever a stable model is found. The algorithm returns true if there is a stable model of ϕ and false otherwise. Moreover it can be adapted to find all stable models of a program or to determine whether a given formula is satisfied in all stable models.

Definition 217 (Negative Antecedents). *Given a logic program P, the set of its negative antecedents $NAnt(P)$ is defined as all those atoms a such that $not(a)$ appears within the body of a rule of P.*

Definition 218 (Deductive closure). *The deductive closure $Dcl(P, L)$ of a program P with respect to a set of literals L is the smallest set of atoms containing the negative literals L^- of L, which is closed under the following set of rules:*

$$R(P,L) := \{\, c \leftarrow a_1, \ldots, a_n \mid c \leftarrow a_1, \ldots, a_n, not(b_1), \ldots, not(b_m) \in P, \quad \{not(b_1), \ldots, not(b_m) \subseteq L^-\}\} \tag{1}$$

Definition 219 (Full Sets (from [123])). *A set Λ of not-atoms (negated atoms) is called P-*full *iff for all $\phi \in NAnt(P)$, $not(\phi) \in \Lambda$ iff $\phi \notin Dcl(P, \Lambda)$.*

Example 220. Consider the logic program $P := \{q \leftarrow not(r).q \leftarrow not(p).r \leftarrow q.\}$ The negative antecedents of the program are $NAnt(P) = \{r, p\}$. $\Lambda_1 := \{not(r), not(p)\}$ is not a full set with respect to P because $not(r)$ is in Λ_1, but r is in the deductive closure $Dcl(P, \Lambda_1)$. The only full set with respect to P is $\Lambda_2 := \{not(p)\}$: $not(p) \in \Lambda_2$ and $p \notin Dcl(P, \Lambda_2)$ holds for p and $not(r) \notin \Lambda_2$ and $r \in Dcl(P, \Lambda_2)$ holds for the other element r of $NAnt(P)$.

Theorem 221 (Relationship between full sets and stable models ([123])). *Let P be a ground program and Λ a set of not-atoms (negated atoms).*
(i) If Λ is a full set with respect to P then $Dcl(P, \Lambda)$ is a stable model of P.
(ii) If Δ is a stable model of P then $\Lambda = not(NAnt(P) - \Delta)$ is a full set with respect to P such that $Dcl(P, \Lambda) = \Delta$.

According to theorem 221 it is sufficient to search for full sets instead of for stable models, because stable models can be constructed from these full sets. This approach is pursued by the algorithm in Listing 7_23.

Definition 222 (L covers A). *Given a set of ground literals L and a set of ground atoms A, L is said to* cover *A, iff for every atom a in A either $a \in L$ or $not(a) \in L$ holds.*

Listing 7_23. Efficient computation of the stable model semantics for a logic program

```
function stable_model(P,B,φ)
  let B' = expand(P,B) in
    if conflict(P,B') then false
    else
      if (B' covers NAnt(P)) then test(Dcl(P,B'),φ)
      else
        take some χ ∈ NAnt(P) not covered by B'
        if stable_model(P, B' ∪ {not(χ)},φ) then true
           else stable_model(P, B' ∪ {χ},φ)
```

The algorithm is not fully specified, but relies on the two functions `expand` and `conflict`, which undergo further optimization. The idea behind the function expand is to derive as much further information as possible about the common part B of the stable models that are to be constructed without losing any model. One could also employ the identity function as an implementation for `expand`, but by burdening the entire construction of the full sets on the backtracking search of the function `stable_model`, this approach would be rather inefficient. A good choice for the `expand` function is to return the least fixpoint of the *Fitting operator* $F_P(B)$:

Definition 223 (Fitting operator F_P). *Let B be a set of ground literals, and P a ground logic program. The set $F_P(B)$ is defined as the smallest set including B that fulfills the following two conditions:*
(i) for a rule $h \leftarrow a_1, \ldots, a_n, not(b_1), \ldots, not(b_m)$ with $a_i \in B, 1 \leq i \leq n$ and $not(b_j), \in B, 1 \leq j \leq m$, h is in $F_P(B)$.
(ii) if for an atom a such that for all of its defining rules, a positive premise p is in its body and $not(p)$ is in B, or a negative literal $not(p)$ is in its body with p in B, then a is in $F_P(B)$.

The function `conflict` returns true whenever (i) B covers $NAnt(P)$, and (ii) if there is an atom a in the set B such that $a \notin Dcl(P, B)$ or a literal $not(a) \in B$ such that $a \in Dcl(P, B)$. In this way, `conflict` prunes the search for full sets (and therefore the search for stable models) by detecting states in which no stable model can be constructed as early as possible. For further optimizations regarding the computation of the stable model semantics with the function `stable_model` the reader is referred to [123].

7.5 Computation of the Well-Founded Model Semantics

Another approach to defining a semantics for logic programs that are neither stratifiable nor locally stratifiable is the *well-founded model approach* [152] (see Section 5.3.3).

Recall that in this context the term *interpretation* refers to a set of positive or negative literals $\{p_1, \ldots, p_k, not(n_1), \ldots, not(n_i)\}$ and that the notation $\overline{S}$, with $S = \{s_1, \ldots, s_n\}$ being a set of atoms, refers to the set $\{\neg s_1, \ldots, \neg s_n\}$ in which each of the atoms is negated.

An *unfounded set* (see Section 5.3.3) of a logic program P with respect to an interpretation I is a set of (positive) atoms U, such that for each instantiated rule in P which has head $h \in U$, at least one of the following two conditions applies.

- the body of the rule is not fulfilled, because it contains either a negative literal `not(a)` with `a`$\in I$ or a positive literal `a` with `not(a)`$\in I$.
- the body of the rule contains another (positive) atom $a \in U$ of the unfounded set.

The greatest unfounded set turns out to be the union of all unfounded sets of a program. Note that the definition above does not immediately provide an algorithm for finding the greatest unfounded set. In this subsection, however, a straight-forward algorithm is derived from the definition and in the following subsection, a more involved algorithm for computing the well-founded semantics is introduced.

The computation of the well founded semantics is an iterative process mapping interpretations to interpretations and involving the computation of immediate consequences and greatest unfounded sets. The initial interpretation is the empty set, which reflects the intuition that at the beginning of the program examination, nothing is known about the entailments of the program. The iteration uses the following three kinds of mappings:

- the immediate consequence mapping $\mathbf{T}_P(I)$ of the program with respect to an interpretation I
- the greatest unfounded set mapping $\mathbf{U}_P(I)$, which finds the greatest unfounded set of a program with respect to an interpretation I
- $\mathbf{W}_P(I) := \mathbf{T}_P(I) \cup \overline{\mathbf{U}_P(I)}$ which maps an interpretation to the union of all of its immediate consequences and the set of negated atoms of the greatest unfounded set.

The well-founded semantics (see Section 5.3.3) of a logic program is then defined as the least fixpoint of the operator $\mathbf{W}_P(I)$. The computation of well founded sets and of the well founded semantics is best illustrated by an example (see Listing 7_24). The set of immediate consequences $\mathbf{T}_P(\emptyset)$ of the empty interpretation of the program 7_24 is obviously the set $\{c(2)\}$. The greatest unfounded set $\mathbf{U}_P(\emptyset)$ of the program with respect to the empty interpretation is the set $\{d(1), f(2), e(2), f(1)\}$. $f(1)$ is in $\mathbf{U}_P(\emptyset)$, because there are no rules with head $f(1)$, and therefore the conditions above are trivially fulfilled.

Note that the fact `a(1)` is not an unfounded fact with respect to the interpretation $\emptyset$, although one is tempted to think so when reading the program as a logic program with negation as failure semantics.

The three atoms $\{d(1), f(2), e(2)\}$ form an unfounded set, because the derivation of any of them would require one of the others to be already derived. There is no possibility to derive any of them first. Hence, according to the well-founded semantics, they are considered false, leading to $I_1 := \mathbf{W}_P(\emptyset) = \mathbf{T}_P(\emptyset) \cup \overline{\mathbf{U}_P(\emptyset)} = \{c(2)\} \cup \overline{\{d(1), f(2), e(2), f(1)\}} = \{c(2), \neg d(1), \neg f(2), \neg e(2), \neg f(1)\}$.

In the second iteration `a(1)` is an immediate consequence of the program, but still neither one of the atoms `a(2)` and `b(2)` can be added to the interpretation (also their negated literals cannot be added). After this second iteration the fixpoint $\{c(2), \neg d(1), \neg f(2), \neg e(2), \neg f(1), a(1)\}$ is reached without having assigned a truth value to the atoms `a(2)` and `b(2)`.

Listing 7_24. Example program for the well-founded semantics

```
b(2) ← ¬ a(2).
a(2) ← ¬ b(2).

d(1) ← f(2), ¬ f(1).
e(2) ← d(1).
f(2) ← e(2).

a(1) ← c(2), ¬ d(1).
c(2).
```

The computation of this partial well-founded model involves guessing the unfounded sets of a program P. If P is finite, all subsets of the atoms occurring in P can be tried as candidates for the unfounded sets. In practice those atoms that have already been shown to be true or false do not need to be reconsidered, and due to the fact that the union of two unfounded sets is an unfounded set itself, the greatest unfounded set can be computed in a bottom up manner, which decreases the average case complexity of the problem. Still, in the worst case $O(2^a)$ sets have to be tried for each application of the operator $\mathbf{U}_P$, with a being the number of atoms in the program. In [154] a deterministic algorithm for computing the well-founded semantics of a program is described.

7.6 Computing the Well-Founded Semantics by the Alternating Fixpoint Procedure

The central idea of the alternating fixpoint procedure[154] is to iteratively build up a set of negative conclusions $\tilde{A}$ of a logic program, from which the positive conclusions can be derived at the end of the process in a straightforward way. Each iteration is a two-phase process transforming an underestimate of the negative conclusions $\tilde{I}$ into a temporary overestimate $\tilde{\mathbf{S}}_P(\tilde{I})$ and back to an underestimate $\mathbf{A}_P(\tilde{I}) := \tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\tilde{I}))$. Once this two-phase process does not yield further negative conclusions, a fixpoint is reached. The set of negative conclusions of the program is then defined as the least fixpoint $\tilde{A} := \mathbf{A}_P^{\omega}(\emptyset)$ of the monotonic transformation $\mathbf{A}_P$.

In each of the two phases of each iteration the fixpoint $S_P(\tilde{I}) := \mathbf{T}_{P'}^{\omega}(\emptyset)$ of an adapted version of the immediate consequence operator corresponding to the

ground instantiation of the program P_H plus the set $\tilde{I}$ of facts that are already known to be false, is computed.

In the first phase the complement $\tilde{\mathbf{S}}_P(\tilde{I}) := \overline{(H - \mathbf{S}_P(\tilde{I}))}$ of this set of derivable facts constitutes an overestimate of the set of negative derivable facts, and in the second phase the complement $\tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\tilde{I}))$ is an underestimate.

Let's now turn to the adapted immediate consequence operator. As in the previous sections, the algorithm does not operate on the program P itself, but on its Herbrand instantiation P_H. Based on the Herbrand instantiation P_H and a set of negative literals $\tilde{I}$ a derived program $P' := P_H \cup \tilde{I}$ and a slightly altered version of the immediate consequence operator $\mathbf{T}_{P'}$ of P' are defined.

A fact f is in the set $\mathbf{T}_{P'}(I)$ of immediate consequences of the program P' if all literals l_i in the body of a rule with head f are fulfilled. The difference to the previous definition of the immediate consequence operator is that the bodies of the rules are not required to be positive formulas, but may contain negative literals as well. A negative literal in the rule body is only fulfilled, if it is explicitly contained in the program P' (stemming from the set $\tilde{I}$ which is one component of P'). A positive literal in the body of P' is fulfilled if it is in the interpretation I.

For the proof of the equivalence of the partial models computed by the well-founded model semantics and the alternating fixpoint algorithm the reader is referred to [154].

The computation of the well-founded semantics of the program in Listing 7_24 with the alternating fixpoint procedure is achieved without guessing well-founded sets by the following steps.

- $\mathbf{S}_P(\emptyset) = \{c(2)\}$. The fact $a(1)$ cannot be derived because negation is not treated as negation as failure, but only if the negated literal is in $\tilde{I}$. Similarly, neither one of the facts $b(2)$ and $a(2)$ are in $\mathbf{S}_P(\emptyset)$.
- $\tilde{\mathbf{S}}_P(\emptyset) = \overline{(H - \mathbf{S}_P(\emptyset))} = \overline{\{a(2), b(2), f(2), f(1), d(1), e(2), f(2), c(2), a(1)\}}$-$\overline{\{c(2)\}} = \{\neg a(2), \neg b(2), \neg f(2), \neg f(1), \neg d(1), \neg e(2), \neg f(2), \neg a(1)\}$ is the first overestimate of the derivable negative facts.
- $\mathbf{S}_P(\tilde{\mathbf{S}}_P(\emptyset)) = \{c(2), a(1), b(2), a(2)\}$ and thus $\mathbf{A}_P(\emptyset) = \tilde{\mathbf{S}}_P(\tilde{\mathbf{S}}_P(\emptyset)) = \{\neg f(2), \neg f(1), \neg d(1), \neg e(2)\}$ is the second underestimate of the derivable negative literals (the first one was the emptyset).
- $\tilde{\mathbf{S}}_P(\tilde{A}_P(\emptyset)) = \{\neg a(2), \neg b(2), \neg f(2), \neg f(1), \neg d(1), \neg e(2)\}$
- $\mathbf{A}_P(\mathbf{A}_P(\emptyset)) = \{\neg f(2), \neg f(1), \neg d(1), \neg e(2)\} = \mathbf{A}_P(\emptyset)$ means that the fixpoint has been reached and $\tilde{A} = \mathbf{A}_P(\emptyset)$ is the set of of negative literals derivable from the program.
- The well founded partial model of the program is given by $\tilde{A} \cup \mathbf{S}_P(\tilde{A}) = \{\neg f(2), \neg f(1), \neg d(1), \neg e(2), c(2), a(1)\}$, which is the same result as in the previous section.

For finite Herbrand universes the partial well-founded model is computable in $O(h)$ with h being the size of the Herbrand Universe [154].

7.7 Other Methods for Query Answering for Logic Programs with Negation

While this chapter gives a first idea on methods for answering queries on stratified and general logic programs with negation, many approaches have not been mentioned. The previous sections have shown different ways of implementing the stable model semantics and the well-founded model semantics for general logic programs mainly in a forward chaining manner (except for the magic set transformation, which is a method of introducing goal-directed search into forward chaining).

The best known backward chaining method for evaluating logic programs with negation is an extension of SLD resolution with negation as failure, and is called SLDNF [33][10][145][58]. SLDNF is sound with respect to the completion semantics [42] of a logic program and complete for Horn logic programs [93].

Przymusinski introduced SLS resolution[135] as a backward chaining operational semantics for general logic programs under the perfect model semantics [132]. SLS resolution was extended by Ross to global SLS-resolution[140], which is a procedural implementation of the well-founded model semantics.

8 Complexity and Expressive Power of Logic Programming Formalisms

8.1 Complexity Classes and Reductions

In this section we recall what is meant by the complexity of logic programming. Moreover, we provide definitions of the standard complexity classes encountered in this survey and provide other related definitions. For a detailed exposition of the complexity notions, the reader is referred to e.g., [94, 126].

8.1.1 Decision Problems. In this section, we only deal with *decision problems*, i.e., problems where the answer is "yes" or "no". Formally, a decision problem is a language L over some alphabet Σ. An *instance* of such a decision problem is given as a word $x \in \Sigma^*$. The question to be answered is whether $x \in L$ holds. Accordingly, the answer is "yes" or "no", respectively. The resources (i.e., either time or space) required in the worst case to find the correct answer for any instance x of a problem L is referred to as the *complexity* of the problem L.

8.1.2 Complexity of Logic Programming. There are three main kinds of decision problems (and, thus, three main kinds of complexity) connected to plain datalog and its various extensions [156]:

- The ***data complexity*** is the complexity of the following decision problem: Let P be some *fixed* datalog program.

  Instance. An input database D_{in} and a ground atom A.
  Question. Does $D_{in} \cup P \models A$ hold?

- The ***program complexity*** (also called *expression complexity*) is the complexity of the following decision problem: Let D_{in} be some *fixed* input database.

  Instance. A datalog program P and a ground atom A.
  Question. Does $D_{in} \cup P \models A$ hold?

- The ***combined complexity*** is the complexity of the following decision problem:

  Instance. A datalog program P, an input database D_{in}, and a ground atom A.
  Question. Does $D_{in} \cup P \models A$ hold?

Note that for all versions of datalog considered in this paper, the combined complexity is equivalent to the program complexity with respect to polynomial-time reductions. This is due to the fact that with respect to the derivation of ground atoms, each pair $\langle D_{in}, P\rangle$ can be easily reduced to the pair $\langle D_\emptyset, P^*\rangle$, where $D_\emptyset$ is the empty database instance associated with a universe of two constants c_1 and c_2, and P^* is obtained from $P \cup D_{in}$ by straightforward encoding of the universe $U_{D_{in}}$ using n-tuples over $\{c_1, c_2\}$, where $n = \lceil |U_{D_{in}}| \rceil$. For this reason, we mostly disregard the combined complexity in the material concerning datalog.

As for logic programming in general, a generalization of the combined complexity may be regarded as the main complexity measure. Below, when we speak about the ***complexity*** of a fragment of logic programming, we mean the complexity of the following decision problem:

Instance. A datalog program P and a ground atom A.
Question. Does $P \models A$ hold?

8.1.3 Complexity Classes. Normally, the time complexity of a problem is expressed in terms of the steps needed by a Turing machine which decides this problem. Likewise, the space complexity corresponds to the number of cells visited by a Turing machine. However, the complexity classes we are interested in here can be defined by any "reasonable" machine model, e.g. random access machines, which are more closely related to real-world computers.

We shall encounter the following complexity classes in this survey.

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXPTIME} \subseteq \mathsf{NEXPTIME}$$

These are the classes of problems which can be solved in logarithmic space (L), non-deterministic logarithmic space (NL), polynomial time (P), non-deterministic polynomial time (NP), polynomial space (PSPACE), exponential time (EXPTIME), and non-deterministic exponential time (NEXPTIME).

Any complexity class $\mathcal{C}$ has its *complementary class* denoted by co-$\mathcal{C}$, which is defined as follows. For every language $L \subseteq \Sigma^*$, let $\overline{L}$ denote its *complement*, i.e. the set $\Sigma^* \setminus L$. Then co-$\mathcal{C}$ is $\{\overline{L} \mid L \in \mathcal{C}\}$.

Another interesting kind of complexity classes are the classes of the polynomial hierarchy. Formally, they are defined in terms of oracle Turing machines. Intuitively, an oracle is a subroutine for solving some sub-problem where we do not count the cost of the computation. Let $\mathcal{C}$ be a set of languages. For a language L, we say that $L \in \mathsf{P}^{\mathcal{C}}$ (or $L \in \mathsf{NP}^{\mathcal{C}}$) if and only if there is some language $A \in \mathcal{C}$ such that L can be decided in polynomial-time (resp. in non-deterministic polynomial-time) by an algorithm using an oracle for A. The *polynomial hierarchy* consists of classes Δ_i^p, Σ_i^p, and Π_i^p defined as follows:

$$\begin{aligned} \Delta_0^p = \Sigma_0^p = \Pi_0^p &= \mathsf{P} \\ \Delta_{i+1}^p &= \mathsf{P}^{\Sigma_i^p} \\ \Sigma_{i+1}^p &= \mathsf{NP}^{\Sigma_i^p} \\ \Pi_{i+1}^p &= \mathsf{co}\text{-}\Sigma_{i+1}^p \end{aligned}$$

for all $i \geq 0$. The class PH is defined as

$$\mathsf{PH} = \bigcup_{i \geq 0} \Sigma_i^p$$

.

8.1.4 Reductions. Let L_1 and L_2 be decision problems (i.e., languages over some alphabet Σ). Moreover, let $R : \Sigma^* \to \Sigma^*$ be a function which can be computed in logarithmic space and which has the following property: for every $x \in \Sigma^*$, $x \in L_1$ if and only if $R(x) \in L_2$. Then R is called a *logarithmic-space reduction* from L_1 to L_2 and we say that L_1 is *reducible* to L_2.

Let $\mathcal{C}$ be a set of languages. A language L is called $\mathcal{C}$*-hard* if any language L' in $\mathcal{C}$ is reducible to L. If L is $\mathcal{C}$-hard and $L \in \mathcal{C}$ then L is called *complete for* $\mathcal{C}$ or simply $\mathcal{C}$*-complete*.

Besides the above notion of a reduction, complexity theory also considers other kinds of reductions, like polynomial-time reductions or Turing reductions (which are more liberal kinds of reductions). In this paper, unless otherwise stated, a reduction means a logarithmic-space reduction. However, we note that in several cases, results that we shall review have been stated for polynomial-time reductions, but the proofs establish that they hold under logarithmic-space reductions as well.

8.1.5 Turing Machines. As was already mentioned above, the complexity classes considered here are usually defined in terms of Turing machines. On the other hand, as soon as one has several complete problems for some complexity class $\mathcal{C}$, further $\mathcal{C}$-hardness results are usually obtained by reducing one of the already known $\mathcal{C}$-hard problems to the new problem under investigation. In other words, Turing machines are no longer needed explicitly. However, in the context of logic programming, a great portion of the hardness results recalled below have very intuitive proofs "from first principles" (i.e., via reductions from the computations of Turing machines rather than via reductions from other problems).

We therefore briefly recall the definition of deterministic and non-deterministic Turing machines.

A *deterministic Turing machine (DTM)* is defined as a quadruple (S, Σ, δ, s_0) with the following meaning: S is a finite set of *states*, Σ is a finite alphabet of *symbols*, δ is a *transition function*, and $s_0 \in S$ is the *initial state*. The alphabet Σ contains a special symbol $\sqcup$ called the *blank*. The transition function δ is a map

$$\delta: \quad S \times \Sigma \quad \rightarrow \quad (S \cup \{\texttt{yes}, \texttt{no}\}) \times \Sigma \times \{\texttt{-1, 0, +1}\},$$

where `yes`, and `no` denote two additional states not occurring in S, and `-1, 0, +1` denote *motion directions*. It is assumed here, without loss of generality, that the machine is well-behaved and never moves off the tape, i.e., $d \neq \texttt{-1}$ whenever the cursor is on the leftmost cell; this can be easily ensured by proper design of δ (or by a special symbol which marks the left end of the tape).

Let T be a DTM (Σ, S, δ, s_0). The tape of T is divided into *cells* containing symbols of Σ. There is a *cursor* that may move along the tape. At the start, T is in the initial state s_0, and the cursor points to the leftmost cell of the tape. An *input string* I is written on the tape as follows: the first $|I|$ cells $c_0, \ldots, c_{|I|-1}$ of the tape, where $|I|$ denotes the length of I, contains the symbols of I, and all other cells contain $\sqcup$.

The machine takes successive *steps* of computation according to δ. Namely, assume that T is in a state $s \in S$ and the cursor points to the symbol $\sigma \in \Sigma$ on the tape. Let

$$\delta(s, \sigma) = (s', \sigma', d).$$

Then T changes its current state to s', overwrites σ' on σ, and moves the cursor according to d. Namely, if $d = \texttt{-1}$ or $d = \texttt{+1}$, then the cursor moves to the previous cell or the next one, respectively; if $d = \texttt{0}$, then the cursor remains in the same position.

When any of the states `yes` or `no` is reached, T halts. We say that T *accepts* the input I if T halts in `yes`. Similarly, we say that T *rejects* the input in the case of halting in `no`.

A *non-deterministic Turing machine (NDTM)* is defined as a quadruple (S, Σ, Δ, s_0), where S, Σ, s_0 are the same as before. Possible operations of the machine are described by Δ, which is no longer a function. Instead, Δ is a relation:

$$\Delta \subseteq (S \times \Sigma) \times (S \cup \{\texttt{yes}, \texttt{no}\}) \times \Sigma \times \{\texttt{-1, 0, +1}\}.$$

A tuple whose first two members are s and σ respectively, specifies the action of the NDTM when its current state is s and the symbol pointed at by its cursor is σ. If the number of such tuples is greater than one, the NDTM non-deterministically chooses any of them and operates accordingly.

Unlike the case of a DTM, the definition of acceptance and rejection by a NDTM is asymmetric. We say that an NDTM *accepts* an input if there is at least one sequence of choices leading to the state `yes`. An NDTM *rejects* an input if no sequence of choices can lead to `yes`.

8.2 Propositional Logic Programming

We start our complexity analysis of logic programming with the simplest case, i.e., propositional logic programming.

Theorem 224. *(implicit in [95, 156, 89]) Propositional logic programming is* P*-complete.*

Proof. Membership. Let a program P be given. Recall from Section 6.1 that the semantics of P can be defined as the least fixpoint of the immediate consequence operator $\mathbf{T}_P$) and that this least fixpoint $lfp(\mathbf{T}_P)$ can be computed in polynomial time even if the "naive" evaluation algorithm from Listing 6_2 is applied. Indeed, the number of iterations (i.e. applications of $\mathbf{T}_P$) is bounded by the number of rules plus one. Moreover, each iteration step is clearly feasible in polynomial time.

Hardness. Let A be an arbitrary language in P. Thus A is decidable by a deterministic Turing machine (DTM) T in at most $q(|I|)$ steps for some polynomial q, for any input I. We show that the computation of the Turing machine T on any input I can be simulated by a propositional logic program as follows: Let $N = q(|I|)$. W.l.o.g., we assume that the computation of T on input I takes exactly N steps.

The transition function δ of a DTM with a single tape can be represented by a table whose rows are tuples $t = \langle s, \sigma, s', \sigma', d\rangle$. Such a tuple t expresses the following if-then-rule:

> **if** at some time instant τ the DTM is in state s, the cursor points to cell number π, and this cell contains symbol σ
> **then** at instant $\tau + 1$ the DTM is in state s', cell number π contains symbol σ', and the cursor points to cell number $\pi + d$.

It is possible to describe the complete evolution of a DTM T on input string I from its initial configuration at time instant 0 to the configuration at instant N by a propositional logic program $L(T, I, N)$. To achieve this, we define the following classes of propositional atoms:

symbol$_\alpha[\tau, \pi]$ for $0 \leq \tau \leq N$, $0 \leq \pi \leq N$ and $\alpha \in \Sigma$. Intuitive meaning: at instant τ of the computation, cell number π contains symbol α.

cursor$[\tau, \pi]$ for $0 \leq \tau \leq N$ and $0 \leq \pi \leq N$. Intuitive meaning: at instant τ, the cursor points to cell number π.

state$_s[\tau]$ for $0 \leq \tau \leq N$ and $s \in S$. Intuitive meaning: at instant τ, the DTM T is in state s.

accept Intuitive meaning: T has reached state `yes`.

Let us denote by I_k the k-th symbol of the string $I = I_0 \cdots I_{|I|-1}$. The initial configuration of T on input I is reflected by the following *initialization facts* in $L(T, I, N)$:

$$\begin{aligned} symbol_{\sigma}[0,\pi] \leftarrow & \qquad \text{for } 0 \leq \pi < |I|, \text{ where } I_\pi = \sigma \\ symbol_{\sqcup}[0,\pi] \leftarrow & \qquad \text{for } |I| \leq \pi \leq N \\ cursor[0,0] \leftarrow & \\ state_{s_0}[0] \leftarrow & \end{aligned}$$

Each entry $\langle s, \sigma, s', \sigma', d\rangle$ of the transition table δ is translated into the following propositional Horn clauses, which we call the *transition rules*. We thus need the following clauses for each value of τ and π such that $0 \leq \tau < N$, $0 \leq \pi < N$, and $0 \leq \pi + d$.

$$\begin{aligned} symbol_{\sigma'}[\tau+1,\pi] &\leftarrow state_s[\tau], symbol_\sigma[\tau,\pi], cursor[\tau,\pi] \\ cursor[\tau+1,\pi+d] &\leftarrow state_s[\tau], symbol_\sigma[\tau,\pi], cursor[\tau,\pi] \\ state_{s'}[\tau+1] &\leftarrow state_s[\tau], symbol_\sigma[\tau,\pi], cursor[\tau,\pi] \end{aligned}$$

These clauses almost perfectly describe what is happening during a state transition from an instant τ to an instant $\tau + 1$. However, it should not be forgotten that those tape cells which are not changed during the transition keep their old values at instant $\tau + 1$. This must be reflected by what we term *inertia rules*. These rules are asserted for each time instant τ and tape cell numbers π, π', where $0 \leq \tau < N$, $0 \leq \pi < \pi' \leq N$, and have the following form:

$$\begin{aligned} symbol_\sigma[\tau+1,\pi] &\leftarrow symbol_\sigma[\tau,\pi], cursor[\tau,\pi'] \\ symbol_\sigma[\tau+1,\pi'] &\leftarrow symbol_\sigma[\tau,\pi'], cursor[\tau,\pi] \end{aligned}$$

Finally, a group of clauses termed *accept rules* derives the propositional atom *accept*, whenever an accepting configuration is reached.

$$accept \leftarrow state_{\mathsf{yes}}[\tau] \qquad \text{for } 0 \leq \tau \leq N.$$

Denote by L the logic program $L(T, I, N)$. Note that $\mathbf{T}_L^0 = \emptyset$ and that $\mathbf{T}_L^1$ contains the initial configuration of T at time instant 0. By construction, the least fixpoint $lfp(\mathbf{T}_L)$ of L is reached at $\mathbf{T}_L^{N+2}$, and the ground atoms added to $\mathbf{T}_L^\tau$, $2 \leq \tau \leq N + 1$, i.e., those in $\mathbf{T}_L^\tau \setminus \mathbf{T}_L^{\tau-1}$, describe the configuration of T on the input I at the time instant $\tau - 1$. The least fixpoint $lfp(\mathbf{T}_L)$ contains *accept* if and only if an accepting configuration has been reached by T in at most N computation steps. Hence, $L(T, I, N)) \models accept$ if and only if T has reached an accepting state within $q(N)$ steps with $N = |I|$.

The translation from I to $L(T, I, N)$ with $N = q(|I|)$ is very simple and is clearly feasible in logarithmic space, since all rules of $L(T, I, N))$ can be generated independently of each other and each has size logarithmic in $|I|$; note that the numbers τ and π have $O(\log |I|)$ bits, while all other syntactic constituents of a rule have constant size. We have thus shown that every language A in P is logspace reducible to propositional logic programming. Hence, propositional logic programming is P-hard. □

Note that the the polynomial-time upper bound can be even sharpened to a *linear time* upper bound, as was shown in [57, 120]. As far as the lower bound is concerned, the above proof could be greatly simplified by using reductions from other P-complete problems like, e.g., from the monotone circuit value problem (see [126]). However, the proof from first principles provides a basic framework from which further results will be derived by slight adaptations in the sequel.

An interesting kind of syntactical restrictions on programs is obtained by restricting the number of atoms in the body. Let LP(k) denote logic programming where each clause has at most k atoms in the body. Then, by results in [156, 90], one easily obtains that LP(1) is NL-complete. Indeed, the correspondence between the well-known NL-complete reachability problem of directed graphs and LP(1) is immediate. On the other hand, observe that the DTM encoding in the proof of Theorem 224 can be easily modified to programs in LP(2). Hence, LP(k) for any $k \geq 2$ is P-complete.

8.3 Conjunctive Queries

For the complexity analysis of conjunctive queries (CQs), we restrict ourselves to *boolean* conjunctive queries (cf. Definition 25), i.e. the queries under consideration are of the form

$$Q : ans() \leftarrow r_1(\mathbf{u}_1) \wedge \ldots \wedge r_n(\mathbf{u}_n)$$

where $n \geq 0$; $r_1, \ldots, r_n$ are (not necessarily distinct) extensional relation symbols and $ans()$ is a 0-ary intensional relation symbol; moreover, $\mathbf{u}_1, \ldots, \mathbf{u}_n$ are lists of terms of appropriate length.[28]

Query Q evaluates to *true* if there exists a substitution θ such that $r_i(\mathbf{u}_i)\theta \in D_{in}$ for all $i \in \{1, \ldots, n\}$; otherwise, the query evaluates to *false*.

Theorem 225. *The program complexity of conjunctive queries is* NP*-complete [37].*

This problem appears as Problem SR31 in Garey and Johnson's book [76].

Proof. Membership. We guess an assignment for each variable of the query and check whether all the resulting ground atoms in the query body exist in D_{in}. This check is obviously feasible in polynomial time.

Hardness. We reduce the NP-complete 3-SAT problem to our problem. For this purpose, we consider the following input database (over a ternary relation symbol c and a binary relation symbol v) as fixed:

$$\begin{aligned} D_{in} = \{\, & c(1,1,1),\ c(1,1,0),\ c(1,0,1),\ c(1,0,0), \\ & c(0,1,1),\ c(0,1,0),\ c(0,0,1),\ v(1,0),\ v(0,1) \,\} \end{aligned}$$

[28] Note that without this restriction to boolean CQs, the head literal of a conjunctive query would have the form $ans(\mathbf{u})$, where $\mathbf{u}$ is a list of terms. However, as far as the complexity of query evaluation is concerned, this difference is inessential.

Now let an instance of the 3-SAT problem be given through the 3-CNF formula

$$\Phi = \bigwedge_{i=1}^{n} l_{i,1} \vee l_{i,2} \vee l_{i,3}$$

over propositional atoms $x_1, \ldots, x_k$. Then we define a conjunctive query Q as follows:

$$ans() \leftarrow c(l^*_{1,1}, l^*_{1,2}, l^*_{1,3}), \ldots, c(l^*_{1,1}, l^*_{1,2}, l^*_{1,3}), v(x_1, \bar{x}_i), \ldots, v(x_k, \bar{x}_k)$$

where $l^* = x$ if $l = x$, and $l^* = \bar{x}$ if $l = \neg x$. By slight abuse of notation, we thus use x_i to denote either a propositional atom (in Φ) or a first-order variable (in Q).

It is straightforward to verify that the 3-CNF formula Φ is satisfiable if and only if $D_{in} \cup Q \models ans()$ holds. □

8.4 First-Order Queries

Recall from Definition 6 that we are mainly considering first-order queries *without* equality here. Hence, atoms are of the form $r(\mathbf{u})$ for some relational symbol r from the signature $\mathcal{L}$ and a list of terms $\mathbf{u}$ whose length corresponds to the arity of r. Compound formulae are constructed from simpler ones by means of quantification (with $\forall$ and $\exists$) and the conjuncts $\wedge, \vee, \neg$. Note however that the following complexity result also holds if we consider first-order predicate logic *with* equality.

Theorem 226. *(implicit in [90, 156]) First-order queries are program-complete for* PSPACE*. Their data complexity is in the class* AC^0*, which contains the languages recognized by unbounded fan-in circuits of polynomial size and constant depth [94].*

Proof. We only prove the PSPACE-completeness. Actually, we show that the combined complexity is PSPACE-complete. However, by the considerations in Section 8.1, the PSPACE-completeness of the program-complexity follows immediately.

Membership. Let φ be a first-order sentence and D_{in} be an input database with domain elements **dom**. Let $n = |D_{in}| + |\mathbf{dom}|$ and $m = |\varphi|$. There are maximal m alternations of $\forall$ and $\exists$ (because the number of variables in φ is less than m), thus the evaluation has maximal m nested loops. In each loop we need to store

1. the position of the currently processed variable, and
2. for each variable with the assigned value in **dom**, its position in **dom**.

The space for these operations is then $O(m)$, hence in PSPACE.

Hardness. The PSPACE-hardness can be shown by a reduction from the QBF problem. Assume that ϕ is the quantified Boolean formula

$$Qx_1 \ldots Qx_n \alpha(x_1, \ldots, x_n)$$

where Q is either $\forall$ or $\exists$ and α is a quantifier-free Boolean formula.

We first define the signature $\mathcal{L} = \{\mathsf{istrue}, \mathsf{isequal}, \mathsf{or}, \mathsf{and}, \mathsf{not}\}$ The predicates $\mathsf{and}, \mathsf{or}, \mathsf{not}$ are used to define the operators of Boolean algebra. The predicate istrue is unary and defines the truth value *true*, whereas the predicate $\mathsf{isequal}$ is binary and defines the equality of two values.

For each sub-formula β of α, we define a quantifier-free, first-order formula $T_\beta(z_1, \ldots, z_n, x)$ with the following intended meaning: if the variables x_i have the truth value z_i, then the formula $\beta(x_1, \ldots, x_n)$ evaluates to the truth value x. Note that $T_\beta(z_1, \ldots, z_n, x)$ can be defined inductively w.r.t. the structure of α as follows:

Case $\beta =$

$$\begin{aligned}
x_i \text{ (with } 1 \leq i \leq n): &\quad T_\beta(\bar{z}, x) \equiv \mathsf{isequal}(x, z_i) \\
\neg\beta': &\quad T_\beta(\bar{z}, x) \equiv \exists t_1 T_{\beta'}(\bar{z}, t_1) \wedge \mathsf{not}(x, t_1) \\
\beta_1 \wedge \beta_2: &\quad T_\beta(\bar{z}, x) \equiv \exists t_1, t_2\ T_{\beta_1}(\bar{z}, t_1) \wedge T_{\beta_2}(\bar{z}, t_2) \wedge \mathsf{and}(t_1, t_2, x) \\
\beta_1 \vee \beta_2: &\quad T_\beta(\bar{z}, x) \equiv \exists t_1, t_2\ T_{\beta_1}(\bar{z}, t_1) \wedge T_{\beta_2}(\bar{z}, t_2) \wedge \mathsf{or}(t_1, t_2, x)
\end{aligned}$$

Finally, we define the input database as $D_{in} = \{\mathsf{istrue}(1), \mathsf{isequal}(0,0), \mathsf{isequal}(1,1), \mathsf{or}(1,1,1), \mathsf{or}(1,0,1), \mathsf{or}(0,1,1), \mathsf{or}(0,0,0), \mathsf{and}(1,1,1), \mathsf{and}(1,0,0), \mathsf{and}(0,1,0), \mathsf{and}(0,0,0), \mathsf{not}(1,0), \mathsf{not}(0,1)\}$, and the first-order query φ is defined as follows:

$$\varphi \equiv \exists x\ Qz_1 \ldots Qz_n\ \mathsf{istrue}(x) \wedge T_\alpha(\bar{z}, x)$$

It is then straightforward to show that the formula ϕ is satisfiable if and only if the evaluation of φ returns *true*. □

8.5 Unification

Unification is used extensively in several areas of computer science, including theorem proving, database systems, natural language processing, logic programming, computer algebra, and program verification. We briefly introduce the basic notions here. Additional material can be found in [13] or [52].

Recall from Definition 50 that two atoms or terms s and t are called *unifiable* if there exists a substitution σ with $s\sigma = t\sigma$. Such a substitution σ is called a *unifier* of s and t. A unifier σ of s and t is called *most general* if any other unifier σ' of s and t is an instance of σ, i.e., there exists a substitution η with $\sigma' = \sigma\eta$.

The **unification problem** is the following decision problem: given terms s and t, are they unifiable?

Robinson described an algorithm that solves this problem and, if the answer is positive, computes a most general unifier of given two terms (see [138]). His algorithm had exponential time and space complexity mainly because of the representation of terms by strings of symbols. However, by using more sophisticated data structures (like directed acyclic graphs), unification was later shown to be feasible in polynomial time. In fact, even linear time suffices (see [116, 130]).

Theorem 227. *([59, 162, 60]) The unification problem is* P*-complete.*

Note that in the above definition of unification, a unifier σ of s and t makes the terms $s\sigma$ and $t\sigma$ syntactically equal. More generally, one may look for so-called *E-unifiers* which make the terms $s\sigma$ and $t\sigma$ equal modulo some equational theory.

Equational theories are usually presented by finite sets E of identities of the form $l = r$, which are referred to as *equational axioms*. The *equational theory* $Th(E)$ presented by E is the smallest congruence relation over terms (for a given signature $\mathcal{L}$) containing E and closed under substitutions, i.e., $Th(E)$ is the smallest congruence containing all pairs $l\rho = r\rho$, where $l = r$ is in E and ρ is a substitution. We write $s =_E t$ to denote that the pair (s, t) of terms is a member of $Th(E)$. In this survey, we only consider the equationl axioms (for some function symbol f) depicted in Figure 7.

Fig. 7 Equational Axioms

Associativity	A(f)	$f(f(x,y),z) = f(x,f(y,z))$
Commutativity	C(f)	$f(x,y) = f(y,x)$
Idempotence	I(f)	$f(x,x) = x$
Existence of Unit	U(f)	$f(x,1) = x,\ f(1,x) = x$

An *E-unifier* of s and t is a substitution ρ such that $s\rho =_E t\rho$ holds. Whenever such an E-unifier exists, we say that the terms s and t are E-unifiable. For every equational theory E, the **E-unification problem** is the following decision problem: given terms s and t, are they E-unifiable, i.e., is there a substitution ρ, such that $s\rho =_E t\rho$?

By examining the signature $\mathcal{L}$ over which the terms of unification problems in the theory $Th(E)$ have been built, we distinguish between three different kinds of E-unification. Let $sig(E)$ be the set of all function and constant symbols occurring in the equational axioms of E. If $\mathcal{L} = sig(E)$ holds, then we speak about *elementary E-unification*. If the signature $\mathcal{L}$ contains in addition free constant symbols, but no free function symbols, then we speak about *E-unification with constants*. Finally, if the signature $\mathcal{L}$ contains free function symbols of arbitrary arities, then we speak about *general* E-*unification*. Figure 8 summarizes the complexity results for some equational unification decision problems. The entry $\longleftarrow$ means that upper bounds carry over to the simpler case.

8.6 Positive Definite Rule Sets

Let us now turn to datalog. We first consider the data complexity.

Theorem 228. *(implicit in [156, 89]) Datalog is data complete for* P.

Fig. 8 Complexity Results for Equational Unification Decision Problems

theory	complexity		
	elementary	with constants	general
$\emptyset$	⟵	⟵	linear [116, 130]
A	⟵	NP-hard [20] and in NEXPTIME [131]	NP-hard
C		NP-complete (folkore, see e.g. [76, 13])	NP-complete
AC	NP-complete	NP-complete	NP-complete [96]
ACI	⟵	in P	NP-complete [96]
ACIU	⟵	in P	NP-complete [121]

Proof. (Sketch) Grounding P on an input database D yields polynomially many clauses in the size of D; hence, the complexity of propositional logic programming is an upper bound for the data complexity.

The P-hardness can be shown by writing a simple datalog *meta-interpreter* for propositional LP(k), where k is a constant.

Represent rules $A_0 \leftarrow A_1, \ldots, A_i$, where $0 \leq i \leq k$, by tuples $\langle A_0, \ldots, A_i \rangle$ in an $(i+1)$-ary relation R_i on the propositional atoms. Then, a program P in LP(k) which is stored this way in a database $D(P)$ can be evaluated by a fixed datalog program $P_{MI}(k)$ which contains for each relation R_i, $0 \leq i \leq k$, a rule

$$T(X_0) \leftarrow T(X_1), \ldots, T(X_i), R_i(X_0, \ldots, X_i).$$

$T(x)$ intuitively means that atom x is true. Then, $P \models A$ just if $P_{MI} \cup P(D) \models T(A)$. P-hardness of the data complexity of datalog is then immediate from Theorem 224. □

The program complexity is exponentially higher.

Theorem 229. *(implicit in [156, 89]) Datalog is program complete for* EXPTIME.

Proof. Membership. Grounding P on D leads to a propositional program P' whose size is exponential in the size of the fixed input database D. Hence, by Theorem 224, the program complexity is in EXPTIME.

Hardness. In order to prove EXPTIME-hardness, we show that if a DTM T halts in less than $N = 2^{n^k}$ steps on a given input I where $|I| = n$, then T can be simulated by a datalog program over a fixed input database D. In fact, we use $D_\emptyset$, i.e., the empty database with the universe $U = \{0, 1\}$.

We employ the scheme of the DTM encoding into logic programming from Theorem 224, but use the predicates $symbol_\sigma(X, Y)$, $cursor(X, Y)$ and $state_s(X)$

instead of the propositional letters $symbol_\sigma[X,Y]$, $cursor[X,Y]$ and $state_s[X]$ respectively. The time points τ and tape positions π from 0 to 2^m-1, $m=n^k$, are represented by m-ary tuples over U, on which the functions $\tau+1$ and $\pi+d$ are realized by means of the successor $Succ^m$ from a linear order $\leq^m$ on U^m.

For an inductive definition, suppose $Succ^i(\mathbf{X},\mathbf{Y})$, $First^i(\mathbf{X})$, and $Last^i(\mathbf{X})$ tell the successor, the first, and the last element from a linear order $\leq^i$ on U^i, where $\mathbf{X}$ and $\mathbf{Y}$ have arity i. Then, use rules

$$\begin{aligned} Succ^{i+1}(Z,\mathbf{X},Z,\mathbf{Y}) &\leftarrow Succ^i(\mathbf{X},\mathbf{Y}) \\ Succ^{i+1}(Z,\mathbf{X},Z',\mathbf{Y}) &\leftarrow Succ^1(Z,Z'), Last^i(\mathbf{X}), First^i(\mathbf{Y}) \\ First^{i+1}(Z,\mathbf{X}) &\leftarrow First^1(Z), First^i(\mathbf{X}) \\ Last^{i+1}(Z,\mathbf{X}) &\leftarrow Last^1(z), Last^i(\mathbf{X}) \end{aligned}$$

Here $Succ^1(X,Y)$, $First^1(X)$, and $Last^1(X)$ on $U^1=U$ must be provided. For our reduction, we use the usual ordering $0\leq^1 1$ and provide those relations by the ground facts $Succ^1(0,1)$, $First^1(0)$, and $Last^1(1)$.

The initialization facts $symbol_\sigma[0,\pi]$ are readily translated into the datalog rules

$$symbol_\sigma(\mathbf{X},\mathbf{t}) \leftarrow First^m(\mathbf{X}),$$

where $\mathbf{t}$ represents the position π, and similarly the facts $cursor[0,0]$ and $state_{s_0}[0]$. The remaining initialization facts $symbol_\sqcup[0,\pi]$, where $|I|\leq\pi\leq N$, are translated to the rule

$$symbol_\sqcup(\mathbf{X},\mathbf{Y}) \leftarrow First^m(\mathbf{X}),\ \leq^m(\mathbf{t},\mathbf{Y})$$

where $\mathbf{t}$ represents the number $|I|$; the order $\leq^m$ is easily defined from $Succ^m$ by two clauses

$$\begin{aligned} \leq^m(\mathbf{X},\mathbf{X}) &\leftarrow \\ \leq^m(\mathbf{X},\mathbf{Y}) &\leftarrow Succ^m(\mathbf{X},\mathbf{Z}),\ \leq^m(\mathbf{Z},\mathbf{Y}) \end{aligned}$$

The transition and inertia rules are easily translated into datalog rules. For realizing $\tau+1$ and $\pi+d$, use in the body atoms $Succ^m(\mathbf{X},\mathbf{X}')$. For example, the clause

$$symbol_{\sigma'}[\tau+1,\pi] \leftarrow state_s[\tau], symbol_\sigma[\tau,\pi], cursor[\tau,\pi]$$

is translated into

$$symbol_{\sigma'}(\mathbf{X}',\mathbf{Y}) \leftarrow state_s(\mathbf{X}), symbol_\sigma(\mathbf{X},\mathbf{Y}), cursor(\mathbf{X},\mathbf{Y}), Succ^m(\mathbf{X},\mathbf{X}').$$

The translation of the accept rules is straightforward.

For the resulting datalog program P', it holds that $P'\cup D_\emptyset \models accept$ if and only if T accepts input I in at most N steps. It is easy to see that P' can be constructed from T and I in logarithmic space. Hence, datalog has EXPTIME-hard program complexity. □

Note that, instead of using a generic reduction, the hardness part of this theorem can also be obtained by applying complexity upgrading techniques [127, 17].

8.7 Stratified Definite Rule Sets (stratified Datalog)

Recall from Definition 21 that a *normal clause* is a rule of the form

$$A \leftarrow L_1, \ldots, L_m \qquad (m \geq 0)$$

where A is an atom and each L_i is a literal. A *normal logic program* is a finite set of normal clauses. As was explained in Section 7, if a normal logic program P is *stratified*, then the clauses of P can be partitioned into disjoint sets $S_1, \ldots, S_n$ s.t. the semantics of P is computed by successively computing fixpoints of the immediate consequence operators $\mathbf{T}_{S_1}$, ..., $\mathbf{T}_{S_n}$. More precisely, let $\mathbf{I}_0$ be the initial instance over the extensional predicate symbols of P and let $\mathbf{I}_i$ (with $1 \leq i \leq n$) be defined as follows:

$$\mathbf{I}_1 := \mathbf{T}^{\omega}_{S_1}(\mathbf{I}_0), \;\; \mathbf{I}_2 := \mathbf{T}^{\omega}_{S_2}(\mathbf{I}_1), \;\; \ldots, \;\; \mathbf{I}_n := \mathbf{T}^{\omega}_{S_n}(\mathbf{I}_{n-1})$$

Then the semantics of program P is given through the set $\mathbf{I}_n$.

Note that in the propositional case, $\mathbf{I}_n$ is clearly polynomially computable. Hence, stratified negation does not increase the complexity. Analogously to Theorems 224, 228, and 229, we thus have:

Theorem 230. *(implicit in [9]) Stratified propositional logic programming with negation is* P*-complete. Stratified datalog with negation is data complete for* P *and program complete for* EXPTIME*.*

Note that nonrecursive logic programs with negation are trivially stratified since, in this case, the dependency graph is acyclic and the clauses can be simply partitioned into strata according to a topological sort of the head predicates. Actually, any nonrecursive datalog program with negation can be easily rewritten to an equivalent first-order query and vice versa (cf. Theorem 243). Hence, analogously to Theorem 226, we have the following complexity results.

Theorem 231. *(implicit in [90, 156]) Nonrecursive propositional logic programming with negation is* P*-complete. Nonrecursive datalog with negation is program complete for* PSPACE*. Its data complexity is in the class* AC^0*, which contains the languages recognized by unbounded fan-in circuits of polynomial size and constant depth [94].*

8.8 Well-Founded and Inflationary Semantics

In Section 7.6, an *alternating fixpoint* procedure for computing the well-founded semantics was presented. This computation aims at iteratively building up a set of negative conclusions $\tilde{A}$ of a logic program. It starts with an underestimate of the negative conclusions $\tilde{I} = \emptyset$ from which an overestimate $\widetilde{\mathbf{S}}_P(\tilde{I})$ is computed which is in turn used to move back to an underestimate $\mathbf{A}_P(\tilde{I}) := \widetilde{\mathbf{S}}_P(\widetilde{\mathbf{S}}_P(\tilde{I}))$, etc. The iteration of this monotonic transformation $\mathbf{A}_P$ leads to a least fixpoint $\tilde{A} := \mathit{lfp}(\mathbf{A}_P)$. The well-founded semantics of the program P is given through the set $\tilde{A} \cup \widetilde{\mathbf{S}}_P(\tilde{A})$. Clearly, for a propositional logic program, this fixpoint computation can be done in polynomial time. Together with Theorem 224, we thus get

Theorem 232. *(implicit in [154, 155]) Propositional logic programming with negation under well-founded semantics is* P*-complete. Datalog with negation under well-founded semantics is data complete for* P *and program complete for* EXPTIME*.*

As was mentioned in Section 5.3.5, the *inflationary semantics* is defined via the *inflationary operator* $\widetilde{\mathbf{T}}_P$, which is defined as $\widetilde{\mathbf{T}}_P(I) = I \cup \mathbf{T}_{P^I}(I)$ (cf. Definition 189). Clearly, the limit $\widetilde{\mathbf{T}}_P^{\omega}(\mathbf{I})$ is computable in polynomial time for a propositional program P. Therefore, by the above results, we have

Theorem 233. *([5]; implicit in [86]) Propositional logic programming with negation under inflationary semantics is* P*-complete. Datalog with negation under inflationary semantics is data complete for* P *and program complete for* EXPTIME*.*

8.9 Stable Model Semantics

An interpretation I of a normal logic program P is a *stable model* of P [77] if I is the (unique) minimal Herbrand model of P^I. As was mentioned in Section 5.3.2, a program P may have zero, one, or multiple stable models.

Note that every stratified program P has a unique stable model, and its stratified and stable semantics coincide. Unstratified rules increase the complexity as the following theorem illustrates.

Theorem 234. *([115], [22]) Given a propositional normal logic program P, deciding whether P has a stable model is* NP*-complete.*

Proof. Membership. Clearly, P^I is polynomial time computable from P and I. Hence, a stable model M of P can be guessed and checked in polynomial time.

Hardness. Modify the DTM encoding in the proof of Theorem 224 for a non-deterministic Turing machine T as follows.

1. For each state s and symbol σ, introduce atoms $B_{s,\sigma,1}[\tau],\ldots,\ B_{s,\sigma,k}[\tau]$ for all $1 \leq \tau < N$ and for all transitions $\langle s, \sigma, s_i, \sigma'_i, d_i\rangle$, where $1 \leq i \leq k$.
2. Add $B_{s,\sigma,i}[\tau]$ in the bodies of the transition rules for $\langle s, \sigma, s_i, \sigma'_i, d_i\rangle$.
3. Add the rule

$$B_{s,\sigma,i}[\tau] \leftarrow \neg B_{s,\sigma,1}[\tau], \ldots, \neg B_{s,\sigma,i-1}[\tau], \neg B_{s,\sigma,i+1}[\tau], \ldots, \neg B_{s,\sigma,k}[\tau].$$

   Intuitively, these rules non-deterministically select precisely one of the possible transitions for s and σ at time instant τ, whose transition rules are enabled via $B_{s,\sigma,i}[\tau]$.
4. Finally, add a rule

$$\mathit{accept} \leftarrow \neg \mathit{accept}.$$

   It ensures that *accept* is true in every stable model.

It is immediate from the construction that the stable models M of the resulting program correspond to the accepting runs of T. □

Notice that, as shown in [115], the hardness part of this result holds even if all rules in P have exactly one literal in the body and, moreover, this literal is negative. As an easy consequence of Theorem 234, we obtain

Theorem 235. *([115]; [142] and [102]) Propositional logic programming with negation under stable model semantics is* co-NP*-complete. Datalog with negation under stable model semantics is data complete for* co-NP *and program complete for* co-NEXPTIME.

The co-NEXPTIME result for program complexity, which is not stated in [142], follows from an analogous result for datalog under fixpoint models in [102] and a simple, elegant transformation of this semantics to the stable model semantics [142].

8.10 Disjunctive Rule Sets

A *disjunctive logic program* is a set of clauses

$$A_1 \vee \cdots \vee A_k \leftarrow L_1, \ldots, L_m \text{ with } (k \geq 1, m \geq 0),$$

where each A_i is an atom and each L_j is a literal, see [112, 119]. The semantics of negation-free disjunctive logic programs is based on *minimal* Herbrand models. As was pointed out in Section 5.1.4, in general, disjunctive logic programs do not have a unique minimal Herbrand model.

Denote by $\mathrm{MM}(P)$ the set of all minimal Herbrand models of P. The *Generalized Closed World Assumption (GCWA)* [118] for negation-free P amounts to the meaning $\mathcal{M}_{GCWA}(P) = \{L \mid \mathrm{MM}(P) \models L\}$.

Theorem 236. *([62, 64]) Let P be a propositional negation-free disjunctive logic program and A be a propositional atom. Deciding whether $P \models_{GCWA} A$ is* co-NP*-complete.*

Proof. (Sketch) It is not hard to argue that for an atom A, we have $P \models_{GCWA} A$ if and only if $P \models_{PC} A$, where $\models_{PC}$ is the classical logical consequence relation. In addition, any set of clauses can be clearly represented by a suitable disjunctive logic program. Hence, the co-NP-hardness follows from the well-known NP-completeness of SAT. □

Stable negation naturally extends to disjunctive logic programs, by adopting that I is a *(disjunctive) stable model* of a disjunctive logic program P if and only if $I \in \mathrm{MM}(P^I)$ [133, 78]. The disjunctive stable model semantics subsumes the disjunctive stratified semantics [132]. For well-founded semantics, no such natural extension is known; the semantics in [24, 134] are the most appealing attempts in this direction.

Clearly, P^I is easily computed, and $P^I = P$ if P is negation-free. Thus,

Theorem 237. *([63–65]) Propositional disjunctive logic programming under stable model semantics is Π_2^p complete. Disjunctive datalog under stable model semantics is data complete for Π_2^p and program complete for* co-NEXPTIME$^{\mathsf{NP}}$.

8.11 Rule Sets with Function Symbols

If we allow function symbols, then logic programs become undecidable.

Theorem 238. *([8, 151]) Logic programming is r.e.-complete.*[29]

Proof. (Sketch) On the one hand, the undecidability can be proved by a simple encoding of (the halting problem of) Turing machines similar to the encoding in the proof of Theorem 229 (use terms $f^n(c)$, $n \geq 0$, for representing cell positions and time instants). This reduction from the halting problem also establishes the r.e.-hardness. On the other hand, the least fixpoint $lfp(\mathbf{T}_P)$ of any logic program P is clearly a recursively enumerable set. This shows the r.e.-membership and, thus, in total, the r.e.-completeness of logic programming. □

A natural decidable fragment of logic programming with functions are *nonrecursive programs*. Their complexity is characterized by the following theorem.

Theorem 239. *([47]) Nonrecursive logic programming is* NEXPTIME*-complete.*

Proof. (Sketch) The NEXPTIME-membership is established by applying SLD-resolution with constraints. The size of the derivation turns out to be exponential. The NEXPTIME-hardness is proved by reduction from the tiling problem for the square $2^n \times 2^n$. □

8.12 Expressive Power

The expressive power of query languages such as datalog is a topic common to database theory [2] and finite model theory [61] that has attracted much attention by both communities. By the expressive power of a (formal) *query language*, we understand the set of all queries expressible in that language.

In general, a *query* q defines a mapping $\mathcal{M}_q$ that assigns to each suitable input database D_{in} (over a fixed input schema) a result database $D_{out} = \mathcal{M}_q(D_{in})$ (over a fixed output schema); more logically speaking, a query defines global relations [85]. For reasons of representation independence, a query should, in addition, be *generic*, i.e., invariant under isomorphisms. This means that if τ is a permutation of the domain $Dom(\mathcal{D})$, then $\mathcal{M}(\tau(D_{in})) = \tau(D_{out})$. Thus, when we speak about queries, we always mean generic queries.

Formally, the *expressive power* of a query language Q is the set of mappings $\mathcal{M}_q$ for all queries q expressible in the language Q by some *query expression* (program) E; this syntactic expression is commonly identified with the semantic query it defines, and simply (in abuse of definition) called a query.

There are two important research tasks in this context. The first is comparing two query languages Q_1 and Q_2 in their expressive power. One may prove, for instance, that $Q_1 \subsetneqq Q_2$, which means that the set of all queries expressible in

[29] In the context of recursion theory, reducibility of a language (or problem) L_1 to L_2 is understood in terms of a Turing reduction, i.e., L_1 can be decided by a DTM with oracle L_2, rather than logarithmic-space reduction.

Q_1 is a proper subset of the queries expressible in Q_2, and hence, Q_2 is strictly more expressive than Q_1. Or one may show that two query languages Q_1 and Q_2 have the same expressive power, denoted by $Q_1 = Q_2$, and so on.

The second research task, more related to complexity theory, is determining the absolute expressive power of a query language. This is mostly achieved by proving that a given query language Q is able to express exactly all queries whose evaluation complexity is in a complexity class $\mathcal{C}$. In this case, we say that Q *captures* $\mathcal{C}$ and write simply $Q = \mathcal{C}$. The *evaluation complexity* of a query is the complexity of checking whether a given atom belongs to the query result, or, in the case of Boolean queries, whether the query evaluates to *true* [156, 85].

Note that there is a substantial difference between showing that the query evaluation problem for a certain query language Q is $\mathcal{C}$-complete and showing that Q captures $\mathcal{C}$. If the evaluation problem for Q is $\mathcal{C}$-complete, then *at least one* $\mathcal{C}$-hard query is expressible in Q. If Q captures $\mathcal{C}$, then Q expresses *all* queries evaluable in $\mathcal{C}$ (including, of course, all $\mathcal{C}$-hard queries).

To prove that a query language Q captures a machine-based complexity class $\mathcal{C}$, one usually shows that each $\mathcal{C}$-machine with (encodings of) finite structures as inputs that computes a generic query can be represented by an expression in language Q. There is, however, a slight mismatch between ordinary machines and logical queries. A Turing machine works on a string encoding of the input database D. Such an encoding provides an implicit *linear order* on D, in particular, on all elements of the universe U_D. The Turing machine can take profit of this order and use this order in its computations (as long as genericity is obeyed). On the other hand, in logic or database theory, the universe U_D is a pure set and thus unordered. For "powerful" query languages of inherent non-deterministic nature at the level of NP this is not a problem, since an ordering on U_D can be non-deterministically guessed. However, for many query languages, in particular, for those corresponding to complexity classes below NP, generating a linear order is not feasible. Therefore, one often assumes that a linear ordering of the universe elements is predefined, i.e., given explicitly in the input database. More specifically, by *ordered databases* or *ordered finite structures*, we mean databases whose schemas contain special relation symbols *Succ*, *First*, and *Last*, that are always interpreted such that $Succ(x, y)$ is a successor relation of some linear order and $First(x)$ determines the first element and $Last(x)$ the last element in this order. The importance of predefined linear orderings becomes evident in the next two theorems.

Before coming to the theorems, we must highlight another small mismatch between the Turing machine and the datalog setting. A Turing machine can consider each input bit independently of its value. On the other hand, a plain datalog program is not able to detect that some atom is *not* a part of the input database. This is due to the representational peculiarity that only positive information is present in a database, and that the negative information is understood via the closed world assumption. To compensate this deficiency, we will slightly augment the syntax of datalog. *Throughout this section, we will assume that input predicates may appear negated in datalog rule bodies; the resulting lan-*

guage is datalog$^+$. This extremely limited form of negation is much weaker than stratified negation, and could be easily circumvented by adopting a different representation for databases.

The difference between unordered and ordered databases becomes apparent in the next two theorems:

Theorem 240. *(cf. [36])* *datalog*$^+ \subsetneq \mathsf{P}$.

Proof. (Hint.) Show that there exists no datalog$^+$ program P that can tell whether the universe U of the input database has an even number of elements. □

Theorem 241. *([125, 80]; implicit in [156, 89, 106])* *On ordered databases, datalog*$^+$ *captures* P.

Proof. (Sketch) By Theorem 230, query answering for a fixed datalog$^+$ program is in P. It thus remains to show that each polynomial-time DTM T on finite input databases D can be simulated by a datalog$^+$ program. This is shown by a simulation similar to the ones in the proofs of Theorems 224 and 229. □

Next, we compare the expressive power of nonrecursive datalog and, in particular, nonrecursive range-restricted datalog with well-known database query languages. Recall that in datalog with negation, the rules are of the form:

$$q : S(\boldsymbol{x}_0) \leftarrow L_1, \ldots, L_m.$$

where $m \geq 0$ and S is an intensional predicate symbol. Each L_i is an atom $R_i(\boldsymbol{x}_i)$ or a negated atom $\neg R_i(\boldsymbol{x}_i)$. $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_m$ are vectors of variables or constants (from **dom**). Moreover, this rule is *range-restricted* if every variable in $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_m$ occurs in some unnegated atom $L_i = R_i(\boldsymbol{x}_i)$ in the body.

A datalog program is called range-restricted if all its rules are range-restricted. Nonrecursive range-restricted datalog is referred to as *nr-datalog*$^\neg$. An nr-datalog$^\neg$ *query* is a query defined by some nr-datalog$^\neg$ program with a specified target relation.

Note that equality may be incorporated into nr-datalog$^\neg$ by permitting literals of the form $s = t$ and $s \neq t$ for terms s and t. However, as the following proposition shows, any nr-datalog$^\neg$ program with equality can be simulated by an nr-datalog$^\neg$ program not using equality.

Proposition 242. *Any nr-datalog*$^\neg$ *program with equality can be simulated by an nr-datalog*$^\neg$ *program not using equality.*

Proof. Assume that an nr-datalog$^\neg$ contains literals of the form $s = t$ and $s \neq t$. It suffices to describe the construction for a single nr-datalog$^\neg$ rule.

We consider the unnegated equalities $s = t$ first. We can easily get rid of equalities where one of the terms (say s) is a variable and the other one is a constant. In this case, we simply replace all occurrences of the variable s by the constant t. It remains to consider the case of equalities $s = t$ where both s

and t are variables. In this case, we can partition all the variables occurring in equations into l disjoint sets $C_1, \ldots, C_l$, such that for each $C_i = \{x_{i1}, x_{i2} \ldots, x_{ik}\}$ where $(1 \leq i \leq l)$, the equalities $x_{i1} = x_{i2} = \ldots = x_{ik}$ can be derived from the body of the rule. Without loss of generality, we choose x_{i1} from each partition C_i and whenever a variable $x \in C_i$ occurs in the rule, we replace it with x_{i1}. We apply this transformation also to all literals of the form $s \neq t$: if $s \in C_i$ or $t \in C_i$, we replace it with x_{i1} too.

After this transformation, we obtain a rule with equality literals only in negated form $s \neq t$ as follows:

$$S(u) \leftarrow L_1, \ldots, L_n, D_1, \ldots, D_m.$$

where every $D_i, (1 \leq i \leq m)$ is an inequality $s_i \neq t_i$. Now let $G_1, \ldots, G_m$ be m new relation symbols. Then we add the following rules:

$$\begin{array}{l} G_1(u_1) \leftarrow L_1, \ldots, L_n, s_1 = t_1 \\ G_2(u_2) \leftarrow L_1, \ldots, L_n, s_2 = t_2 \\ \ldots \\ G_m(u_m) \leftarrow L_1, \ldots, L_n, s_m = t_m \end{array}$$

Where $u_{i(1 \leq i \leq m)} = var(L_1) \cup \ldots \cup var(L_n)$. With the above method of eliminating the equality literals, we can easily obtain m new rules with rule heads $G'_1, \ldots, G'_m$ in which no equality literal occurs. Finally we rewrite the original rule as follows:

$$S(u) \leftarrow L_1, \ldots, L_n, \neg G'_1, \ldots, \neg G'_m.$$

Now all equality literals (either unnegated or negated) have indeed been removed from our nr-datalog$^\neg$ program. □

Theorem 243. *(cf. [2]) Nonrecursive range-restricted datalog with negation = relational algebra = domain-independent relational calculus. Nonrecursive datalog with negation = first-order logic (without function symbols).*

Proof. We prove here only the first equivalence: Nonrecursive range-restricted datalog (nr-datalog$^\neg$) and relational algebra have the equivalent expressive power. Proofs of the other equivalence results can be found in Chapter 5 of [2].
"⇒": We have to show that, given any range-restricted nr-datalog$^\neg$ program, there is an equivalent relational algebra expression. By Proposition 242, we may restrict ourselves w.l.o.g. to nr-datalog$^\neg$ programs not using equality. It suffices to show how the construction of an equivalent relational algebra expression works for a single nr-datalog$^\neg$ rule. Since relational algebra is closed under composition, the simulation of the program with a relational algebra expression is then straightforward.

Consider an nr-datalog$^\neg$ rule of the following form:

$$S(u) \leftarrow P_1(u_1), \ldots, P_n(u_n), \neg N_1(v_1), \ldots, \neg N_m(v_m).$$

where the P_i's are unnegated atoms and the $\neg N_j$'s are negated ones. We need first to construct a new relation A as $A = P_1 \bowtie \ldots \bowtie P_n$. Now the relational algebra expression for S is as follows:

$S = \pi_u(P_1 \bowtie \ldots \bowtie P_n \bowtie (\pi_{v_1} A - N_1) \bowtie \ldots \bowtie (\pi_{v_m} A - N_m))$

Note that if the same relation symbol (for example S) occurs in more than one rule head (for example $r_1, \ldots, r_l$), then we have to rename the algebra expressions for $r_1, \ldots, r_l$ as $S_1, \ldots, S_l$ and thus S can be written as $S = S_1 \cup \ldots \cup S_l$.

Due to the ordering of the rules in the program, we can start with the rules with the smallest ordering number and simulate the rules one by one until all the rules containing the target relation are processed.

"$\Leftarrow$": It remains to show that, given a relational algebra expression, we can construct an equivalent range-restricted nr-datalog$^\neg$ program. We consider here only the six primitive operators: selection, projection, Cartesian product, rename, set union, and set difference. One algebra fragment, the so-called SPJR algebra, consists of the first 4 operators, namely selection, projection, Cartesian product, and rename.

In [2] (page 61) the simulation of SPJR algebra by conjunctive queries is given. Since conjunctive queries are a fragment of nr-datalog$^\neg$, we only need to consider the remaining 2 operators: set union and set difference. The simulation is trivial: for set union we construct two rules with the same rule head. The set difference operation $R - S$ corresponds to $ans(x) \leftarrow R(x), \neg S(x)$. □

The expressive power of relational algebra is equivalent to that of a fragment of the database query language SQL (essentially, SQL without grouping and aggregate functions). The expressive power of SQL is discussed in [108, 55, 107].

On ordered databases, Theorem 241 together with the Theorems 230, 232, and 233 implies

Theorem 244. *On ordered databases, the following query languages capture* P*: stratified datalog, datalog under well-founded semantics, and datalog under inflationary semantics.*

Syntactical restrictions allow us to capture classes within P. Let datalog$^+(1)$ be the fragment of datalog$^+$ where each rule has at most one non-database predicate in the body, and let datalog$^+(1, d)$ be the fragment of datalog$^+(1)$ where each predicate occurs in at most one rule head.

Theorem 245. *([80, 157]) On ordered databases, datalog*$^+(1)$ *captures* NL *and the fragment datalog*$^+(1, d)$ *captures* L*.*

Due to the inherent non-determinism, the stable model semantics is much more expressive.

Theorem 246. *([142]) Datalog under stable model semantics captures* co-NP*.*

Note that for this result an order on the input database is not needed. Informally, in each stable model such an ordering can be guessed and checked by the program.

Finally, we briefly address the expressive power of disjunctive logic programs.

Theorem 247. *([64, 65]) Disjunctive datalog under stable model semantics captures* Π_2^p*.*

9 Optimization

This section concludes the thread on evaluation and operational semantics beginning with Section 6. Where Sections 6 and 7 focus on the evaluation of entire query programs containing many rules and the interaction or chaining of such rules, this section's focus is on the evaluation of individual queries (cf. Section 3 for the definition of conjunctive and first-order queries). Where appropriate, though, we also remind of related results on query programs.

We focus on two aspects of query evaluation only, viz., query rewriting and containment and (logical) query algebras. A more complete picture of challenges and techniques for efficient query evaluation can be found, e.g., in [75], Chapters 15–16, or in [82].

Conjunctive or first-order queries are useful tools for the declarative specification of one's query intent. Figuring out the details of how such queries are evaluated is left to the query engine. For actual evaluation, however, a query engine needs to determine a detailed specification on how to evaluate a given query. Such a specification is commonly called a *query plan*. Query plans are typically expressed in an algebra, i.e., a set of operators on the domain of discourse. Query algebras thus serve to demonstrate how to specify and optimize evaluation plans for queries and are the focus of the first part of this section (Section 9.1).

Both on the level of the declarative query and on the level of a query plan for that query, we might want to find semantically equivalent queries that are better suited for execution. Furthermore, given a set of queries (e.g., resulting from the bodies of several rules) we might want to avoid doing the same or similar work for different queries several times. Rather we would like to compare these queries to find common sub-queries that can be evaluated once and then shared for the execution of multiple other queries. Such problems are considered in the second part (Section 9.3) of this section on query rewriting and containment.

9.1 An Algebraic Perspective on Queries

The relational algebra, introduced in [43, 44] and refined in [37], has long been the formal foundation for relational database systems. Though practical systems often deviate notably (see Section 9.2.2), it has proved to be an invaluable tool for understanding formal properties of query languages (primarily, completeness, complexity, and semantics), for query planning and optimization, and for implementing query engines.

In the following, we give a brief definition of (a variant of) the relation algebra and its relation to rule-based query languages. For a more detailed discussion of the relational algebra see, e.g., [75]. An outlook on extensions of the relational algebra covering two aspects (duplicates and order) of query languages mostly ignored in the rest of this article follows. We conclude this section with some remarks on algebras for complex values, where the limitation to constants as attribute values is relaxed.

Let $\mathcal{L}$ be a signature (cf. Section 3, Definition 3), $\mathcal{D}$ a database schema over $\mathcal{L}$ (cf. Section 3, and I a database instance for $\mathcal{D}$. In the following, we use the relational view of logic as described in Section 3.3. In particular, we use the unnamed or ordered perspective of relations: Attributes in a relation are identified by position (or index) not by name. For a tuple $t = (x_1, \ldots, x_n)$ we use $t[i]$ to denote the value of the i-th attribute in t, i.e., x_i. We assume a finite domain and, where convenient, the presence of a domain relation enumerating the elements of the domain.

Definition 248 (Selection). *Let P be an n-ary relation from I and C be a conditional expression $i = c$ with $i \leq n$ and c a constant from $\mathcal{L}$ or $i = j$ with $i, j \leq n$.*

The relational selection $\sigma_C(P)$ returns the set of tuples from P that fulfill C, viz. $\sigma_{i=c}(P) = \{t \in P : t[i] = c\}$ and $\sigma_{i=j}(P) = \{t \in P : t[i] = t[j]\}$.

As discussed in Section 3.3, relations can be seen as tables with each tuple forming a row and each attribute forming a column. Selection can than be seen as a "vertical" restriction of such a table, where some rows (i.e., tuples) are omitted. From a perspective of first-order queries, selection corresponds to body atoms containing some constants c if C is $i = c$ and to multiple occurrences of the same variable if C is $i = j$.

In practice, C is often extended to allow further conditional expression over elements of the domain, e.g., on ordered sorts comparisons such as $i \leq c$ are possible.

Definition 249 (Projection). *Let P be an n-ary relation from I and $i_1, \ldots, i_m \leq n$ with $k < l \implies i_k < i_l$.*

The relational projection $\pi_{i_1,\ldots,i_m}(P) = \{(t_1, \ldots, t_m) : \exists t_i \in \mathcal{D} : t[i_1] = t_1 \wedge \ldots \wedge t[i_m] = t_m\}$ returns the m-ary relation made up of tuples from P dropping all but the $i_1, \ldots, i_m$-th attributes.

Projection can be seen as "horizontal" restriction of a relation (imagined as a table). In contrast to the selection, relational projection however may incur additional cost beyond the linear "slicing" off of a few columns: Dropping columns may lead to duplicates that are not allowed in a (pure) relation.

The first-order correspondent of relational projection is the body-only occurrence of variables.

Definition 250 (Cartesian product). *Let P be an n-ary, Q an m-ary relation from I.*

The Cartesian (or cross) product $P \times Q = \{(t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}) : (t_1, \ldots, t_n) \in P \wedge (t_{n+1}, \ldots, t_{n+m}) \in Q\}$ returns the $(n + m)$-ary relation of all tuples consisting of concatenations of first tuples from P and second tuples from Q.

Cartesian product can be seen as table multiplication as the name indicates and corresponds to the conjunction of atoms (with no shared variables) in first-order queries.

The relational algebra is completed by standard set operations on relations:

Definition 251 (Relational union, intersection, and difference). *Let P and Q be n-ary relations.*

The relational union $P \cup Q = \{t \in \mathcal{D}^n : t \in P \vee t \in Q\}$, intersection $P \cap Q = \{t \in \mathcal{D}^n : t \in P \wedge t \in Q\}$, and difference $P - Q = \{t \in \mathcal{D}^n : t \in P \wedge t \notin Q$ are specialisations of standard set operations to sets of relations.

Notice that all three operations require that P and Q have the same arity. As usual either union or intersection can be defined in terms of the other two (at least under the assumption of a domain relation enumerating all elements of the domain).

Two more operators are commonly used in relational algebra expressions though they can be defined by means of the previous ones: relational join and division.

Definition 252 (Join). *Let P be an n-ary, Q be an m-ary relation from I, $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$ a partial, injective function, and $k = |f^{-1}(\{1, \ldots, m\}|$ the number of pairs in f.*

The relational join $P \bowtie_f Q = \{(t_1, \ldots, t_{n+m-k}) : (t_1, \ldots, t_n) \in P \wedge \exists t \in Q : \forall n < i \leq n + m - k : t[i - n] = t_i \wedge \forall (i, j) \in f : t_i = t[j]\}$ returns the $(n + m - k)$-ary relation of all tuples from P combined with those tuples from Q where the i-th attribute value in P is equal to the $f(i)$-th in Q (omitting the $f(i)$-th attribute from the result).

Often f is small and we write directly $P \bowtie_{i \to j, \ldots} Q$ instead of giving f separately. Relational join corresponds to multiple occurrences of a variable in a conjunctive query.

A join over k attributes can be rewritten to k selections on a Cartesian product: Let $f : \{(i_1, j_1), \ldots, (i_k, j_k)\}$ and P be an n-ary relation.

$$P \bowtie_f Q = \sigma_{i_k = j_k + n}(\sigma_{i_{k-1} = j_{k-1} + n}(\ldots \sigma_{i_1 = j_1 + n}(P \times Q) \ldots)$$

Definition 253 (Division). *Let P be an n-ary, Q be an m-ary relation from I with $m < n$.*

The relational division (or quotient)

$$P \div Q = \{(t_1, \ldots, t_{n-m}) : \forall (t_{n-m+1}, \ldots, t_n) \in Q : (t_1, \ldots, t_n) \in P\}$$

returns the $n - m$-ary relation of all tuples t such that any combination of t with a tuples from Q forms a tuple from P

The division is the relational algebra's counterpart to universal quantification in bodies of first-order queries.

Division can be rewritten to an expression using only projection, difference, and Cartesian product:

$$P \div Q = \pi_{A_1, \ldots, A_n}(P) - \pi_{A_1, \ldots, A_n}((\pi_{A_1, \ldots, A_n}(P) \times Q) - P)$$

In early formulations of the relational algebra including Codd's original proposal, additional operators such as the permutation are provided:

Definition 254 ([44, 37] Permutation). *Let P be an n-ary relation from I and $f : \{1, \ldots, n\} \to \{1, \ldots, n\}$ a bijection.*

The permutation

$$\mathsf{Perm}_f(P) = \{(t_1, \ldots, t_n) : \exists t \in P : t[f(1)] = t_1, \ldots, t[f(n)] = t_n\}$$

returns an n-ary relation containing all permutations of tuples of P.

However, permutation is usually not considered as part of the relational algebra due to its undesirable complexity (the size of $\mathsf{Perm}_f(P)$ may be exponentially higher than the size of P, viz. in $O(|P| \times n^n)$) and as it can be expressed as $\pi_{2,\ldots,n+1}(\sigma_{n=n+1}(\ldots \pi_{2,\ldots,n+1}(\sigma_{2=n+1}(\pi_{2,\ldots,n+1}(\sigma_{1=n+1}(P \times \pi_1(P))) \times \pi_2(P))) \times \ldots \pi_n(P)))$. Notice, however, that the equivalence is of another quality than the equivalences for join and division: It depends on the schema of P and its size is linear in the arity of P (compensating for the lower complexity of π, σ, and $\times$ compared to Perm).

Another common extension of the relational algebra is the semi-join operator.

Definition 255 (Semi-join). *Let P be an n-ary, Q an m-ary relation from I, $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$ a partial, injective function, and $k = |f^{-1}(\{1, \ldots, m\}|$ the number of pairs in f.*

The relational semi-join $P \ltimes_f Q = \{(t_1, \ldots, t_n) : (t_1, \ldots, t_n) \in P \wedge \exists t \in Q : \forall (i, j) \in f : t_i = t[j]\}$ returns the n-ary relation of all tuples from P for which a tuple from Q exists such that the i-th attribute value in P is equal to the $f(i)$-th in Q.

Intuitively, the semi-join corresponds to a filter on P that only retains tuples from P with join partners from Q. In contrast to the Cartesian product and the normal join its result is thus always linear in P and no trace of Q occurs in the result.

Let P be an n-ary relation, $i_1, \ldots, i_k$ be the attributes from Q occurring in f, then $P \ltimes_f Q = \pi_{1,\ldots,n}(P \bowtie_f Q) = P \bowtie_f \pi_{i_1,\ldots,i_k}(Q)$. Thus the semi-join can be expressed using only projection and join or only projection, selection, and Cartesian product (as join can be expressed using selection and Cartesian product only).

However, a rewriting of semi-joins is often not desirable. To the contrary [21] proposes to use semi-joins to reduce query processing in some cases (tree queries) even to polynomial time complexity. Recent work [105] shows that the semi-join algebra, i.e., relational algebra with the join and Cartesian product replaced by the semi-join is equivalent to the guarded fragment of first-order logic.

The semi-join operator is an example for an operator that though actually weaker than the existing operators in the relational algebra might actually be exploited to obtain equivalent, but faster formulations for a restricted class of queries.

A similar observation can be made for the usual handling of universal quantification as division (with or without subsequent rewriting) as well as existential quantification as projection may result in poor performance as intermediary results are unnecessary large. This has lead to the development of several (more

efficient, but also more involved) direct implementations of division as well as of alternatives such as the semi- and complement-join:

Definition 256 (Complement-Join). *(cf. [26]) Let P be an n-ary, Q be an m-ary relation from I, $f : \{1,\ldots,n\} \to \{1,\ldots,m\}$ a partial, injective function, and $k = |f^{-1}(\{1,\ldots,m\}|$ the number of pairs in f.*

The relational semi-join $P \overline{\ltimes}_f Q = \{(t_1,\ldots,t_n) : (t_1,\ldots,t_n) \in P \wedge \nexists t \in Q : \forall (i,j) \in f : t_i = t[j]\}$ returns the n-ary relation of all tuples from P for which no *tuple from Q exists such that the i-th attribute value in P is equal to the $f(i)$-th in Q.*

Obviously, $P \overline{\ltimes}_f Q = P - \pi_{1,\ldots,n}(P \ltimes_f Q)$ and thus $P \overline{\ltimes}_f Q = P - P \ltimes_f Q$.

For details on the use of the complement- and semi-join for realizing quantification, see [26].

9.1.1 Translating First-Order Queries. Chandra and Merlin [37] show that

Theorem 257. *([37]) For each relational expression there is an equivalent first order query, and vice versa.*

There is one caveat: This result only holds if, as Chandra and Merlin assume, the domain is finite or the queries are domain independent. In the following, the former is assumed (an exhaustive discussion of domain independence can be found, e.g., in [2]). Assuming a finite domain allows "domain closure" for queries such as $ans(x) \leftarrow \neg q(x,y)$. Such a query is transformed into the expression $\pi_1(\mathcal{D}^2 - Q)$ with $\mathcal{D}$ domain of the query and Q the relation corresponding to the predicate symbol q.

Proof (Sketch). The equivalence proof from [37] is a fairly straightforward structural induction over relational expressions and first-order queries respectively. Instead of a full proof, we give a number of illustrative translation:

1. $ans(x) \leftarrow q(x,y) \wedge r(y,z)$ is equivalent to $\pi_1(Q \ltimes_{2\to 1} R)$ as well as to $\pi_1(\sigma_{2=3}(Q \times R))$ as well as to $\pi_1(Q \ltimes_{2\to 1} \pi_1(R))$.
2. $ans(x) \leftarrow \neg q(x,y)$ is equivalent to $\pi_1(\mathcal{D}^2 - Q)$ as well as to $\mathcal{D} - \pi_1(Q)$.
3. $ans(x,y) \leftarrow (p(x,y) \wedge \neg q(y)) \vee (p(x,y) \wedge \forall z : s(x,y,z)$ is equivalent to $\pi_{1,2}((P - (\mathcal{D} \times Q)) \cup (P \ltimes_{1\to 1,2\to 2} \pi_{1,2}(S \div \mathcal{D})))$.

Theorem 258. *(cf. [37]) Conjunctive queries and relational expressions formed only from selection, projection, Cartesian product, intersection, and join have the same expressiveness. This extends trivially to expressions formed only from selection, projection, and Cartesian product, as join can be rewritten as above and, for n-ary relations P and Q, $P \cap Q = \pi_{1,\ldots,n}(\sigma_{1=n+1}(\sigma_{2=n+2}(\ldots \sigma_{n=n+n}(P \times Q)\ldots)))$.*

For illustration, see equivalence (1) in the proof of Theorem 257.

9.1.2 Query Rewriting. It is important to notice in the above examples of equivalent relational algebra expressions for a given first-order query, that there are always several reasonable expressions for the same query. Consider, e.g., again the query $ans(x, y) \leftarrow p(x, y) \wedge (\neg q(y, z) \vee \neg r(y, w))$. This query is equivalent to, e.g.,

- $\pi_{1,2}(P \ltimes_{2\to 1} ((\mathcal{D}^2 - Q) \cup (\mathcal{D}^2 - R)))$
- $\pi_{1,2}(P \ltimes_{2\to 1} (\pi_1(\mathcal{D}^2 - (Q \cap R))))$
- $\pi_{1,2}(P \ltimes_{2\to 1} (\mathcal{D} - \pi_1(Q \cap R)))$

Of these three equivalent queries only the application of de Morgan's law carries over to the first-order queries. Many other equivalences have no correspondence in first-order queries. In this respect, first-order queries are more declarative and more succinct representations of the query intent than relational expressions. However, that relational expressions allow such fine differences has shown to be of great value for query optimization, i.e., determining which of the equivalent formulations of a query should be used for evaluation: The relational algebra allows the separation of this planning phase from the actual evaluation. A query planner or optimizer can decide, based, e.g., on general equivalences as discussed here and on estimated costs of different relational expressions which variant to use.

Equivalence Laws. Equivalence laws between relational algebra expressions comprise common laws for set operators $\cup, \cap$, and $-$. In particular, $\cup$ and $\cap$ are associative and commutative, de Morgan's law and distribute laws for intersection and union apply.

More interesting are equivalence laws for the additional operators such as $\ltimes$, $\times$, σ, and π:

Cartesian Product and Join. The Cartesian product $\times$ is commutative and associative just as $\cup$ and $\cap$. For the relational $\ltimes$ a proper adjustment of the join condition is required[30]. If the join expression is flipped, $\ltimes$ is commutative, i.e., $P \ltimes_{1=2} Q \equiv Q \ltimes_{2=1} P$. For associativity, the join condition needs to be adapted, e.g., $P \ltimes_{1=3} (R \ltimes_{1=2} S) \equiv (P \times R) \ltimes_{1=1,3=2} S$ (assuming all relations are binary) since both join conditions involve attributes from S.

Selection. Selection is generally a good candidate for optimization, pushing selections (a fast but possibly fairly selective operation) inside of an expression thus limiting the size of intermediary results. Also selection can generally be propagated "down" into an expression: Selection distributes over $\times$, $\ltimes$ and the set operators. For $-$ and $\cap$ it suffices to propagate the selection to the first relation, e.g., $\sigma_C(P - Q) = \sigma_C(P) - \sigma_C(Q) = \sigma_C(P) - \sigma_C(Q)$. For Cartesian

[30] Note, that this when using the named perspective of the relational algebra (where attributes are identified by name rather than position) no adjustment is necessary and $\ltimes$ is trivially associative.

product and Join it suffices to propagate the selection to those sub-expressions that contain the attributes referenced in C. Let P be an n-ary relation. Then

$$\sigma_{i=c}(P \times Q) = \begin{cases} \sigma_{i=c}(P) \times Q & \text{if } i \leq n \\ P \times \sigma_{i=c}(Q) & \text{if } i > n \end{cases}$$

Projection. In contrast to selection, we can only propagate selection "down" in an expression to the point where the attributes dropped by the projection are last referenced in the expression. Conversely, an expression might benefit from introducing additional projections to get rid of attributes not used in the remainder of an expression as early as possible, viz. immediately after the innermost expression referencing them. Moreover, since as a result of a projection some tuples may become duplicates and thus be dropped in the set semantics considered so far, projection can in general not be distributed over any of the set operators. It can be propagated over join and Cartesian product: Let P be an n-ary, Q an m-ary relation and $i_1, \ldots, i_k \leq n$, $i_{k+1}, \ldots, i_{k+l} > n$ then

$$\pi_{i_1,\ldots,i_{k+l}}(P \times Q) = \pi_{i_1,\ldots,i_k}(P) \times \pi_{i_{k+1},\ldots,i_{k+l}}(Q).$$

9.2 "Real" Queries

"Real" relational databases and queries deviate in a few, but important points from both the relational algebra, first-order queries, and datalog. Some of these are highlighted in the remainder of this section.

We start by moving from relations as sets to relations as bags (then called multi-relations). Bag semantics is in practice often faster as it allows to ignore duplicates except when specifically noted rather than an implicit and costly elimination of duplicates at each relational operator (though, of course, only projection and union can actually create new duplicates).

9.2.1 From Sets to Bags. The need for multi-relations is evident when considering aggregation queries like *"what is the sum of the values of some column"*. In a set relational algebra the projection to the aggregation column collapses all same value tuples and thus makes this query impossible to express. The same argument can be made for any query that returns just a projection of the columns of queried relations, as the number of duplicates may be significant to the user (e.g., in *"what are the titles of all first and second level sections"*). Also, many practical systems support multi-relations to save the cost of duplicate handling. Indeed, neither QUEL [148] nor SQL [11], the now dominating "relational" query language, are (set) relational query languages, but rather possess features that are not expressible in (set) relational algebra, viz. aggregation, grouping, and duplicate elimination. The semantic of these expressions assumes that the underlying data structure is a bag (or even a sequence) rather than a set.

Therefore, in practical query languages duplicate handling must be addressed. Based on the control over duplicate creation and elimination, one can distinguish

relational query languages into *weak* and *strong* duplicate controlling languages. QUEL [148] and SQL [11] provide little control over duplicates (essentially just the `DISTINCT` operator and `GROUP-BY` clauses) and thus fall into the first class. The only means is an explicit duplicate elimination. Similarly, Prolog's operational semantics [160] also contains operations for explicit duplicate handling (e.g., `bagof` vs. `setof`).

In contrast, DAPLEX [146] is based on "iteration semantics" and gives precise control over the creation and elimination of duplicates. An example of a DAPLEX query is shown in the following listing:

```
FOR EACH Student
  SUCH THAT FOR SOME Course(Student)
    Name(Dept(Course)) = "EE" AND
    Rank(Instructor(Course)) = "ASSISTANT PROFESSOR"
  PRINT Name(Student)
```

A first formal treatment of this "iteration semantics" for relational databases is found in [49], where a *generalization of the relational algebra* to multi-relations is proposed. This extension is not trivial and raises a number of challenges for optimizations: joins are no longer idempotent, the position of projections and selections is less flexible, as $\pi_R(R \times S) \neq R$ and $\sigma_P(R) \uplus \sigma_Q(R) \neq \sigma_{P \vee Q}(R)$ due to duplicates in the first expression[31]. Though this algebra provides a useful theoretical foundation, it does little to address the concerns regarding efficient processing of "iteration semantics" expressions.

[84] shows that (nested) relational algebra on multi-relations (i.e., relations as bags) is strictly more expressive than on relations as sets. Unsurprisingly, the core difference lies in the "counting power" of the bag algebra. More precisely, the bag algebra no longer exhibits a $0-1$ law (i.e., the property that queries are either on almost all input instances true or on almost all input instances false). E.g., the query that tests whether, given two relations, the cardinality of R is bigger than the cardinality of S can be expressed as $(\pi_1(R \times R) - \pi_1(R \times S)) \neq 0$ where π is a bag algebra projection *without* duplicate elimination and $-$ is difference on multi-sets, where the multiplicity of a tuple t in the result is the multiplicity of t in the first argument minus the multiplicity in the second argument. Observe, that $\pi_1(R \times R)$ and $\pi_1(R \times S)$ contain the same tuples but if $R > S$, $\pi_1(R \times R)$ contains those tuples with greater multiplicity than $\pi_1(R \times S)$ and vice verse if $R < S$. This query observes no $0 - 1$ law: For each instance on which it is true we can construct an instance on which it is true. Unsurprisingly, it can not be expressed in set relational algebra.

Similarly, [109] proposes a query language called $\mathcal{BQL}$ over bags whose expressiveness amounts to that of a relational language with aggregates. This approach provides a formal treatment of aggregations and grouping as found, e.g., in SQL (`GROUP-BY` clause and aggregation operators such as `AVG`, `COUNT`, and

[31] Assuming π and σ to be the multi-set generalizations of their relational counterparts. $\uplus$ is understood here as *additive union*, i.e., t occurs n times in $R \cup S$, if t occurs i times in R and j times in S and $n = i + j$. [84] considers additionally maximal union (i.e., where $n = \max(i, j)$), which does not exhibit this particular anomaly.

`SUM`). $\mathcal{BQL}$ is "seen as a rational reconstruction of SQL" that is fully amenable to formal treatment. [109] also considers extensions of $\mathcal{BQL}$ with power operators, structural recursion, or loops and shows that the latter two extensions are equivalent.

[100] proposes a different view of multi-relations as *incomplete information*: though conceptually "a relation represents a set of entities", one tuple per entity, and thus does not contain duplicates, the concept of a multi-relation allows a formal treatment of partial information about entities. A multi-relation is a projection of a complete relation, i.e., it consists of a subset of columns within some relation (without duplicates). Thus it may contain duplicates in contrast to the relation. [100] considers multi-relations only as output of queries not as first class data items in the database. Semantically, they can not exist independently of the base relation. No (base or derived) relation should contain duplicates.

Moving from sets to bags (or multi-sets) in the relational algebra, obviously affects query containment and optimization. Most notably many of the algebraic laws mentioned in the previous section do no longer apply. For more details, see [38] and, more recently, [91].

9.2.2 From Bags to Sequences. Basic relational algebra (and first order logic) are fundamentally order agnostic. This is a desirable property as it allows understanding and implementation of operators without considering a specific order of the result. However, it limits the *expressiveness* of the algebra: We can not (conveniently) express a query asking for the "five students with the highest marks" nor a query asking for every second student nor a query testing if a certain set is even (e.g., the participants in a chess competition).

Moreover, queries involving order occur in many *practical* applications, in particular where reporting (presentation) and analysis is concerned. Reporting is certainly the most common use for order (return the results in some order, return only top k results, etc.). This has lead to the addition of order as an "add on" to the usual query evaluation frameworks, e.g., in SQL where the `ORDER BY` clause may not be used as part of a view definition but only at the outer-most (or reporting) level of a query.

Finally, the *physical* algebra (i.e., the actual algorithms used for evaluating a query) of most relational database systems (including all major commercial systems) is based on the concept of iterators (or streams or pipelines): A physical query plan is a hierarchy of operators, each iterating over the output of its dependent operators and creating its own result on demand. Conceptually, such operators support (aside of `open` and `close` for initialization and destruction) only one operation, viz. `next`, which returns the next element in the operator's result. Note, that such a design intrinsically supports order. However, if used for the evaluation of pure relational expressions, the order in which an operator produces its results is up to the implementation of that operator and we can choose to implement the same logical operator with physical operators that produce results in different orders. This flexibility (in choosing an efficient operator implementation regardless of the result order) is lost, if the logical operators

already require a specific order. Nevertheless, exploiting the ability of physical operators to manage order to provide that concept also to the query author is tempting. For more details on physical operators and order see the seminal survey [82].

These considerations have, more recently, revived interest in a proper treatment of order in an algebra driven by two main areas: analytical queries (OLAP-style) and queries against (intrinsically ordered) XML data and their realization in relational databases.

List-based Relational Algebra. Focused on the first aspect, [147] proposes a list-based recasting of the relational algebra.

First, we redefine a relation to be a *finite sequence* of tuples over a given relation schema (consisting as usual in a finite set of attributes and associated domains). A relation may, in particular, contain duplicates (if key attributes are present, the values of the key attributes of every two tuples *must* be distinct as usual).

Under this definition, *selection* and *projection* remain essentially unchanged except that they are now order and duplicate preserving. Only projection may introduce *new* duplicates, if the same tuple results from more than one input tuple.

Union is, obviously, affected considerably by ordered relations. [147] proposes standard list append as union, appending the tuples of the second relation in order after the tuples of the first relation.

Similar, *Cartesian product* $A \times B$ is defined as the result of appending for each tuple of A (in order) the product with each tuple of B (again in order). Thus the result contains first the combination of the first tuple of A with each tuple of B (preserving B's order), followed by the combination of the second tuple of A with each tuple of B, etc. *Join* is defined as usual as combination of selection and product. Note that this definition is immediately consistent with a nested loop implementation. However, other join and product operators (such as sort-merge or hash join) produce tuples in different orders and are no longer correct physical realisations of the logical join or product.

Difference $A \setminus B$ becomes order-preserving, multi-set difference, i.e., for each distinct tuple $t \in A$ with m occurrences in A and n occurrences in B, the last $m-n$ occurrences of t in A are preserved. We could also choose to preserve the first $m - n$ occurrences, however, preserving the *last* occurrences leads immediately to an implementation where the tuples of A must be visited only once.

As in the case of bags or multi-sets, we introduce two additional operations duplicate elimination and grouping. *Duplicate elimination* retains, as difference, the *last* occurrence. Aggregation is treated analogously to projection. For details on duplicate elimination and grouping see [75] on multi-sets and [147] on sequences.

In additional to these adapted standard or multi-set operators, we need two additional operators to properly support order: sorting and top k.

Definition 259 (Sort). *Let P be an ordered n-ary relation and $<_S$ some order relation on $\mathcal{D}^n$. Let $<_P$ denote the (total) order of tuples in P.*

The $<_s$-sorted relation $\mathsf{sort}_S(P) = [t_1, \ldots, t_n]$ such that for all $t_i : \forall t_j : i < j \implies t_i <_s t_j \vee \neg(t_i <_s t_j) \wedge t_i <_p t_j$ returns the tuples in P ordered by $<_s$. If $<_s$ is not a total order, the order of P is preserved on any "gaps" in $<_s$.

In practical cases, S might, e.g., consist of a list of attributes and order specifications of the form "`ASCENDING`" or "`DESCENDING`" as in the case of SQL's `ORDER BY`.

Definition 260 (Top k). *Let $P = [t_1, \ldots t_n]$ be an ordered n-ary relation and $k \in \mathbb{N}$ some positive integer.*

The top-k relation $\mathsf{top}_k(P) = [s_1, \ldots, s_l]$ with $l = \min(k, n)$ such that $s_i \in \mathsf{top}_k(P) \implies P = [t_1, \ldots, t_{i-1}, s_i, t_{i+1}, \ldots t_n]$ returns the k first entries of P (or all entries of P if P has less than k entries).

The top k operation is available, e.g., in Microsoft's SQL Server (`TOP` N clause), in IBM's DB2 (`FETCH FIRST` N `ROWS ONLY`), and in Oracle DBMS (using a selection on the pseudo-column `ROWNUM`).

With this sequence algebra, we can now express *all* SQL constructs including `ORDER BY`, `GROUP BY`, and `DISTINCT` which were not expressible in the standard relational set algebra. To also cover null values some additional modifications (mostly to selection) are needed, see, e.g., [159]. For more details on translating SQL into relational algebra (and/or relational calculus) see [34].

However, this comes at the price of considerably more involved equivalences (cf. [147]). Many of the associativity and commutativity laws for the relational (set) algebra no longer hold in the bag or set case. Therefore, practical optimizers often go to considerable length to find order-agnostic parts of queries even if some ordering is required, e.g., at the end result. For those parts, we can then use standard relational algebra optimization. This desire is reflected in the limitations of the `ORDER BY` clause in SQL and, to give just one more recent example, the introduction of `unordered` contexts in XQuery, where the usual strictly ordered semantics of XQuery is (temporarily) "disabled". Handling order only where necessary, is an important aim of XML and XQuery optimization and algebras, see, e.g., [128] for details.

9.2.3 From Constants to Complex Values.

So far, we have considered the values contained in an attribute of a relation as atomic or simple, i.e., without further *structure*. Examples of such values are, of course, numbers, strings, truth values, and enumerated values.

In many applications, we are, however, concerned also with values of another nature: structured or hierarchical values such as sets, lists, or trees. Though such data may be decomposed and stored in first-normal form, such an approach may not be desirable if the values are very irregular, often considered as a single "value", or their shape is dictated by application or other external requirements [113].

Research on complex values has been a focus of the database community in the first half of the 1990s, see, e.g., [84, 25, 103, 32, 1]. They have seen renewed interest in the context of XML and other semi-structured data and its querying and storing (both natively and in relational databases). They serve, e.g., as an expressive foundation for a large fragment of XQuery [101].

Complex values: Tuples and Sets. There are several variant definitions in the literature for complex values. In this section, we follow mostly [1] as its notions are conveniently simple and yet expressive enough to discuss important variations.

First, we need to define a complex value:

Definition 261 (Complex value). *Let $\mathcal{D}_1, \ldots, \mathcal{D}_n$ be domains for* atomic *values. Then*

- *each $a \in \mathcal{D}_i$ for $i \leq n$ is a (atomic) value of type D_i.*
- *each $[v_1, \ldots, v_n]$ is a (tuple) value of type $[T_1, \ldots, T_n]$ if $v_1, \ldots, v_n$ are values of types $T_1, \ldots, T_n$, respectively, and $n \in \mathbb{N}$.*
- *each finite $S \in \mathcal{P}(\mathsf{values}(T))$ is a (set) value of type $\{T\}$ if T is a type and $\mathsf{values}(T)$ is the set of all values of T.*

A complex value is either an atomic, a tuple, or a set value.

Note that the definition allows *only finite* values (both tuples and sets are required to be finite and there are *no cycles*. Furthermore, values do not carry identity (as in the case of object- or object-relational databases), i.e., there are no two distinct values with the same structure.

Example 262. Examples of complex values (for simplicity using strings and integers as only domains):

- "Caesar", 17, "Flamen Dialis", 44 are all atomic values of type string or integer, respectively.
- ["Caesar", 44], [44], [], ["Caesar", [17, "Flamen Dialis"]] are all tuple types.
- {["Caesar", 44], [44], {[]}, {}} is a set type.

Obviously there are infinitely more complex values (actually, even if the domains for atomic values are empty that is the case).

Often, for complex values a named perspective on tuples is often preferred. It has the additional advantage of highlighting the close relationship to XML and other semi-structured data models. In a named perspective, we would, e.g., obtain [Name: "Caesar", Office: [AtAge: 17, Title: "Flamen Dialis"]]. However, for consistency with the rest of this article we use the unnamed perspective in the following.

A well-known variant of this definition of complex values is the nested relations model, see, e.g., [92]. The idea is to allow entire relations to occur as values of attributes of other relations. In the terms of our above definition this means that sets and tuples constructors must alternate in any complex value: A set may contain only tuples (but not sets) and a tuple may contain only sets or atomic values (but not tuples). However, this limitation does not affect the expressiveness of the data model and is not considered further.

An algebra for complex values. For an algebra, the main difference if we consider not only atomic but also complex values are operators that allow the construction and destruction of set and tuple values as well as there restructuring.

The first addition to the algebra may be surprising: It is essentially a higher-order function to apply some restructuring operation to all elements of a set:

Definition 263 (Map (Replace)). *Let R be a set type and f a restructuring function. Then $\mathsf{map}\langle f\rangle(R) = \{f(r) : r \in R\}$ is the set of elements of R restructured according to f, i.e., the application of f to all elements of R. This can be considered a higher-order function familiar from functional programming.*

Restructuring functions are, e.g., projections, set and tuples construction, and their compositions. For details see [1] and Example 264.

Note that, in particular, $\mathsf{map}\langle f\rangle$ itself is a restructuring function allowing, which allows for restructuring of nested elements. However, restructuring is limited to a depth fixed in the query (no recursive tree transformations).

In some ways, $\mathsf{map}\langle\rangle$ can be considered a generalization of relational project, which restructures each tuple in a relation by retaining only some of its attributes.

In a similar generalization of the relational select, *select* on complex values allows arbitrary boolean functions as predicates including comparators $=$, $\in$, and $\subset$.

The basic set operations ($\cup$, $\cap$, and, $\setminus$) are defined as usual. Both arguments must be sets of the same type. The *cross product* is changed to an n-ary operation on set types $\times(R_1, \ldots, R_n) = \{[t_1, \ldots, t_n] : t_1 \in R_1, \ldots, t_n \in R_n\}$ such that the result tuples obtain each component from one of the parameter sets.

Note, that in particular binary cross product no longer combines the components of the tuples of the two parameter sets (relations) but rather creates simply a set of binary tuples containing as components the unchanged elements of the original sets. To obtain classical product, a compound expression is needed: Let R, S be two binary relations, then $\mathsf{map}\langle[1.1, 1.2, 2.1, 2.2]\rangle(\times(R, S))$ computes the classical product of R and S: First the compute a set of binary tuples with the first component a tuple from R and the second a tuple from S ($\times(R, S)$, then we transform that result using the restructuring function $[1.1, 1.2, 2.1, 2.2]$, a shorthand for the function that creates from each binary tuple in the parameter set a new tuple with the first components first attribute (1.1) followed by the first components second attribute (1.2) etc.

Finally, we add a "flattening" or "set collapse" operation $\mathsf{collapse}$ that takes a set of sets as argument and returns the union of all members in that set.

Example 264. Let S of type $\{[Int, \{Int\}]\}$, i.e., S is a nested relation with the second attribute containing sets of integers as values.

- Select from S tuples where the first component is a member of the second component: $\sigma\langle 1 \in 2\rangle(S)$.
- Select pairs sets of (set) values in S where the first is a subset of the second: $\mathsf{map}\langle[1.2, 2.2]\rangle(\sigma\langle 1.2 \subseteq 2.2\rangle(\times(S, S)))$ which is equivalent to the expression

$\sigma\langle 1 \subseteq 2\rangle(\times(\mathsf{map}\langle[1.2]\rangle(S), \mathsf{map}\langle[1.2]\rangle(S)))$ where the restructuring (in this case a projection) is pushed to the leaves of the expression.
- The Join of R and S on the first attribute again is expressed as a compound expression $\sigma\langle 1 = 3\rangle(\mathsf{map}\langle[1.1, 1.2, 2.1, 2.2]\rangle(\times(R, S)))$ in analogy to the standard equivalence $R \bowtie_C S = \sigma_C(R \times S)$.
- Unnesting (as introduced in [92]) creates from a relation like S a set of flat tuples containing combinations of the first attribute of the original tuples and one of the elements of the second attribute. It can be expressed as $\mathsf{collapse}(\mathsf{map}\langle\times(\{1\}, 2)\rangle(S))$: First the restructuring function $\times(\{1\}, 2)$ is applied to all elements of S creating sets of tuples with the first attribute from the values of the first attribute of S and the second attribute from the members of the set values of the second attribute of S. Then all these sets are collapsed into a single one.
- To add the value of the first attribute of S to the second, we can use $\mathsf{map}\langle\{1\} \cup 2\rangle(S)$.

Note, that all queries are also functions, either boolean or restructuring functions and can thus be used as parameters for $\mathsf{map}\langle\rangle$ or $\sigma\langle\rangle$, respectively.

[1] also establishes a calculus that is, for safe queries, equivalent to the above algebra. In particular, it is shown that queries in this algebra are domain-independent in the sense of Section 3.2.3).

A possible extension to the above algebra to allow in particular the expression of recursive queries such as the transitive closure of a relation directly in the algebra, is the *powerset* operator: Given a set as parameter, powerset returns the set of all subsets of R.

The idea of expressing transitive closure in the above algebra is to first construct the powerset of a relation and then eliminate all sets that do not (1) contain the original relation and (2) are not transitively closed. For details on the complex expression for computing transitive closure, see [1].

The price of the powerset operator is further examined in [149]. They show that any algorithm in the above algebra for expressing transitive closure using powerset (which is the required for expressing transitive closure, cf. [129]) needs *exponential space*. In fact, they even prove that result for deterministic transitive closure, i.e., transitive closure of a graph with out-degree ≤ 1. This contrasts with PSPACE algorithms for transitive closure using, e.g., relational algebra and a fixpoint operator such as WHILE. It is an open problem, whether there are queries expressible with powerset that are not expressible without

[25] shows that the above algebra (and equivalent or similar proposed query languages such as the nested relational algebra [92]) without powerset operator can be elegantly formalized as monad algebra, studied in the context of category theory and programming languages, and based, unsurprisingly, on structural recursion on sets.

Combining complex values with bags (as discussed for flat relational algebra in Section 9.2.1) is discussed in [83]. As expected, the nested bag algebra is more expressive than the nested set algebra (as introduced above), however, with increasing nesting depth the expressiveness difference becomes fairly subtle.

Combining complex values with sequences (as discussed for flat relational algebra in Section 9.2.2) leads sequences of complex values very similar to the data model of the W3C XQuery language. [101] shows that a core of that language can indeed be formalized in notions similar to monad algebra (which, as mentioned above, is equivalent to the algebra discussed in this section without powerset operator) and gives complexity of various sublanguages of (non-recursive) XQuery.

The rest of this section considers adding complex values to relational algebra. How about adding them to, e.g., non-recursive datalog? [48] shows that non-recursive datalog (or Prolog) with trees and lists is equivalent to first-order logic over lists and trees and that, in the non-recursive case, this addition is benign w.r.t. data complexity (it remains in AC_0).

9.2.4 From Values to Objects. Managing structured or semi-structured data involves the determination of what defines the identity of a data item (be it a node in a tree, graph, or network, an object, a relational tuple, a term, or an XML element). Identity of data items is relevant for a variety of concepts in data management, most notably for joining, grouping and aggregation, as well as for the representation of cyclic structures.

"What constitutes the identity of a data item or entity?" is a question that has been answered, both in philosophy and in mathematics and computer science, essentially in two ways: based on the extension (or structure and value) of the entity or separate from it (and then represented through a surrogate).

Extensional Identity. Extensional identity defines identity based on the extension (or structure and value) of an entity. Variants of extensional identity are Leibniz's law[32] of the *identity of indiscernibles*, i.e., the principle that if two entities have the same properties and thus are indiscernible they must be one and the same. Another example of this view of identity is the *axiom of extensionality* in Zermelo-Fraenkel or von Neumann-Bernays-Gödel set theory stating that a set is uniquely defined by its members.

Extensional identity has a number of desirable properties, most notably the compositional nature of identity, i.e., the identity of an entity is defined based on the identity of its components. However, it is insufficient to reason about identity of entities in the face of changes, as first pointed out by Heracleitus around 500 BC: *You cannot step twice into the same river; for fresh waters are flowing in upon you.* (Fragment 12).

He postulates that the composition or extension of an object defines its identity and that the composition of any object changes in time. Thus, nothing retains its identity for any time at all, there are no persistent objects.

This problem has been addressed both in philosophy and in mathematics and computer science by separating the extension of an object from its identity.

Surrogate Identity. Surrogate identity defines the identity of an entity independent from its value as an external surrogate. In computer science surrogate

[32] So named and extensively studied by Willard V. Quine.

identity is more often referred to as *object identity*. The use of identity separate from value has three implications (cf. [12] and [99])

- In a model with surrogate identity, naturally two notions of object *equivalence* exist: two entities can be identical (they are the same entity) or they can be equal (they have the same value).
- If identity is separate from value, identity is no longer necessarily compositional and it is possible that two distinct entities *share* the same (meaning identical, not just same value) properties or sub-entities.
- *Updates* or changes on the value of an entity are possible without changing its identity, thus allowing the tracking of changes over time.

In [12] value, structure, and location independence are identified as essential attributes of surrogate identity in data management. An identifier or identity surrogate is value and structure independent if the identity is not affected by changes of the value or internal structure of an entity. It is location independent if the identity is not affected by movement of objects among physical locations or address spaces.

Object identity in object-oriented data bases following the ODMG data model fulfill all three requirements. Identity management through primary keys as in relational databases violates value independence (leading to Codd's extension to the relation model [45] with separate surrogates for identity). Since object-oriented programming languages are usually not concerned with persistent data, their object identifiers often violate the location independence leading to anomalies if objects are moved (e.g., in Java's RMI approach).

Surrogate or object identity poses, among others, two challenges for query and programming languages based on a data model supporting this form of identity: First, where for extensional identity a single form of equality (viz. the value and structure of an entity) suffices, object identity induces at least two, often three flavors of *equality* (and thus three different *joins*): Two entities may be equal w.r.t. identity (i.e., their identity surrogates are equivalent) or value. If entities are complex, i.e., can be composed from other objects, one can further distinguish between "shallow" and "deep" value equality: Two entities are "shallow" equivalent if their value is equal and their components are the same objects (i.e., equal w.r.t. identity) and "deep" equivalent if their value is equal and the values of their components are equal. Evidently, "shallow" value-based equality can be defined on top of identity-based and "deep" value-based equality.

The same distinction also occurs when *constructing* new entities based on entities selected in a query: A selected entity may be linked as a component of a constructed entity (object sharing) or a "deep" or "shallow" copy may be used as component.

Summarizing, surrogate or object identity is the richer notation than extensional identity addressing in particular object sharing and updates, but conversely also requires a slightly more complex set of operators in query language and processor.

The need for surrogate identity in contrast to extensional identity as in early proposals for relational databases has been argued for [45], as early as 1979 by

Codd himself. He acknowledges the need for unique and permanent identifiers for database entities and argues that user-defined, user-controlled primary keys as in the original relational model are not sufficient. Rather permanent *surrogates* are suggested to avoid anomalies resulting from user-defined primary keys with external semantics that is subject to change.

In [99] an extensive review of the implications of object identity in data management is presented. The need for object identity arises if it is desired to "distinguish objects from one another regardless of their content, location, or addressability" [99]. This desire might stem from the need for dynamic objects, i.e., objects whose properties change over time without loosing their identity, or versioning as well as from object sharing.

[99] argues that identity should neither be based on address (-ability) as in imperative programming languages (variables) nor on data values (in the form of identifier keys) as in relational databases, but rather a separate concept maintained and guaranteed by the database management system.

Following [99], programming and query languages can be classified in two dimensions by their support for object identity: the first dimension represents to what degree the identity is managed by the system vs. the user, the second dimension represents to what degree identity is preserved over time and changes.

Problems of user defined identity keys as used in relational databases lie in the fact that they cannot be allowed to change, although they are user-defined descriptive data. This is especially a problem if the identifier carries some external semantics, such as social security numbers, ISBNs, etc. The second problem is that identifiers can not provide identity for some subsets of attributes.

The value of object identities (OIDs) as query language primitives is investigated in [3]. It is shown that OIDs are useful for

- object sharing and cycles in data,
- set operations,
- expressing any computable database query.

The data model proposed in [3] generalizes the relational data model, most complex-object data models, and the logical data model [103]. At the core of this data model stands a mapping from OIDs to so-called *o*-values, i.e., either constants or complex values containing constants or further OIDs. Repeated applications of the OID-mapping yield *pure values* that are regular infinite trees. Thus trees with OIDs can be considered finite representations of infinite structures.

The OID-mapping function is partial, i.e., there may be OIDs with no mapping for representing incomplete information.

It is shown that "a primitive for OID invention must be in the language ... if unbounded structures are to be constructed" [3]. Unbounded structures include arbitrary sets, bags, and graph structures.

Lorel [4] represents a semi-structured query language that supports both extensional and object identity. Objects may be shared, but not all "data items" (e.g., paths and sets) are objects, and thus not all have identity. In Lorel construc-

tion defaults to object sharing and grouping defaults to duplicate elimination based on OIDs.

9.3 Optimal Views: Equivalence and Containment

9.3.1 Beyond Relational Containment. For Web queries against semi-structured data early research on query containment and optimization has focused on regular path expressions [6], short RPEs. More recently, XPath has been in the focus of research with its central role in upcoming Web query standards becoming apparent.

(Regular) path queries (or expressions, short RPEs) are regular expressions over the label alphabet of some tree or graph. They select all nodes in a tree or graph that are reachable from the root via a path (with nodes) whose labels form a word in the language given by the regular (path) expression. *Path (inclusion) constraints* (e.g., $p_1 \subset p_2$ or $p_1 \equiv p_2$ indicating that the nodes selected by p_1 are a subset, resp. identical to the nodes selected by p_2) can be exploited to rewrite regular path expressions. [6] shows that equivalence of RPEs under path inclusion constraints is, in fact, decidable in EXPSPACE. [71] extends this result to conjunctive queries with regular expressions (a subset of STRUQL) and shows that it is still decidable and in EXPSPACE.

[68] gives a practical algorithm for rewriting RPEs containing wildcards and a closure axis like XPath's `descendant`. They employ, as we do in this work, graph schemata and automata for processing such schemata. However, as the queries they consider are only regular path expressions, they can also use an automaton for (each of) the regular path expressions to be rewritten. The product of the query with the schema automaton is computed and the resulting product automaton is "pruned" to obtain a (query) automaton equivalent to the original one under the given graph schema. In effect, this allows the rewriting of regular path expressions such as `*.a.*.b.c`. If the schema specifies that there is at most one `a` and at most one `b` on the path from the root to a `c`, this can be specialized to `(not(a))*.a.((not(b))*.b.c`, an expression that prunes some search paths in the data graph earlier than the original one.

[31] shows that a restriction to a deterministic semi-structured data model where the labels of all children of a node are distinct, for paths without closure axes (i.e., with only child axis), to decidable containment and minimization, but remains undecidable for general regular expressions.

XPath containment and minimization differs from containment and minimization of RPEs in that basic XPath expressions are simpler than RPEs, (no `(a.b)*`) but full XPath contains additional constructs such as node identity join that easily make containment and minimization undecidable. Other additions of XPath such as reverse axes have been shown [124] to be reducible to a core XPath involving only forward axes and can thus be safely ignored in the following. Therefore, various subsets have been considered, for a more complete survey of the state-of-the-art in XPath query containment in ab- or presence of schema constraints see [143].

The essential results are positive results if only tree pattern queries (understood as XPath queries with only child and descendant axis and no wildcard labels) are considered (e.g., [7] presenting an $O(N^4)$, with n the number of nodes in the query, algorithm for minimizing in the absence of integrity constraints; [136] proposed an $O(n^2)$ algorithm for the same problem and an $O(n^4)$ algorithm in the presence of required-child, required-descendant, and subtype integrity constraint). In the latter paper, an $O(n^2)$ algorithm is given for the case that only required-child and required-descendant constraints are allowed. Miklau and Suciu [117] show that the problem becomes CO-NP complete if the tree patterns may also contain wildcards. [35] shows how to obtain wildcard free XPath expressions but needs to introduce new language constructs ("layer" axes) for restricted rather than arbitrary-length path traversals in the document tree (thus not contradicting the results from [117]). [143] refines this result showing that adding disjunction does not affect this complexity.

If arbitrary regular path expressions are allowed (for vertical navigation), query minimization becomes PSPACE-hard, as subsumption of regular expressions r_1 and r_2 is equivalent to testing whether $L(r_1) \subseteq L(r_2)$ which is known to be PSPACE-hard.

Whereas early work on minimization under DTDs, e.g., [161] focused on simple structural constraints (basically only child and parent constraints) and weak subsets of XPath for optimizing XML queries, Deutsch and Tannen [53] show that the problem of XPath containment in the presence of DTDs and simple integrity constraints (such as key or foreign key constraints from XML Schema) is undecidable in general and that this result holds for a large class of integrity constraints. If only DTDs are considered [143] shows that containment of XPath with only horizontal axes but including wildcards, union, filters, and descendant axes is EXPTIME-complete. [143] further shows that even for XPath with only child axis and filters containment in presence of DTDs is already CONP-complete. Adding node-set inequality to the mix makes the containment problem immediately undecidable, using XPath expressions with variables even PSPACE-complete (in the absence of DTDs)

[70] discusses also the minimization of XPath queries with only child and descendant axes (as well as filters and wildcards), focusing in particular on the effect of the wildcard operator. It proposes a polynomial algorithm for computing minimal XPath queries of limited branched XPath expressions, i.e., XPath expressions where filters (XPath predicates `[]`) occur only in one of the branches under each XPath step.

Recently, attention has turned to optimization of full *XQuery* as well: In [39], where a heuristic optimization technique for XQuery is proposed: Based on the PAT algebra, a number of normalizations, simplification, reordering, and access path equivalences are specified and a deterministic algorithm developed. Though the algorithm does not necessarily return an optimal query plan it is expected and experimentally verified to return a reasonably good one.

References

1. S. Abiteboul and C. Beeri. The Power of Languages for the Manipulation of Complex Values. *VLDB Journal*, 4(4):727–794, 1995.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Co., 1995.
3. S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. *Journal of the Association for Computing Machinery*, 45(5):798–842, 1998.
4. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wienerm. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
5. S. Abiteboul and V. Vianu. Datalog Extensions for Database Queries and Updates. *Journal of Computer and System Sciences*, 43:62–124, 1991.
6. S. Abiteboul and V. Vianu. Regular Path Queries with Constraints. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 122–133. ACM Press, 1997.
7. S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *Proc. ACM SIGMOD Symposium on the Management of Data (SIGMOD)*, pages 497–508. ACM Press, 2001.
8. Andréka and Németi. The Generalized Completeness of Horn Predicate Logic as a Programming Language. *Acta Cybernetica*, 4:3–10, 1978. (This is the published version of a 1975 report entitled "General Completeness of PROLOG".).
9. K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.
10. K. Apt and K. Doets. A New Definition of SLDNF-Resolution. *Journal of Logic Programming*, 18:177–190, 1994.
11. M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.
12. M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The Object-oriented Database System Manifesto. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-oriented Database System: The Story of O2*, chapter 1, pages 1–20. Morgan Kaufmann Publishers Inc., 1992.
13. F. Baader and W. Snyder. Unification Theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, pages 447–533. Elsevier Science, 2001.
14. J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In J. Małuszyński and N. Eisinger, editors, *Reasoning Web, First International Summer School 2005*, volume 3564 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
15. I. Balbin, K. Meenakshi, and K. Ramamohanarao. A Query Independent Method for Magic Set Computation on Stratified Databases. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 711–718, 1988.
16. I. Balbin, G. Port, K. Ramamohanarao, and K. Meenakshi. Efficient Bottom-Up Computation of Queries of Stratified Databases. *Journal of Logic Programming*, 11:295–344, 1991.

17. J. Balcázar, A. Lozano, and J. Torán. The Complexity of Algorithmic Problems on Succinct Instances. In R. Baeta-Yates and U. Manber, editors, *Computer Science*, pages 351–377. Plenum Press, New York, NY, USA, 1992.
18. C. Beeri and R. Ramakrishnan. On the Power of Magic. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 269–283. ACM Press, 1987.
19. A. Behrend. Soft Stratification for Magic set based Query Evaluation in Deductive Databases. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 102–110. ACM Press, 2003.
20. D. Benanav, D. Kapur, and P. Narendran. Complexity of Matching Problems. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 417–429. Springer-Verlag, 1985.
21. P. A. Bernstein and D.-M. W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the Association for Computing Machinery*, 28(1):25–40, 1981.
22. N. Bidoit and C. Froidevaux. Negation by Default and Unstratifiable Programs. *Theoretical Computer Science*, 78:85–112, 1991.
23. V. Bönström, A. Hinze, and H. Schweppe. Storing RDF as a Graph. In *Proc. Latin American Web Congress*, pages 27–36, 2003.
24. S. Brass and J. Dix. Disjunctive Semantics Based upon Partial and Bottom-Up Evaluation. In L. Sterling, editor, *International Conference on Logic Programming*, pages 199–213. MIT Press, June 1995.
25. V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *Proc. International Conference on Database Theory (ICDT)*, pages 140–154, 1992.
26. F. Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proc. ACM SIGMOD Symposium on the Management of Data (SIGMOD)*, pages 193–204. ACM Press, 1989.
27. F. Bry. Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled. *Data and Knowledge Engineering*, 5(4):289–312, 1990.
28. F. Bry, P.-L. Pătrânjan, and S. Schaffert. Xcerpt and XChange: Logic Programming Languages for Querying and Evolution on the Web. In *International Conference on Logic Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
29. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *International Conference on Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
30. F. Bry, S. Schaffert, and A. Schröder. A Contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Applications of Declarative Programming and Knowledge Management, Proceedings of 15th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2004, and 18th Workshop on Logic Programming, Potsdam, Germany (14th–16th March 2004)*, volume 3392, pages 258–268. GLP, GI, 2004.
31. P. Buneman, W. Fan, and S. Weinstein. Query Optimization for Semistructured Data Using Path Constraints in a Deterministic Data Model. In *Proc. International Workshop on Database Programming Languages (DBLP)*, pages 208–223. Springer-Verlag, 2000.
32. P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995.

33. L. Cavedon and J. Lloyd. A Completeness Theorem for SLDNF-Resolution. *Journal of Logic Programming*, 7:177–191, 1989.
34. S. Ceri and G. Gottlob. Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985.
35. C.-Y. Chan, W. Fan, and Y. Zeng. Taming XPath Queries by Minimizing Wildcard Steps. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2004.
36. A. Chandra and D. Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.
37. A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM Symposium on Theory of Computing (STOC)*, pages 77–90. ACM Press, 1977.
38. S. Chaudhuri. Optimization of Real Conjunctive Queries. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 59–70. ACM Press, 1993.
39. D. Che, K. Aberer, and T. Özsu. Query Optimization in XML Structured-document Databases. *The VLDB Journal*, 15(3):263–289, 2006.
40. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the Association for Computing Machinery*, 43(1):20–74, 1996.
41. Y. Chen. A Bottom-up Query Evaluation Method for Stratified Databases. In *Proc. International Conference on Data Engineering (ICDE)*, pages 568–575. IEEE Computer Society, 1993.
42. K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Base*, pages 293–322. Plenum Press, New York, NY, USA, 1978.
43. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.
44. E. F. Codd. Relational Completeness of Data Base Sublanguages. *Database Systems*, pages 65–98, 1972.
45. E. F. Codd. Extending the Database Relational Model to Capture more Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.
46. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. In *Proc. IEEE Conference on Computational Complexity*, pages 82–101, 1997.
47. E. Dantsin and A. Voronkov. Complexity of Query Answering in Logic Databases with Complex Values. In S. I. Adian and A. Nerode, editors, *Proc. International Symposium on Logical Foundations of Computer Science (LFCS)*, volume 1234 of *Lecture Notes in Computer Science*, pages 56–66. Springer-Verlag, 1997.
48. E. Dantsin and A. Voronkov. Expressive Power and Data Complexity of Nonrecursive Query Languages for Lists and Trees. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 157–165. ACM Press, 2000.
49. U. Dayal, N. Goodman, and R. H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 117–123. ACM Press, 1982.
50. J. de Bruijn, E. Franconi, and S. Tessaris. Logical Reconstruction of RDF and Ontology Languages. In *Proc. Workshop on Principles and Practice of Semantic Web Reasoning*, volume 3703 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

51. S. Debray and R. Ramakrishnan. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.
52. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–309. Elsevier Science, 1990.
53. A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *Proc. Int'l. Workshop on Knowledge Representation meets Databases (KRDB)*, 2001.
54. S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. Symposium on Logic Programming (SLP)*, pages 264–272, 1987.
55. G. Dong, L. Libkin, and L. Wong. Local Properties of Query Languages. In *Proc. International Conference on Database Theory (ICDT)*, volume 1186 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 1997.
56. R. B. Doorenbos. *Production Matching for Large Learning Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.
57. W. F. Dowling and J. H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.
58. W. Drabent. Completeness of SLDNF-Resolution for Non-Floundering Queries. In *Proc. International Symposium on Logic Programming*, page 643, 1993.
59. C. Dwork, P. Kanellakis, and J. Mitchell. On the Sequential Nature of Unification. *Journal of Logic Programming*, 1:35–50, 1984.
60. C. Dwork, P. Kanellakis, and L. Stockmeyer. Parallel Algorithms for Term Matching. *SIAM Journal of Computing*, 17(4):711–731, 1988.
61. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Perspectives in Mathematical Logic. Springer-Verlag, 1995.
62. T. Eiter and G. Gottlob. Propositional Circumscription and Extended Closed World Reasoning are $\mathbf{\Pi}_2^P$-complete. *Theoretical Computer Science*, 114(2):231–245, 1993. Addendum 118:315.
63. T. Eiter and G. Gottlob. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 15(3/4):289–323, 1995.
64. T. Eiter, G. Gottlob, and H. Mannila. Adding Disjunction to Datalog. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 267–278, 1994.
65. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, Sept. 1997.
66. R. Fagin. Finite-Model Theory — a Personal Perspective. *Theoretical Computer Science*, 116:3–31, 1993.
67. C. Fan and S. W. Dietrich. Extension Table Built-ins for Prolog. *Software — Practice and Experience*, 22(7):573–597, 1992.
68. M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proc. International Conference on Data Engineering (ICDE)*, pages 14–23. IEEE Computer Society, 1998.
69. M. Fitting. Fixpoint Semantics For Logic Programming – A Survey. *Theoretical Computer Science*, 278:25–51, 2002.
70. S. Flesca, F. Furfaro, and E. Masciari. On the Minimization of XPath Queries. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 153–164, 2003.

71. D. Florescu, A. Levy, and D. Suciu. Query Containment for Conjunctive Queries with Regular Expressions. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 139–148. ACM Press, 1998.
72. C. L. Forgy. Rete: a Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Expert systems: a software methodology for modern applications*, pages 324–341, 1990.
73. T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. In P. Barahona, F. Bry, E. Franconi, N. Henze, and U. Sattler, editors, *Reasoning Web, Second International Summer School 2006*, volume 4126 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
74. J. P. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 88–98. ACM Press, 1993.
75. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.
76. M. Garey and D. Johnson. *Computers and Intractability*. Freeman, 1979.
77. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. International Conference and Symposium on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
78. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
79. G. Gottlob and C. Papadimitriou. On the Complexity of Single-rule Datalog Queries. *Information and Computation*, 183:104–122, 2003.
80. E. Grädel. Capturing Complexity Classes with Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.
81. E. Grädel and M. Otto. On Logics with Two Variables. *Theoretical Computer Science*, 224(1-2):73–113, 1999.
82. G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–169, 1993.
83. S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 49–58. ACM Press, 1993.
84. S. Grumbach and V. Vianu. Tractable Query Languages for Complex Object Databases. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 315–327, New York, NY, USA, 1991. ACM Press.
85. Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, chapter 1, pages 1–57. Computer Science Press, 1988.
86. Y. Gurevich and S. Shelah. Fixpoint Extensions of First-Order Logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.
87. P. Hayes. RDF Model Theory. Recommendation, W3C, 2004.
88. T. Hinrichs and M. Genesereth. Herbrand Logic. Technical Report LG-2006-02, Stanford Logic Group, Computer Science Department, Stanford University, November 2006.
89. N. Immerman. Relational Queries Computable in Polynomial Time. *Information and Control*, 68:86–104, 1986.
90. N. Immerman. Languages that Capture Complexity Classes. *SIAM Journal of Computing*, 16:760–778, 1987.

91. Y. E. Ioannidis and R. Ramakrishnan. Containment of Conjunctive Queries: Beyond Relations as Sets. *ACM Transactions on Database Systems*, 20(3):288–324, 1995.
92. G. Jaeschke and H. J. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 124–138. ACM Press, 1982.
93. J. Jaffar, J.-L. Lassez, and J. Lloyd. Completeness of the Negation as Failure Rule. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 500–506, 1983.
94. D. S. Johnson. A Catalog of Complexity Classes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 2. Elsevier Science, 1990.
95. N. Jones and W. Laaser. Complete Problems in Deterministic Polynomial Time. *Theoretical Computer Science*, 3:105–117, 1977.
96. D. Kapur and P. Narendran. Complexity of Unification Problems with Associative-commutative Operators. *Journal of Automated Reasoning*, 9(2):261–288, 1992.
97. J.-M. Kerisit. A Relational Approach to Logic Programming: the Extended Alexander Method. *Theoretical Computer Science*, 69:55–68, 1989.
98. J.-M. Kerisit and J.-M. Pugin. Efficient Query Answering on Stratified Databases. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 719–726, 1988.
99. S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proc. International Conference on Object-oriented Programming Systems, Languages and Applications*, pages 406–416. ACM Press, 1986.
100. A. Klausner and N. Goodman. Multirelations — Semantics and Languages. In *Proc. International Conference on Very Large Data Bases (VLDB)*, volume 11, pages 251–258. Morgan Kaufmann, 1985.
101. C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. *ACM Transactions on Database Systems*, 31(4), 2006.
102. P. Kolaitis and C. Papadimitriou. Why Not Negation by Fixpoint? *Journal of Computer and System Sciences*, 43:125–144, 1991.
103. G. M. Kuper and M. Y. Vardi. The Logical Data Model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.
104. J.-M. Le Bars. Counterexamples of the 0-1 Law for Fragments of Existential Second-order Logic: an Overview. *Bulletin of Symbolic Logic*, 6(1):67–82, 2000.
105. D. Leinders, M. Marx, J. Tyszkiewicz, and J. V. den Bussche. The Semijoin Algebra and the Guarded Fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005.
106. D. Leivant. Descriptive Characterizations of Computational Complexity. *Journal of Computer and System Sciences*, 39:51–83, 1989.
107. L. Libkin. On the Forms of Locality over Finite Models. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 204–215, 1997.
108. L. Libkin and L. Wong. New Techniques for Studying Set Languages, Bag Languages and Aggregate Functions. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 155–166, 1994.
109. L. Libkin and L. Wong. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences*, 55(2):241–272, 1997.
110. P. Lindström. On Extensions of Elementary Logic. *Theoria*, 35, 1969.

111. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
112. J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming*. Logic Programming Series. MIT Press, 1992.
113. A. Makinouchi. A Consideration of Normal Form of Not-necessarily-normalized Relations in the Relational Data Model. *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 447–453, 1977.
114. F. Manola and E. Miller. RDF Primer. Recommendation, W3C, 2004.
115. W. Marek and M. Truszczyński. Autoepistemic Logic. *Journal of the Association for Computing Machinery*, 38(3):588–619, 1991.
116. A. Martelli and U. Montanari. Unification in Linear Time and Space: a Structured Presentation. Technical Report B 76-16, University of Pisa, 1976.
117. G. Miklau and D. Suciu. Containment and Equivalence for a Fragment of XPath. *Journal of the Association for Computing Machinery*, 51(1):2–45, 2004.
118. J. Minker. On Indefinite Data Bases and the Closed World Assumption. In D. Loveland, editor, *Proc. International Conference on Automated Deduction (CADE)*, number 138 in Lecture Notes in Computer Science, pages 292–308, New York, 1982. Springer-Verlag.
119. J. Minker. Overview of Disjunctive Logic Programming. *Annals of Mathematics and Artificial Intelligence*, 12:1–24, 1994.
120. M. Minoux. LTUR: A Simplified Linear-Time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters*, 29(1):1–12, 1988.
121. P. Narendran. Unification Modulo ACI+1+0. *Fundamenta Informaticae*, 25(1):49–57, 1996.
122. W. Nejdl. Recursive Strategies for Answering Recursive Queries - The RQA/FQI Strategy. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 43–50. Morgan Kaufmann Publishers Inc., 1987.
123. I. Niemelä and P. Simons. Efficient Implementation of the Well-founded and Stable Model Semantics. In *Proc. Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.
124. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
125. C. Papadimitriou. A Note on the Expressive Power of Prolog. *Bulletin of the EATCS*, 26:21–23, 1985.
126. C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Co., 1994.
127. C. Papadimitriou and M. Yannakakis. A Note on Succinct Representations of Graphs. *Information and Control*, 71:181–185, 1985.
128. S. Paparizos and H. V. Jagadish. Pattern Tree Algebras: Sets or Sequences? In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 349–360. VLDB Endowment, 2005.
129. J. Paredaens and D. V. Gucht. Converting Nested Algebra Expressions into Flat Algebra Expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992.
130. M. Paterson and M. Wegman. Linear Unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
131. W. Plandowski. Satisfiability of Word Equations with Constants is in PSPACE. In *Proc. Annual Symposium on Foundations of Computer Science (FOCS)*, pages 495–500, 1999.

132. T. Przymusinsik. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.
133. T. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.
134. T. Przymusinski. Static Semantics for Normal and Disjunctive Logic Programs. *Annals of Mathematics and Artificial Intelligence*, 14:323–357, 1995.
135. T. C. Przymusinski. On the Declarative and Procedural Semantics of Logic Programs. *Journal of Automated Reasoning*, 5(2):167–205, 1989.
136. P. Ramanan. Efficient Algorithms for Minimizing Tree Pattern Queries. In *Proc. ACM SIGMOD Symposium on the Management of Data (SIGMOD)*, pages 299–309, New York, NY, USA, 2002. ACM Press.
137. R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Base*, pages 55–76. Plenum Press, New York, NY, USA, 1978.
138. J. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the Association for Computing Machinery*, 12(1):23–41, 1965.
139. K. Ross. Modular Stratification and Magic Sets for Datalog Programs with Negation. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1990.
140. K. Ross. A Procedural Semantics for Well-Founded Negation in Logic Programs. *Journal of Logic Programming*, 13:1–22, 1992.
141. S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proc. Extreme Markup Languages*, 2004.
142. J. Schlipf. The Expressive Powers of the Logic Programming Semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995.
143. T. Schwentick. XPath Query Containment. *SIGMOD Record*, 33(1):101–109, 2004.
144. H. Schwichtenberg. Logikprogrammierung. Institute for Mathematics, University of Munich, 1993.
145. J. Sheperdson. Unsolvable Problems for SLDNF-Resolution. *Journal of Logic Programming*, 10:19–22, 1991.
146. D. W. Shipman. The Functional Data Model and the Data Languages DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.
147. G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Bringing Order to Query Optimization. *SIGMOD Record*, 31(2):5–14, 2002.
148. M. Stonebraker, G. Held, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.
149. D. Suciu and J. Paredaens. Any Algorithm in the Complex Object Algebra with Powerset needs Exponential Space to Compute Transitive Closure. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 201–209, New York, NY, USA, 1994. ACM Press.
150. H. Tamaki and T. Sato. OLDT Resolution with Tablulation. In *International Conference on Logic Programming*, pages 84–98, 1986.
151. S.-A. Tärnlund. Horn Clause Computability. *BIT Numerical Mathematics*, 17:215–216, 1977.
152. J. van den Bussche and J. Paredaens. The Expressive Power of Structured Values in Pure OODBs. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 291–299, 1991.
153. M. van Emden and R. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the Association for Computing Machinery*, 23(4):733–742, 1976.

154. A. van Gelder. The Alternating Fixpoint of Logic Programs With Negation. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 1–10, 1989.
155. A. van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, 1991.
156. M. Vardi. The Complexity of Relational Query Languages. In *ACM Symposium on Theory of Computing (STOC)*, pages 137–146, San Francisco, 1982.
157. H. Veith. Logical Reducibilities in Finite Model Theory. Master's thesis, Information Systems Department, TU Vienna, Austria, Sept. 1994.
158. L. Vieille. A Database-Complete Proof Procedure Based on SLD-Resolution. In *International Conference on Logic Programming*, pages 74–103, 1987.
159. G. von Bültzingsloewen. Translating and Optimizing SQL Queries Having Aggregates. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 235–243, San Francisco, CA, USA, 1987.
160. D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the Language and its Implementation compared with Lisp. In *Proc. Symposium on Artificial Intelligence and Programming Languages*, pages 109–115, 1977.
161. P. T. Wood. Optimising Web Queries using Document Type Definitions. In *Proc. ACM Int'l. Workshop on Web Information and Data Management (WIDM)*, pages 28–32, New York, NY, USA, 1999. ACM Press.
162. H. Yasuura. On Parallel Computational Complexity of Unification. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 235–243. ICOT, 1984.