Rules and Logic Programming for the Web

Adrian Paschke

Freie Universität Berlin
AG Corporate Semantic Web
Königin-Luise-Str. 24/26, 14195 Berlin, Germany
paschke@inf.fu-berlin.de
http://www.corporate-semantic-web.de/

Abstract. This lecture script gives an introduction to rule based knowledge representation on Web. It reviews the logical foundations of logic programming and derivation rule languages and describes existing Web rule standard languages such as RuleML, the W3C Rule Interchange Format (RIF), and the Web rule engine Prova.

1 Introduction to Rule Based Knowledge Representation

Knowledge representation (KR) focuses on methods for describing the world in terms of high-level, abstracted models which can be used to build intelligent applications, i.e., it provides methods to find implicit consequences of explicitly represented knowledge. Approaches can be roughly divided into *logic based formalisms*, usually a variant of first-order predicate calculus and *non-logic based formalisms* such as graphical semantic networks, object frames or (early) production rule systems. Non-logic based approaches, which are often based on ad hoc data structures and graphical representations, typically lack a precise formal semantics which makes it hard to verify the correctness of drawn consequences. On the other hand, logic based approaches use the powerful and general semantics of first-order logic (FOL) (typically a decidable subset of FOL) which allows a precise characterization of the meaning of a world by expressing it as a knowledge base (KB) of statements in a language which has a truth theory. While the syntax may differ, the semantics of FOL KBs is often given in a Tarski-style semantics.

Rule based systems have been investigated comprehensively in the realms of declarative programming and expert systems over the last decades. Using (inference) rules has several advantages: reasoning with rules is based on a semantics of formal logic, usually a variation of first order predicate logic, and it is relatively easy for the end user to write rules. The basic idea is that users employ rules to express *what* they want, the responsibility to interpret this and to decide on *how* to do it is delegated to an interpreter (e.g., an inference engine or a just in-time rule compiler). Traditionally, rule based systems have been supported by two types of inferencing algorithms: forward chaining and backward chaining.

A. Polleres et al. (Eds.): Reasoning Web 2011, LNCS 6848, pp. 326–381, 2011.

1.1 Forward Chaining Rule Systems

Forward chaining is one of the two main methods of reasoning when using "if-then" style inference rules in artificial intelligence. Forward chaining is data-driven. The inference engine makes inferences based on rules from given data. It starts with the available data and uses inference rules to extract more data until an optimal goal is reached. An inference engine using forward chaining searches the inference rules until it finds one where the *if clause* is known to be true. When found it can conclude, or infer, the *then clause*, resulting in the addition of new information to its KB. The most common form of forward chaining is the Rete algorithm. In a nutshell, this algorithm keeps the derivation structure in memory and propagates changes in the fact and rule base. There are many forward chaining implementations in the area of deductive databases and many well-known forward-reasoning engines for production rules ("if condition then action" rules) such as IBM ILOG's commercial rule system or popular open source solutions such as Drools, CLIPS or Jess which are based on variants of the Rete algorithm.

1.2 Backward Chaining Rule Systems

The other main reasoning method for if-then rules is backward chaining which is typically used in logic programming, where the rules are called derivation rules. Backward chaining starts with a list of goals (hypothesis) and works backwards to see if there are data available that will support any of these goals. Accordingly, backward chaining is goal-driven. An inference engine using backward chaining would search the inference rules until it finds one which has a *then clause* that matches a desired goal. If the *if clause* of that inference rule is not known to be true, then it is added to the list of goals. The common deductive computational model of logic programming uses backward-reasoning (goal-driven) *resolution* to instantiate the program clauses via goals and uses *unification* to determine the program clauses to be selected and the variables to be substituted by terms. The unification algorithm supports backtracking usually according to depth-first recursive backward chaining, but forward chaining bottom-up approaches are also possible.

1.3 Discussion Backward Chaining vs. Forward Chaining in the Web Context

Forward chaining, e.g. based on the Rete algorithm in production rules, can be very effective, e.g., if you just want to find out what new facts are true or when you have a small set of initial facts and when there tends to be lots of different rules which allow you to draw the same conclusion. However, in the context of reasoning on top of Web content backward chaining often qualifies to be the better choice:

- In forward-reasoning additional software must propagate changes to the memory based fact base which leads to a lot of redundancy and difficulties, e.g., a Web content database normally does not propagate changes and

for dynamic real-time access the fact base and the Web database must be synchronized.
- In Web applications often large set of initial facts are provided which are likely to change. Using forward chaining, lots of rules would be eligible to fire in any cycle and a lot of irrelevant conclusions are drawn. In backward-reasoning the knowledge base can be temporarily populated with the needed facts from external Web systems to answer a particular goal at query time which can be discarded from the memory afterwards. Forward-reasoning on the Web works best only for closed scopes, e.g., firing rules when certain events occur.
- Open-distributed environments such as the Web are usually based on a pull-model and most implementations of push-architectures (the push model relates to active event processing) are basically pull-concepts, i.e., the push functionality is simulated by frequently issuing queries, e.g., a mail client which queries the mailbox every second for new mails. Therefore, a goal-driven backward-reasoning system perfectly fits to those architectures.
- Forward-reasoning production rules have an operational semantics but no clear logical semantics and a restricted expressiveness, e.g. no recursion, only inflationary negation etc.

The further paper is structured as follows: Section 2 describes logical foundations of logic programming, rules and reasoning. Section 3 introduces standard Web rule languages on the platform independent *interchange* level and the platform specific *execution* level. In particular, the W3C Rule Interchange Format (RIF), RuleML and the Prova rule language (ISO Prolog like syntax) are detailed in this section. Finally, the conclusion summarizes the current state-of-art and future trends.

2 Logic Foundations

This section reviews general background knowledge about logic and logic programming and its use for rule based knowledge representation and reasoning.

2.1 First-Order Logic

This subsection recalls the definition of a first order logic (FOL) language and classical FOL models (structures) under Tarski semantics adopted from [62, 63, 46]. Both are interrelated concepts and play a central role in logic and form a general basis that allows to cover a wide range of logical formalisms for rule based reasoning and knowledge representation.

Syntax. This subsection defines the syntax of a first order language according to [62, 63, 46].

Definition 1. *(**Signature**) S is a signature if S is a four-tuple $\langle \overline{P}, \overline{F}, arity, \overline{c} \rangle$ where:*

1. $\overline{P}$ *is a finite sequence of predicate symbols* $\langle P_1, .., P_n \rangle$.
2. $\overline{F}$ *is a finite sequence of function symbols* $\langle F_1, .., F_m \rangle$
3. *For each* P_i *respectively each* F_j, $arity(P_i)$ *resp.* $arity(F_j)$ *is a non-zero natural number denoting the arity of* P_i *resp.* F_i.
4. $\overline{c} = \langle c1, .., c_o \rangle$ *is a finite or infinite sequence of constant symbols.*

A signature is called function-free if $\overline{F} = \emptyset$.

Definition 2. *(**Alphabet**) An alphabet* Σ *consists of the following class of symbols:*

1. *A signature* $S = \langle \overline{P}, \overline{F}, arity, \overline{c} \rangle$.
2. *A collection of variables V which will be denoted by identifiers starting with a capital letter like U,V,X*
3. *Logical connectives / operators:* $\neg$. *(negation),* $\wedge$ *(conjunction),* $\vee$ *(disjunction),* $\rightarrow$ *(implication),* $\equiv$ *(syntactical equivalent),* $=$ *(equivalence),* $\bot$ *(bottom),* $\top$ *(top).*
4. *Quantifier:* $\forall$ *(forall),* $\exists$ *(exists).*
5. *Parentheses and punctation symbols:* (,) *and* ,.

Definition 3. *(**Terms**) A term is defined inductively as follows:*

1. *A variable is a term.*
2. *A constant in* $\overline{c}$ *is a term.*
3. *If f is a function symbol with arity n and* $t_1, .., t_n$ *are terms, then* $f(t_1, .., t_n)$ *is a (complex) term.*

Function symbols are written in prefix notation whereby a function always precedes its terms. However, usually terms are also composed of both prefix and infixed symbols, e.g., $(f(2) - f(1)/f(1))$. There are standard ways of dealing with these issues.

Definition 4. *(**Atom**) Let p be a predicate symbol with arity* $n \in \aleph$. *Let* $t_1, .., t_n$ *be terms, then* $p(t_1, .., t_n)$ *is an atomic formula of terms. A* ground atom *is an atomic formula without variables.*

Definition 5. *(**Well-formed Formula**) A (well-formed) formula is defined as follows:*

1. *An atom is a formula.*
2. *If H and G are formulas then*
   - $\neg H$ *is a formula (negation)*
   - $(H \wedge G)$ *is a formula (conjunction)*
   - $(H \vee G)$ *is a formula (disjunction)*
   - $(H \rightarrow G)$ *is a formula (implication)*
   - $(H \equiv G)$ *is a formula (equivalence)*
3. *If H is a formula and X is a variable, then* $(\forall X H)$ *and* $(\exists X H)$ *are formulas.*

The following precedences are defined:

1. $\neg$, $\forall$, $\exists$
2. $\wedge$, $\vee$
3. $\rightarrow$, $\equiv$

Definition 6. *(**First-Order Language**) A* FOL language *is defined over an alphabet Σ where the signature S may vary from language to language. It consists of the set of all formulas that can be constructed according to the definitions of well-founded formulas using the symbols of Σ. A FOL language is called function-free if the signature is function-free.*

Thus, a language in addition to a signature also contains the logical symbols and a list of variables. The notion "first-order" refers to the fact that quantification is over individuals rather than classes (or functions).

Definition 7. *(**Scope of Variables**) Let X be a variable and H be a formula. The scope of $\forall X$ in $\forall X H$ and of $\exists X$ in $\exists X H$ is H. Combinations of $\forall X$ and $\exists X$ bind every occurrence of X in their scope. Any occurrences of variables that are not bound are called free.*

Definition 8. *(**Open and Closed Formula**) A formula is open if it has free variables. A formula is closed if it has no free variables.*

Definition 9. *(**Literal**) A literal L is an atom or the negation of an atom.*

Definition 10. *(**Complement**) Let L be a literal. The complement $-L$ of L is defined as follows:*

$$-L := \begin{cases} \neg A \text{ if } L \equiv A \\ A \text{ if } L \equiv \neg A \end{cases}$$

where A is an atom.

Definition 11. *(**Theory**) A FOL theory Φ or FOL knowledge base is a set of formulas in a FOL language Σ: $\Phi \subseteq \Sigma$. The signature S of Φ is obtained from all the constant, function and predicate symbols which occur in Φ.*

Every finite FOL knowledge base (FOL KB) is equivalent to the conjunction of its elements, i.e., it might be equivalently written as a conjunction of formulas.

Interpretations and Models. This subsection is concerned with attributing meaning (or truth values) to sentences (well-formed formulas) in a FOL language. The definitions follow [62, 63, 46]. Informally, the sentences are mapped to some statements about a chosen domain through a process known as interpretation. An interpretation which gives the value true to a sentence is said to satisfy the sentence. Such an interpretation is called a model for the sentence and an interpretation which does not satisfy a sentence is called a counter-model.

Definition 12. *(**Interpretation / Structure**) Let $S = \langle \overline{P}, \overline{F}, arity, \overline{c} \rangle$ be a signature. I is called an interpretation (or a structure) for S if $I = \langle |M|, \overline{P}^I, \overline{F}^I, \overline{c}^I \rangle$ consists of:*

1. *a non-empty set $|M|$ called the universe of I or the domain of the interpretation. The members of $|M|$ are called individuals of I.*
2. *$\overline{P}^I = \langle P_1^I, .., P_k^I \rangle$ associates with each predicate P_i in S of arity $n = arity(P_i)$ an n-ary relation P_i^I on $|M|$, i.e., $P_i^I \subseteq |M|^n$, where $|M|^n$ denotes the collection of all n-tuples from $|M|$.*
3. *$\overline{F}^I = \langle F_1^I, .., F_l^I \rangle$ is an interpretation for each function symbol F_j of arity m, where F_j^I is an m-place function $\overline{F}_j^I : |M|^m \rightarrow |M|$, i.e., F_j^I is defined on the set of m-tuples of individuals $|M|^m$ with values in $|M|$.*
4. *$\overline{c}^I = \langle c^I | c = constant \rangle$ is an interpretation for the constants of S: $c \in S$, where c^I is an individual of M: $c^I \in |M|$.*

Definition 13. ***(Assignment)***

1. Variable Assignment*: Let Σ be a FOL language with $\overline{X}$ its set of variables, and I an interpretation for Σ. An assignment is a function σ from $\overline{X}$ into the universe of Σ.*
2. Term Assignment*: Let I be an interpretation of a FOL language Σ with domain $|M|$ and variable assignment σ. The term assignment wrt σ of the term in Σ is defined as:*
   - Each variable is given its assignment according to σ.
   - Each constant is given its assignment according to I.
   - If $t'_1, .., t'_n$ are the term assignments of $t_1, .., t_n$ and f' is the assignment of the function symbols f with arity n, then $f'(t'_1, .., t'_n) \in |M|$ is the term assignment of $f(t_1, .., t_n)$.

That is, given an assignment σ, any variable term of the language that is in the domain of σ is given a constant value in $|M|$.

Definition 14. ***(Truth Values)*** *Let I be an interpretation of a FOL language Σ with domain $|M|$ and σ be a variable assignment. A formula $F \in \Sigma$ can be given a truth value "false" or "true" as follows:*

1. *If the formula is an atom $p(t_1, .., t_n)$ then the truth value is obtained by calculating the value of $p'(t'_1, .., t'_n)$ where p' is the mapping assigned to p by I and $t'_1, .., t'_n$ are the term assignments of $t_1, ..t_n$ wrt I and $\overline{X}$.*
2. *The truth values of the following formulas is given by the following table:*

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F = G$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

3. *If $\exists X F$, then the truth value of the formula is true if there exists $c \in |M|$ such that the formula F has truth value "true" wrt I and $\sigma(X/c)$; otherwise it is false.*
4. *If the formula has the form $\forall X F$, then the truth value of the formula is true if, for all $c \in |M|$ F is "true" wrt I and $\sigma(X/c)$; otherwise, its truth value is false.*

The satisfaction relation $\models$ goes back to A. Tarski and is a major achievement in logic.

Definition 15. *(**Satisfaction**) If F is a formula and σ is an assignment to the interpretation I of a FOL language Σ, then the relation $I \models F[\sigma]$ means that F is true in I when there is a substitute for each free variable X of F with the value of $\sigma(X)$. The inductive requirements of "$\models$" are:*

1. *For any atomic formula of the form $p(t_1, ..t_n)[\sigma]$ iff $\langle t_1^\sigma, .., t_n^\sigma \rangle \in p^I$.*
2. *$I \models \neg F[\sigma]$ iff it is not the case that $I \models F[\sigma]$*
3. *$I \models (F \wedge G)[\sigma]$ iff both $I \models F[\sigma]$ and $I \models G[\sigma]$. Similarly, for the other statements.*
4. *$I \models \exists X F[\sigma]$ iff there exists some assignment σ' such that*
   - for every variable Y different from X $\sigma'(Y) = \sigma(Y)$
   - $\sigma'(X)$ is defined and $I \models F[\sigma']$
5. *$I \models \forall X F[\sigma]$ iff for any assignment σ', if $\sigma'(X)$ is defined and σ' is equal to σ on each variable different from X, then $I \models F[\sigma']$*

Accordingly, a formula F is satisfied by an interpretation I (F is true in I: $I \models F$) iff $I \models_\sigma F$ for all variable assignments σ. F is valid iff $I \models F$ for every interpretation I.

Definition 16. *(**Model**) Let I be an interpretation of a FOL language Σ. Then I is a model of a closed formula F, if F is true wrt I. Further, I is a model of a set $\overline{F}$ of closed formulas, if I is a model of each formula of $\overline{F}$. I is a model of an FOL KB Φ iff $I \models F$ for every formula $F \in \Phi$: $I \models \Phi$.*

Definition 17. *(**Logical Consequence, Entailment, Logical Implication**) A formula $F \in \Sigma$ is a logical consequence of a FOL KB Φ written as $\Phi \models F$, i.e., Φ entails F iff for all models $I \in \Sigma$ for which $I \models \Phi$ also $I \models F$. For a fixed FOL language (and signature) Σ let Φ and Ψ be two sets of sentences (two KBs), then $\Phi \rightarrow \Psi$ means that for every interpretation I of Σ, if I is a model for Φ then it is also a model for Ψ.*

$\Phi \rightarrow \Psi$ is also meaningful when Φ and Ψ are sets of formulas with variables, i.e., for every interpretation I of Σ and every assignment σ in I, if $I[\sigma]$ satisfies every formula in Φ then it also satisfies every formula in Ψ.

2.2 Logic Programming

Full first-order logic is not suitable as a declarative programming language, e.g. due to the following reasons:

- unrestricted FOL is in general undecidable
- the results are not always unique
- finding (most general) unifier and solving formula is highly complex
- large search domains, which must be restricted using complex control structures
- danger of implementation incompleteness

Hence, logic programming is based on a subset of FOL which deals with a specific class of well-formed formulas, so called statement clauses which consist of an antecedent part and a consequent. The declarative meaning for such clauses is that the consequent part is true, if the antecedents are true. The procedural meaning is, that the consequent is proven by reducing it to a set of sub-goals given by the antecedent part. The most common form of logic programming is based on Horn Logic where clauses in normal form only have one positive literal which is the consequent. Such programs are called definite LPs or Horn LPs. The semantics of definite Logic Programs (LPs) is based on minimal Herbrand models. Although definite LPs are expressive enough to model many problems the formulation is often neither easy nor elegant. Hence, extensions to definite LPs like different forms of negations have been proposed. This subsection introduces relevant terms, concepts, syntax and semantics of different classes of logic programs (LPs) derived from [62, 63, 46].

Syntax of Logic Programs

Definition 18. *(**Clause**) A clause is a formula such as* $\forall \overline{X}(L_1 \vee .. \vee L_m)$ *where each* L_i *is a literal and* $\overline{X} = \{X_1, .., X_n\}$ *are all the variables occurring in* $L_1 \vee .. \vee L_m$.

Different classes of clauses are distinguished: *propositional clauses*, *Datalog clauses*, *definite Horn clauses*, *normal clauses*, *extended clauses*, *positive clauses*, *positive-disjunctive clauses*, *disjunctive clauses*, and *extended disjunctive clauses*. Associated with each type of clause is a class of logic programs: *propositional LP*, *Datalog LP*, *definite LP*, *stratified LP*, *normal LP* (aka general LP), *extended LP*, *disjunctive LP* and combinations of classes, with an increasing expressiveness as illustrated in figure 1 for several classes of LPs. Each class can be propositional (without terms), Datalog (without functions) or with terms and variables.

These LPs are defined as follows:

Definition 19. *(**Logic Programs and Rules**) Given a FOL language* Σ*, a (disjunctive extended) logic program* P *consists of logical rules (or program clause) of the form*

$A_1, .., A_k \leftarrow B_1, .., B_m, not\, C_1, .., not\, C_n$

or equivalently

$\forall \overline{X}(A_1 \vee .. \vee A_k \leftarrow B_1 \wedge .. \wedge B_m \wedge not\, C_1 \wedge .. \wedge not\, C_n)$

which is a convenient notation for a FOL clause where all variables $X_i \in \overline{X}$ *occurring in the literals* A_i, B_j, C_k *are universally quantified* $\forall X_1 .. \forall X_s$, *the commas in the antecedent denote conjunction and the commas in the consequent denote disjunction, and not denotes negation by default, rather than classical negation. For short a rule is denoted in set notation as:*

$A \leftarrow B \wedge not\, C$

where $A = A_1 \vee .. \vee A_k$, $B = B_1 \wedge .. \wedge B_m$, $C = C_1 \vee .. \vee C_n$. *Note that* C *is a disjunction and according to De Morgan's law* $not\, C$ *is taken to be a conjunction.*

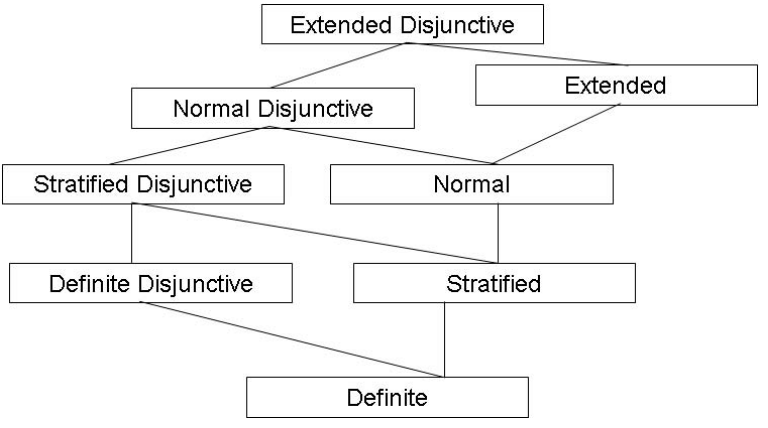

Fig. 1. Classes of LPs

The A is called the rule head *which consists of the set of head literals and B and C is called the* rule body *which consists of the set of body literals. Note that this set notation is legitime because the conjunction is commutative.*

A clause is called:

- *a* fact *if* $m = n = 0$, *i.e.,* $A \leftarrow \emptyset$
- *a* query *(or goal) if* $k = 0$, *i.e.,* $\leftarrow B \wedge C$. *A query or goal is called atomic if it consists of a single literal* B_1, *i.e.,* $m = 1$ *and* $n = 0$.
- *a* propositional rule *if the arity of all predicates is* 0, *i.e., all literals are propositional ones. If all rules in a program P are propositional the P is called a* propositional LP.
- *a* Datalog rule *if it contains no functions, i.e., is function-free and no predicate symbol of the input schema appears in the rule head. A* Datalog LP *(aka deductive database) is a function-free LP.*
- *a* definite or positive rule *(or* Horn clause*) if all literals are atoms,* $n = 0$ *and* $k = 1$, *i.e., it neither contains negation nor disjunction. The corresponding LP is called* positive or definite LP *(or* Horn Program*).*
- positive-disjunctive rule *if all literals are atoms and* $n = 0$, *i.e., it does not contain negation. The corresponding LP is called* positive-disjunctive LP.
- normal rule *if all literals are atoms and* $k = 1$, *i.e., it does not contain disjunction. The corresponding program is called a* normal LP.
- extended rule *if* A_i, B_i *and* C_i *are literals, i.e., are atoms or explicitly negated atoms . The corresponding programm is called an* extended LP.
- disjunctive rule *if* $k > 1$, *i.e., it does contain a disjunction. The corresponding program is called a* disjunctive LP.
- range-restricted *if all variable symbols occurring in the head also occur in the positive body.*
- ground *if no variables occur in it.*

Semantics of Logic Programs. Proof-theoretically the semantics of a logic program P is defined as a set of literals that is (syntactically) derivable from P using a particular derivation mechanism such as SLDNF resolution. Model-theoretically, a semantics for a logic program P is concerned with attributing meaning (truth values) to clauses (rules). The properties of soundness and completeness establish a relation between the notions of syntactic ($\vdash$) and semantic ($\models$) entailment in logic programming. This subsection reviews several approaches to define proof-theoretic and model-theoretic semantics for different types of logic programs.

Substitution and Unification. At first, the concepts of substitution and unification from [62, 52] are introduced which are at the heart of *proof-theoretic semantics* of non-ground LPs.

Definition 20. *(**Substitution**) A substitution θ in a language Σ is a finite set of the form $\{X_1/t_1, .., X_n/t_n\}$, where each X_i is a variable in Σ, each t_i is a term in Σ distinct from X_i and the variables $X_1, .., X_n$ are pairwise distinct. Each element X_i/t_i is called a binding for X_i. θ is called a ground substitution if the t_i are all ground terms. θ is called a variable-pure substitution if the t_i are all variables.*

Definition 21. *(**Expression**) An expression E is either a term, a literal or a conjunction or disjunction of literals.*

Definition 22. *(**Instance**) Let $\theta = \{X_1/t_1, .., X_n/t_n\}$ be a substitution and E be an expression then $E\theta$ is the instance of E by θ is the expression obtained from E by simultaneously replacing each occurrence of the variable X_i in E by the term t_i for $i = 1, .., n$. if $E\theta$ is ground then $E\theta$ is called a ground instance of E.*

Definition 23. *(**Variant**) Let E and D be expressions. E and D are variants if there exists substitutions θ and σ such that $E = D\theta$ and $D = E\sigma$.*

Definition 24. *(**Renaming Substitution**) Let E be an expression and $\overline{X}$ be the set of variables occurring in E. A renaming substitution for E is a variable-pure substitution $\{X_1/Y_1, .., X_n/Y_n\}$ such that $\{X_1, .., X_n\} \subseteq \overline{X}$, the Y_i are pairwise distinct and $(\overline{X} \setminus \{X_1, .., X_n\}) \cap \{Y_1, .., Y_n\} = \emptyset$.*

Definition 25. *(**Composition**) Let $\theta = \{X_1/s_1, .., X_m/s_m\}$ and $\sigma = \{Y_1/t_1, .., Y_n/t_n\}$ be substitutions. The composition $\theta\sigma$ of θ and σ is the substitution obtained from the set*

$$\{X_1/s_1\sigma, .., X_m/s_m\sigma, Y_1/t_1, .., Y_n/t_n\}$$

by deleting any binding $X_i/s_i\sigma$ for which $X_i = s_i\sigma$ and deleting any binding Y_j/t_j for which $Y_j \in \{X_1, .., X_m\}$.

Definition 26. *(**Most General Unifier** (MGU)) Let $\overline{E}$ be a finite set of expressions. A substitution θ is called a unifier for $\overline{E}$ if $\overline{E}\theta$ is a singleton. An unifier for $\overline{E}$ is called most general unifier (MGU) for $\overline{E}$ if for each unifier σ of $\overline{E}$ there exists a substitution γ such that $\sigma = \theta\gamma$. $\overline{E}$ is called unifiable if there exists a unifier for $\overline{E}$.*

Note that a MGU for a set of expressions is unique modulo renaming if there exists a MGU at all.

Minimal Herbrand Model. For the *model-theoretic semantics* first the minimal or least Herbrand model semantics is introduced which is considered as the natural interpretation of a definite LP. Then the minimal Herbrand semantics is extended for other more expressive subclasses of LPs and further (declarative) semantics are introduced together with their proof-theoretic counterparts for logic programming.

Definition 27. *(**Herbrand Universe**) The Herbrand universe of a program P defined over the alphabet Σ, denoted U_P, is the set of all ground terms which can be formed out of the constants and function symbols of the signature S of Σ.*

Definition 28. *(**Herbrand Base**) The Herbrand base of a program P, denoted B_P, is the set of all ground atomic literals which can be formed by using the predicate symbols in the signature S of Σ with the ground terms in U_P as arguments.*

Definition 29. *(**Herbrand Instantiation aka Grounding**) The Herbrand instantiation ground(P) of P consists of all ground instances of all rules in P wrt the Herbrand universe U_P which can be obtained as follows: The ground instantiation of a rule r is the collection of all formulas $r[X_1/t_1, .., X_n/t_n]$ with $X_1, .., X_n$ denoting the variables which occur in r and $t_1, .., t_n$ ranging over all terms in U_P.*

Definition 30. *(**Herbrand Interpretation**) The Herbrand interpretation I^{Herb} of P is a consistent subset of B_P. The interpretation is given as follows:*

1. *The domain of the interpretation is the Herbrand universe U_P.*
2. *Constants are assigned themselves in U_P.*
3. *If f is a function in P with arity n then the mapping $f' : U_P^n \mapsto U_P$ assigned to f is defined by $f'(t_1, ..t_n) := f(t_1, .., t_n)$.*

Note that since the assignment to constant and function symbols is fixed for Herbrand interpretations, it is possible to identify a Herbrand interpretation with a subset of the Herbrand base. For any Herbrand interpretation, the corresponding subset of the Herbrand base is the set of all ground atoms which are true wrt the interpretation.

Definition 31. *(**Herbrand Model**) Let P be a positive / definite program. A Herbrand interpretation I^{Herb} of P is a model of P, denoted as M^{Herb}, iff for every rule $H \leftarrow B_1, .., B_n \in ground(P)$ the following holds: If $B_1, .., B_n \in I^{Herb}$ then $H \in I^{Herb}$.*

The Herbrand model M^{Herb} satisfies the *unique name assumption*, i.e., for any two distinct ground terms in B_P, their interpretations are distinct as well.

Definition 32. *(**Unique Name Assumption and Domain Closure Assumption**) Let Σ be a given language. The unique name assumption (UNA)*

restricts the model M^{Herb}, where syntactically different ground terms t_1, t_2 are interpreted as non-identical elements: $t_1^{M^{Herb}} \neq t_2^{M^{Herb}}$.
The domain closure assumption (DCA) is a restriction to those models M^{Herb} where for any element a in M^{Herb} there is a term t that represents this element: $a = t^{M^{Herb}}$.

Model-theoretically the intended meaning of a LP is that a formula should be true if it is a logical consequence of the program, i.e., it is true in all models of the program. For definite LPs this intention leads to a semantics that coincides with the intuition because of the model intersection property.

Definition 33. *(**Model Intersection Property**) Let $\overline{M}^{Herb}$ be the set of all Herbrand models of a program P.The intersection of all Herbrand models $\bigcap \overline{M}^{Herb}(P)$ of a definite LP P is also a Herbrand model of P.*

Note that since every definite LP P has B_P as an Herbrand model, the set of all Herbrand models for P is always non-empty: $\bigcap \overline{M}^{Herb}(P) \neq \emptyset$.

Definition 34. *(**Minimal Herbrand Model**) Let P be a definite LP then the minimal or least Herbrand model M_P^{Herb} of P is the intersection of all Herbrand models for P.*

The constructive computational characterization of the minimal Herbrand model of a definite LP P is based on the least fixpoint of the immediate consequence operator of P. A detailed description of the theory of lattices and fixpoints can be found in [62, 52]. Here the relevant definitions are recalled.

Definition 35. *(**Immediate Consequence Operator**) Let P be a definite LP. Let $I^{Herb} \subseteq B_P$ be a set of atoms. The set of immediate consequences of I^{Herb} wrt P is defined as follows:*
$T_P(I^{Herb}) := \{A \text{ — there is } A \leftarrow B \in ground(P) \text{ with } B \subseteq I^{Herb}\}$.

Definition 36. *(**Monotonic Mapping**) Let $T : P(U) \rightarrow P(U)$ be a mapping then T is monotonic if $T(X) \subseteq T(Y)$, whenever $X \subseteq Y$.*

Definition 37. *(**Ordinal Power of T**) Let $T : P(U) \rightarrow P(U)$ be a monotonic mapping then:*
$T \uparrow 0 = \emptyset$
$T \uparrow a = T(T \uparrow (a-1))$ *if a is a successor ordinal*
$T \uparrow a = \bigcup (T \uparrow b | b < a)$ *if a is a limit ordinal*

Definition 38. *(**Fixpoint of operators**) An operator T is a function $T : P(U) \rightarrow P(U)$, where $P(U)$ denotes the powerset of a countable set U. A set $X \subseteq U$ is called a fixpoint of the operator $T : P(U) \rightarrow P(U)$ iff $T(X) = X$*

Definition 39. *(**Least Fixpoint**) Let $T : P(U) \rightarrow P(U)$ be a mapping. An element $e \in P(U)$ is called a least fixpoint $lfp(T)$ iff e is a fixpoint of T and for all fixpoints f of T it is that $e \subseteq f$.*

According to the Fixpoint Theorem of Knaster and Tarski (see [56] for more details) each monotonic operator T has a least fixpoint $lfp(T)$, which is the least upper bound of the sequence $T^0 = \emptyset$, $T^{i+1} = T(T^i)$ for $i \geq 0$. It appears that for each set P of clauses $lfp(T)$ coincides with the unique least Herbrand model of P, where a model M^{Herb} is smaller than a model N^{Herb}, if $M^{Herb} \subset N^{Herb}$ [43].

Definition 40. *(**Fixpoints of Monotonic Mappings**) Let T be a monotonic mapping. Then T has a least fixpoint $lfp(T)$. For every ordinal a, $T \uparrow a \subseteq lfp(T)$. Moreover, there exists an ordinal b such that $c \geq b$ implies $T \uparrow c = lfp(T)$.*

If the operator T_P is not only monotonic but also continuous[1], then a least fixpoint of T_P is always reached not later than at the first upper ordinal (see [62]). By Kleene's theorem (see [37]) $lfp = T \uparrow \omega$.

Theorem 1. *(**Fixpoint Characterization of the Minimal Herbrand Model**) Let P be a definite LP then $M_P^{Herb} = lfp(T_P) = T_P \uparrow \omega$.*

In summary, the semantics of LPs is now defined as follows:

Definition 41. *(**Herbrand Semantics of Logic Programs**) Let the grounding of a clause r in a language Σ be denoted as $ground(r, \Sigma)$ where $ground(r, \Sigma)$ is the set of all clauses obtained from r by all possible substitutions of elements of U_Σ for the variables in r. For any definite LP P*

$ground(P, \Sigma) = \bigcup_{r \in P} ground(r, \Sigma)$

The operator $T_P : 2^{B_P} \rightarrow 2^{B_P}$ associates with P is defined by $T_P = T_{ground(P)}$, where $ground(P)$ denotes $ground(P, \Sigma(P))$, and accordingly:

$SEM_{Herb}(P) = M^{Herb}_{ground(P)}$.

Generating $ground(P)$ is often a very complex task, since, even in case of function-free languages, it is in general exponential in the size of P. Moreover, it is not always necessary to compute $M^{Herb}_{ground(P)}$ in order to determine whether $P \models A$ for some particular atom A. In practice, various proof-theoretic strategies of deriving atoms from a LP have been proposed. These strategies are based on variants of Robinson's famous *Resolution Principle* [87]. The major variant is SLD-resolution [57].

SLD Resolution. In a nutshell, in SLD a goal is a conjunction of atoms. A substitution θ is a function that maps variables to terms. Asking a query $Q?$, where $Q?$ may contain variables, to a program P means asking for all possible substitutions θ of the variables in $Q?$ such that $Q\theta$ follows from P, i.e., θ is the answer to Q. In other words, SLD resolution repeatedly transforms the initial goal by applying the resolution rule to an atom Q_i from the query/goal and a rule from P, unifying Q_i with the head H of the rule, i.e., it tries to find a substitution θ such that $H\theta = Q_i\theta$. The typical selection rule is to choose always the first atom in the query. This step is repeated until all goals are resolved and the empty goal is obtained.

[1] In the sense of a Scott-continuous function, which is one that preserves all directed suprema.

Example 1. (**Linear resolution computation step**)

$$\frac{\neg Q_1, .., \neg Q_n \qquad\qquad \neg A_1, .., \neg A_m, H}{\theta = unify(Q_1, \neg H)}$$

Remark: The common deductive computational model of logic programming uses backward-reasoning (goal-driven) resolution to instantiate the program clauses via goals and uses unification to determine the program clauses to be selected and the variables to be substituted by terms. In logic programming unification is used to derive specific information out of general rules which assert general information about a problem. The rules are instantiated via goals, leading to specific instances of these rules. A goal $G?$ initiates a refutation attempt unifying the goal $G?$ with the head of an appropriate rule $H \leftarrow B$ leading to an instance of the rule $(H \leftarrow B)'$ if there exists a substitution $\theta = \{V_1/t_1, .., V_n/t_n\}$ which assigns terms t_i to variables V_i such that $(H \leftarrow B)' = (H \leftarrow B)\theta$. Applying a substitution θ to a term, atom or rule (program clause) yields the instantiated term, atom or clause. For example, the rule $son(X, Y) : -parent(Y, X), male(X).$ and a goal $son(adrian, Y)?$ leads to the more specialized instance $son(adrian, Y) : -parent(Y, adrian), male(adrian)$. The instance body $B\theta$ is the goal reduction (sub goal) for further derivation leading to more specific instances. Repeating this process leads to an instance order $(H \leftarrow B) \geq (H \leftarrow B)' \geq$ whereas $\geq$ denotes the relation "more general as". The unification algorithm finds the greatest lower bounds (glb) of terms under this instance order $\geq$, i.e. if θ is a most general unifier (MGU) for a set of terms T then $T\theta$ is the glb of T.

For a more precise account see [5,62] and [59] for resolution on normal clauses. The task to find substitutions θ such that $Q\theta$ is derivable from the program P as well as M_P^{Herb} is closely related to SLD. The following properties are equivalent:

Theorem 2. ***(Soundness and Completeness of SLD)***

- $P \models \forall Q\theta$, *i.e* $\forall Q\theta$ *is true in all models of* P,
- $M_P^{Herb} \models \forall Q\theta$,
- *SLD computes an answer* τ *that subsumes* θ *wrt* Q, *i.e.,* $\exists\sigma : Q\tau\sigma = Q\theta$.

Since SLD resolution is a top-down approach which starts with the query, the main feature of it is, that it automatically ensures, that it only considers those rules that are relevant for the query to be answered (see also section 1.3 for a discussion of backward vs. forward reasoning). Rules that are not at all related are simply not considered in the course of the proof. Note that there are also several bottom up approaches for computing the least Herbrand model M_P^{Herb} from below. However, the bottom-up approach has two serious shortcomings:

1. The "goal-orientedness" from top-down approaches is lost, i.e the whole M_P^{Herb} has to be computed even for those facts that have nothing to do with the query.
2. In any step facts that are already computed before are recomputed again.

Partial solutions have been proposed, e.g., semi-naive bottom-up evaluation [98, 26] or Magic Sets techniques [13]. However, as discussed in section 1 top-down semantics are more appropriate in Web knowledge representation and the focus is on backward-reasoning logic programming techniques.

Theory of Logic Programming with Negation. Definite LPs are typically not expressive enough for general knowledge representation on the Web which is used to represent e.g. decision logics and situational logics. They e.g., exclude negative information and (non-monotonic) default statements such as *normally a implies c, unless something abnormal holds.* Such statements and the computation of default negation where the main motivation for alternative formulations of non-monotonic reasoning by circumscription [66], default reasoning [84] or autoepistemic reasoning [65]. Independently of these work in non-monotonic reasoning the proof-theory for *negation-as-finite-failure* (NAF), the well-known *SLDNF resolution* (SLD+NAF), originated from SLD resolution. In short, negation-as-finite-failure can be characterized as: A (default) negated literal $\sim L$ succeeds, if L finitely fails. See [62, 5] for the formal definition of SLDNF resolution and NAF. The implementation is often given as a cut-fail test[2]:

```
not([P|Args]) :-
    derive([P|Args]), % derive P(Args)
    !, % cut
    fail(). % fail
not([P|Args]). % positive answer
```

The corresponding model-theoretic semantics is defined by *Clark's completion* (COMP) [33] whose idea was to interpret "$\leftarrow$" in rules as "$\leftrightarrow$" in the classical sense.

Definition 42. ***(Clark's Completion COMP)*** *Clark's completion semantics COMP for a program P is given by the set of all classical models $\overline{M}(comp(P))$ of the completion theory $comp(P)$.*

See Clark's Equational Theory for more details [33]. COMP gives two rules for inferring negative information:

- Infer $\neg A$ iff $B_P \setminus \overline{M}(comp(P)) \models \neg A$
- Infer $\neg A$ iff $\overline{M}(comp(P)) \models \neg A$

But, (two-valued) COMP is incomplete and does not characterize the transitive closure correctly. In [80] various problems with loops in COMP were discussed. Therefore, Fitting [44] introduced a three-valued formulation $comp_3(P)$ of the

[2] The basic idea behind this implementation is to make a closed world assumption (i.e. all knowledge is completely known to the inference interpreter) and positively proof the existence of the negated goal literal, which would refute the negation.

two-valued COMP. It was shown by Kunen [58] that SLDNF is sound and complete wrt $COMP_3$ for propositional LPs and correct but not complete in the predicate logic case [93].

SLDNF resolution suffers from problems with loops and floundering and its implementation is only a simple test, i.e., no variable bindings are produced. See [94] for a discussion of unsolvable problems related to SLDNF. Much work has been done to define restriction properties (on the dependency graph whose vertices are the predicate symbols from a program P) for which SLDNF is complete. The important ones are briefly reviewed here:

- stratified: no predicate depends negatively on itself
- strict: there are no dependencies that are both even and odd
- allowedness: at least every variable occurring in a clause must occur in at least one positive literal of the body
- call-consistent: no predicate depends oddly on itself.
- hierarchical: no form of recursion is allowed

Stratified LPs for which the rules do not have recursion through negation have been defined by [7]. The predicates of stratified LPs can be placed into strata so that one can compute over the strata. The model-theoretic semantics, the *supported Herbrand model* M_P^{supp}, is defined by declaring M_P^{supp} as the intended model among all minimal Herbrand models of $comp(P)$ which could be obtained by iterating over the strata. Przymusinski [77] showed that the selected model was the so-called *perfect model*. The semantics of definite and stratified LPs lead to the unique minimal model semantics which is generally accepted to be the semantics for these classes of LPs.

However, this is not the case for more expressive LPs. Here are several possible ways to determine the semantics and various approaches based on extensions of the 2-valued classical logic to three-valued logics have been proposed, e.g., Fitting [44] or Kunen [58] semantics which are based on Kleene's strong three-valued logics, or the *well founded semantics* (WFS) [101] which is an extension of the perfect model semantics. Another approach is based on the tradition of non-monotonic reasoning in which the definition of entailment is based on the notion of beliefs. The *stable model semantics* (STABLE) [47] is based on this approach. For a discussion of the relationships between non-monotonic theories and logic programming see [67]. In the following, the (declarative) semantics and theory of more expressive types of LPs will be reviewed. Different semantics have been defined in the past. Table 1 gives an incomplete overview.

In the following, the prominent semantics will be described, namely well-founded semantics (WFS) and stable model semantics (STABLE) for normal LPs with its extension answer set semantics (ASS) for extended LPs.

Stable Model Semantics. The Gelfond-Lifschitz transformation P^M [47] of a normal LP P wrt to its interpretation I is obtained from the ground instance $ground(P)$ of P as follows:

Table 1. Semantics for LP Classes (adapted from [36])

Class	Semantics	Ref.
Definite LPs	Least Herbrand model: M_p	[7]
Stratified LPs	Supported Herbrand model: M_p^{supp}	[7]
Normal LPs	Clark's Completion: $COMP$	[33]
	3-valued Completion: $COMP_3$	[58, 44]
	Well-founded Semantics: WFS	[101]
	WFS^+ and $WFS^{'}$	[34]
	WFS_C	[91]
	Strong Well-founded Semantics: WFS_E	[27]
	Stable Model Semantics: $STABLE$	[47]
	Generalized WFS: $GWFS$	[10]
	$STABLE^+$	[35]
	$STABLE_C$	[91]
	$STABLE^{rel}$	[34]
	Pereira's $O - SEM$	[74]
	Partial Model Semantics: $PARTIAL$	[90]
	Regular Semantics: $REG - SEM$	[102]
	Preferred Semantics: $PREFERRED$	[39]
Extended LPs	Extended Well-founded Semantics: WFS_S	[54]
	Answer Set Semantics: ASS	[48, 49]
	Extended Well-founded Semantics: $WFSX$	[73]
General Disjunctive	Disjunctive WFS: $DWFS$	[21]
	Generalized Disjunctive WFS: $GDWFS$	[11]
	Disjunctive Stable: $DSTABLE$	[82]
Stratified Disjunctive	Perfect model $PERFECT$	[77]
	Weakly Perfect: $WPERFECT$	[75]
	Generalized Closed World Assumption: $GCWA$	
Positive Disjunctive	Weak generalized closed world assumption: $WGCWA$	[83]

Definition 43. *(**Gelfond-Lifschitz transform**) Let P be a program and $M \subseteq B_P$. The Gelfond-Lifschitz transform P^M of P (aka reduct of P) wrt M is defined by $P^M = r^M | r \in ground(P)$. It is obtained from $ground(P)$ by:*

1. *Replace in every ground rule $A \leftarrow B \wedge notC \in ground(P)$ the negative body by its truth value wrt M.*
2. *Deleting each rule r in P with $B^-(r) \cap M \neq \emptyset$ where B^- denotes the set of negated atoms in the body of the rule r.*

Based on P^M the concepts of stable models [47] and partial stable models [82] have been defined:

Definition 44. *(**Stable Model**) An interpretation I of a normal LP P is a stable model M^{Stable} of P if I is a minimal model of P^M:*

$$SEM_{Stable}(P) = \bigcap_{M^{Stable} \in SEM_{Stable}(P)} (M^{Stable} \cup neg(B_P \setminus M^{Stable}))$$

Definition 45. *(**Partial Stable Model**) A partial Herbrand interpretation is called a partial stable model of P if it is a partial minimal model of P^M.*

It can be shown that stable models are always partial models and that every stratified LP P has a unique stable model where stratified and stable semantics coincide.

Answer Set Semantics. Gelfond and Lifschitz [48,49] have extended the concept of stable models to extended and disjunctive LPs based on the notion of answer sets. The proposed answer-set semantics is defined as follows:

Definition 46. *(**Answer Set Semantics**) Let P be an extended (disjunctive) LP. P is transformed to a (explicit) negation-free program P' by replacing all negative literals $\neg A$ by positive literals A' over new predicate symbols. Every stable model M^{Stable} of P' defines an answer set of P, which is a set of literals:*
$\overline{L} = A \in B_P | M^{Stable}(A) = t \cup \neg A \in \neg B_P | M^{Stable}(A') = t$
If $\overline{L}$ does not contain complementary pairs $A, \neg A$ of literals, then the answer set is $\overline{L}$ else it is $B_P \cup \neg B_P$ is the set of all ground literals.

Associated with SEM_{Stable} are two entailment relations:

Definition 47. *(**Cautious Entailment**) An extended LP P cautiously entails a ground atomic formula a iff $a \in I$ for every answer set M^{Stable} of P.*

Definition 48. *(**Brave Entailment**) An extended program P bravely entails a ground atomic formula a iff $a \in I$ for some answer set M^{Stable} of P.*

Well-founded Semantics. There exists several definitions to well-founded semantics (WFS), e.g., [101, 45, 12, 81]. Van Gelder, Ross and Schilpf [101] were the first to extend the work of Apt et al. [7] to the class of normal logic programs. The well-founded semantics (WFS) of Gelder et al. is a three-valued logic: *true*, *false* and *unknown*. WFS is an extension of the perfect model semantics, in contrast to Fitting and Jacob's semantics which is based on Kleene's strong three valued logic. For instance, WFS (as well as perfect model semantics) assigns the truth value "false" to a clause $p \leftarrow p$ while Fitting and Jacob assign "unknown". Following the definition from [101] WFS is defined as follows.

Definition 49. *(**Partial Interpretation**) Let P be a normal LP. A partial interpretation I is a set of ground literals such that for no atom A both A and not A are contained in I, i.e., $pos(I) \cap neg(I) = \emptyset$ and whose atoms are contained in B_P of P, i.e., $pos(I) \cup neg(I) \subseteq B_P$. I is a total interpretation, if I is a partial interpretation and for every atom $A \in B_P$ it contains A or not A, i.e., $pos(I) \cup neg(I) = B_P$.*

Definition 50. *(**Unfounded Set**) Let P be a normal LP. Let I be a partial interpretation. Let $\alpha \subseteq B_P$ be a set of ground atoms. α is an unfounded set of P wrt I, if for every atom $A \in \alpha$ and every ground rule instance $A \leftarrow \beta \in ground(P)$ at least one of the following conditions holds:*

1. *at least one body literal $L \in \beta$ is false in I.*
2. *at least one positive body literal $B \in \beta$ is contained in α.*

Definition 51. *(**Greatest Unfounded Set**) Let P be a normal LP. Let I be a partial interpretation. The greatest unfounded set of P wrt I is the union of all unfounded sets of P wrt I.*

Definition 52. *(**Pos. and Neg. Immediate Consequences**) For a ground normal LP P and a partial interpretation $I \subseteq B_P$ the following monotonic transformation operators are defined:*

- $T_P(I) := A \in B_P | \exists (A \leftarrow \beta) \in ground(P) : \beta \subseteq I$
- $U_P(I) :=$ *the greatest unfounded set of P wrt I*
- $W_P(I) := T_P(I) \cup \ \sim U_P(I)$

Lemma 1. *T_P, U_P and W_P are monotonic operators.*

Theorem 3. *Let P be a normal LP. For every countable ordinal α, $W_P \uparrow \alpha$ is a partial model of P.*

Definition 53. *(**Well-founded Model**) The least fixpoint of W_P is the well-founded (partial) model of P denoted W_P^*. The least fixpoint can be computed as follows, $lfp(W_P) = W_P^\infty(\emptyset)^3$. If $lfp(W_P) \subseteq B_P$ is a total interpretation of P then $lfp(W_P)$ is a well-founded model. An atom $A \in B_P$ is well-founded (resp. unfounded) wrt P iff A (resp. $\neg A$) is in $lfp(W_P)$.*

WFS is defined for the grounding of an arbitrary normal LP: $ground(P)$, i.e., it defines a mapping SEM_{WFS}, which assigns to every normal LP P a set $SEM_{WFS}(P)$ of (partial) models of P such that $SEM_{WFS}(P) = SEM_{WFS}(ground(P))$ (i.e., SEM_{WFS} is instantiation invariant).

Definition 54. *(**Well-founded Semantics**) The Well-founded semantics (WFS) assigns to every normal LP P the well-founded partial model W_P^* of P:*
$SEM_{WFS}(P) := \{W_P^*\}$.

Remark: In the (van Gelder)-Definition of the well-founded semantics, W_P is not a function on the set of all three-valued interpretations, i.e. it is not well-defined. Indeed, there are three-valued interpretations I such that $W_P(I)$ is not three-valued (it becomes four-valued). However, this is not a serious problem because the iterates $W_P \uparrow \alpha$ are provably still always all three-valued. [52]

Definition 55. *(**Entailment**) A normal LP P entails a ground atom a under WFS, denoted by $P \models a$, if it is true in $SEM_{WFS}(P)$.*

WFS can be considered an approximation of stable models, i.e., if a program has stable models, then if an atom is true resp. false wrt the WFS then it is true resp. false wrt STABLE. [81] Moreover, for weakly stratified LPs [76] WFS coincides with STABLE. However, there are three important distinction between STABLE and WFS:

1. WFS is a three-valued semantics, whereas STABLE is two-valued.
2. every normal LP has exactly one WFS model, whereas every normal LP has zero or more stable models.
3. Irrelevant clauses (tautologies) lead to the non-existence of stable models, e.g., $p \leftarrow \neg p$ has no stable model.

While the alternating fixpoint on normal logic programs only captures the negation of positive existential closure such as e.g. transitive closure, it does not capture the negation of positive universal closure. As shown by van Gelder [101] the constructive characterization of the well-founded semantics for normal logic programs in terms of alternating fixpoint partial models can be further extended towards an alternating fixpoint semantics for general logic programs. There have been also several proposals for extending WFS by classical negation leading to a well-founded semantics for extended LPs - see e.g., [39,40,9,61,73,24]. Decidable and semi-decidable fragments of the WFS have been discussed in [32].

Procedural Semantics for Normal and Extended LPs. Existing procedural semantics for the computation of the well-founded model can be divided into two groups: (1) *bottom-up approaches* such as the alternating fixpoint approach [99,100,64], the magic set approach [89,55,68,95] and transformation based (aka residual program) approaches [25, 41, 22, 23] and (2) *top down approaches* such as *non-tabling based approaches* such as Global SLS resolution [78,88] or tabling based approaches such as extensions to OLDT resolution [96], e.g., WELL [14], XOLDTNF [29] or the approach of Bol and Degerstedt [16], SLT resolution [92] or the well-known SLG resolution [28] (another prominent extension of OLDT). There are also some proof procedures for well-founded semantics for extended logic programs (WFSX) such as [97] or SLX resolution [3].

The well-known 2-valued top-down SLDNF (classical LP Prolog) resolution [33], a resolution based method derived from SLD resolution [57,8], as a procedural semantics for LPs has many advantages. Due to its linear derivations it can be implemented using efficient stack based memory structures, it supports very useful sequential operators such as cut, denoted by !, or *assert/retract* and the negation-as-finite failure test is computationally quite efficient. Nevertheless, it is a too weak procedural semantics for unrestricted LPs with negations. It does not support goal memoization and suffers from well-known problems such as redundant computations of identical calls, non-terminating loops or floundering. It is not complete for LPs with negation or infinite functions. Moreover, it can not answer free variables in negative subgoals since the negation as finite failure rules is only a simple test. For more information on SLDNF resolution I refer to [62,6]. For typical unsolvable problems related to SLDNF see e.g. [94].

SLG resolution [30,28] is the most prominent tabling based top-down method for computing the well-founded semantics for normal LPs and it has been show in [31] how SLG can be used for query evaluation of general logic programs under WFS alternating fixpoint semantics. SLG resolution overcomes infinite loops and redundant computations by tabling. The basic idea of tabling, as implemented e.g., in ODLT resolution [96], is to answer calls (goals) with the memorized answers from earlier identical goals which are stored in a table. However, SLG

resolution is a non-linear approach. SLG is based on program transformations using six basic transformation rules, instead of the tree based approach of SLDNF. It distinguishes between solution nodes, which derive child nodes using the clauses from the program and look-up nodes, which produce child nodes using the memorized answers in the tables. Since all variant subgoals derive answers from the same solution node, SLG resolution essentially generates a search graph instead of a search tree and jumps back and forth between lookup and solution nodes, i.e., it is non-linear. Special delaying literals are used for temporarily undefined negative literals and a dependency graph is maintained to identify negative loops. Calls to look-up nodes will be suspended until all answers are collected in the table, in contrast to the linear SLD style where a new goal is always generated by linearly extending the latest goal. It is up to this non-linearity of SLG that tabled calls are not allowed to occur in the scope of sequential operators such as cut.

Global-SLS resolution [78, 79, 88] for WFS is a procedural semantics which directly extends SLDNF resolution and hence preserves the linearity property of SLDNF. In contrast to SLDNF-trees, SLS-trees treat infinite derivations as failed and recursions through negation as undefined. However, it assumes a positivistic computation rule that selects all positive literals before negative ones and inherits the problem of redundant computations from SLDNF. Moreover, a query fails if the SLS-trees for the goal either end at a failure leave or are infinite, which makes Global-SLS computationally ineffective [88]. To avoid redundant computations in SLS, a tabling approach called tabulated SLS resolution [15] was proposed. But the approach, like SLG, is based on non-linear tabling.

SLX resolution [3] is a procedural semantics for extended LPs which is sound and complete wrt WFSX semantics. As in SLS resolution it uses a failure rule to solve the problems of infinite positive recursions and distinguishes two kinds of derivations for proving verity (SLX-T tree) and proving non-falsity (SLX-TU tree) in the well-founded model in order to fail or succeed literals involved in recursion through negation. Thus, SLX does not consider a temporal undefined status as the other top-down approaches for WFS do, but implements the following derivations: if a goal L is to be undefined wrt WFS it must be failed, if it occurs in a SLX-T derivation and refuted if it occurs in a SLX-TU derivation. To fulfill the coherence requirement of WFSX a default negated literal $\sim L$ is removed from a goal if there is no SLX-TU refutation for L or if there is one SLX-T refutation for $\sim L$. In short, SLX is very close to SLDNF resolution. As already pointed out by the authors [3, 4] it is only theoretically complete, does not guarantee termination since it lacks loop detection mechanisms, is in general not efficient and makes redundant computations since tabling is not supported. Its implementation is given as a meta program in Prolog.

SLE resolution (Linear resolution with Selection function for Extended WFS) extends linear SLDNF with goal memoization based on linear tabling and loop cutting. In short, it resolves infinite loops and redundant computations by tabling without violating the linearity property of SLD style resolutions. SLE resolution is based on four truth values: t (true), f (false), u (undefined) and u' (temporarily undefined) with $t = \neg f$, $\neg f = t$, $\neg u = u$ and $\neg u' = u$ and a truth ordering

$\neg f > t > u > u'$. u' will be used if the truth value of a subgoal is temporarily undecided. SLE resolution follows SLDNF, where derivation trees are constructed by resolution. For more information on the notion of trees for describing the search space of top-down proof procedures see e.g. [62]. In SLE a node in a tree is defined by $N_i : G_i$, where N_i is the node name and G_i is the first goal labelling the node. Tables are used to store intermediate results. In contrast to SLG resolution, there is no distinction between lookup and solution nodes in SLE. The algorithm, always, first tries to answer the call (goal) with the memorized answers in the tables. If there are no answers available in a table the call is resolved against program clauses which are selected in the same top-down order as in SLDNF. This avoids redundant computations. To preserve the order the answers stored in a table are used in a FIFO (first-in-first-out) style, i.e., the first memorized answer is first used to answer the call. In case of loops the two main issues in top-down procedural semantics for WFS are solutions to infinite positive recursions (positive loops) and infinite recursion through negation by default (negative loops).

3 Web Rule Languages

Web rule languages have been developed for the declarative representation of, e.g., privacy policies, business rules, and Semantic Web rules. Rules are central to knowledge representation for the Semantic Web and are often considered as being side by side with ontologies, e.g. in W3C's hierarchical Semantic Web architecture (2007 version shown in Figure 2).

There are different types of rules which can be used on the Web such as

- *Derivation rules* are sentences of knowledge that are derived from other knowledge by an inference or mathematical calculation.
- *Reaction rules* are behavioral rules which react on occurred events or changed conditions by executing actions.
- *Integrity rules* (or constraints) are assertions which express conditions that must be always satisfied.
- *Deontic rules* describe rights and obligations of roles in the context of evolving states (situations triggered by events/actions) and state transitions.
- *Transformation rules* - specify term rewriting, which can be considered as derivation rules of logics with (oriented) equality
- *Facts* might describe various kinds of information such as events (event/action messages, event occurences), (object-oriented) object instances, class individuals (of ontology classes), norms, constraints, states (fluents), conditions of various forms, actions, data (e.g., relational, XML), etc., which might be qualified, e.g., by priorities, temporally, etc.

Rules can influence the operational and decision processes of Web systems.

- Derivation rules (deduction rules): establish / derive new information from existing Web data that is used, e.g. in a decision process.
- Reaction rules that establish when certain activities should take place:

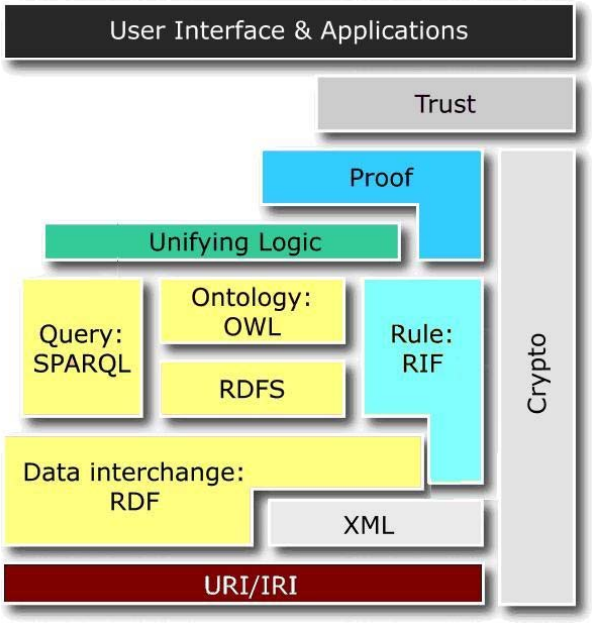

Fig. 2. Semantic Web Layer Cake [adapted from (W3C, 2007)]

- Condition-Action rules (production rules)
- Event-Condition-Action (ECA) rules + variants (e.g. ECAP)
- Messaging reaction rules (event message reaction rules)

Rules can also act as constraints on the Web systems structure, behavior and information.

– Structural constraints (e.g. deontic assignments).
– Integrity constraints and state constraints
– Process and flow constraints.

Web rule markup languages provide the required expressiveness enabling machine-interpretation, automated processing and translation into other such Web languages, some of which also being the execution syntaxes of rule engines. One of these languages may act as a lingua franca to interchange rules and integrate with other markup languages, in particular with Web languages based on XML and with Semantic Web languages (e.g. RDF Schema, OWL and OWL 2) for ontologies based on RDF or directly on XML. Web rule languages may also be used for publication purposes on the Web and for the serialization of external data sources, e.g. of native online XML databases or RDF stores. Recently, there have been several efforts aiming at rule interchange and building a general, practical, and deployable rule markup standard for the (Semantic) Web. These include several important general standardization or standards-proposing efforts including RuleML (`www.`

ruleml.org), the W3C member submission SWRL (www.w3.org/Submission/SWRL/), the W3C recommendation RIF (www.w3.org/2005/rules/), and others.

A complete specification of Web rule languages consists of a formalization of their *syntax*, *semantics* and, often left implicit, *pragmatics*. The syntax of Web rule markup languages always includes the concrete syntax of (XML) markup, perhaps indirectly through other languages such as via RDF/XML. Often, there is another more or less concrete syntax such as a compact shorthand or presentation syntax, which may be parsed into the XML markup. While a presentation syntax can already disregard certain details, an abstract syntax systematically replaces character sequences with abstract constructors, often in a (UML) diagram form or as an abstract syntax tree (AST). Together with different token dictionaries, it can be used to generate corresponding concrete syntaxes. The semantics is formalized in a model-theoretic, proof-theoretic, or procedural manner, sometimes in more than one. When rules and speech-act-like performatives, such as queries and answers, are transmitted between different systems, their pragmatic interpretation, including their pragmatic context, becomes relevant, e.g. in order to explain the effects of performatives - such as the assertion or retraction of facts - on the internal knowledge base [72].

A general distinction of three rule modeling layers can be adopted from OMG's model driven architecture (MDA) engineering approach (http://www.omg.org/mda/):

- A platform specific model (PSM) which encodes the rule statements in the language of a specific execution environment
- A platform independent model (PIM) which represents the rules in a common (standardized) interchange format, a rule markup language
- A computational independent model (CIM) with rules represented in a natural or visual language

The CIM level comprises visual and verbal rendering and rule modeling, e.g. via graphical representation or a controlled natural language syntax for rules, mainly intended for human consumption. Graphical representations such as UML diagrams or template-driven/controlled languages can also be used as presentation languages.

The PIM level should enable platform-independent machine interpretation, processing, interchange and translation into multiple PSM execution syntaxes of concrete rule engines. Hence, the concrete XML (or RDF/XML based) syntax of a Web rule language such as RuleML, SWRL or RIF resides on this level, whereas the abstract syntax is on the borderline between the PIM and CIM levels.

The PSM level is the result of translating/mapping PIM rule (interchange) languages into execution syntaxes, such as ISO Prolog, POSL, Prova (http://prova.ws/), which can be directly used in a specific execution environment such as a rule engine. A general distinction can be made between a compiled language approach, where the rules are statically translated into byte code (at compile time) versus interpreted scripting languages, which are dynamically interpreted (at run-time). While the compiled approach has obvious efficiency benefits, the interpreted approach is more dynamic and facilitates, e.g., updates at

run-time. Often, Semantic Web Rule Languages are directly executable by their respective rule engines; hence reside on the PSM level. As an intermediate step between the concrete PSM level and the PIM level an abstract representation is often introduced, such as N3, which provides an abstract rule syntax based on the RDF syntax.

The correct execution of an interchanged PIM-level rule set serialized in a rule markup language depends on the semantics of both the rule program and the platform-specific rule inference engine (IE). To address this issue, the IE and the interchanged rule set must reveal their intended/implemented semantics. This may be solved via explicit annotations based on a common vocabulary, e.g. an (Semantic Web) ontology which classifies the semantics. Annotations describing the semantics of an interchanged rule set could even be used to find appropriate IEs on the Web to correctly and efficiently interpret and execute the rule program; for example, (1) by configuring the rule engine for a particular semantics in case it supports different ones, (2) by executing an applicable variant of several interchanged semantic alternatives of the rule program, or (3) by automatic transformation approaches which transform the interchanged rule program into a rule program with an applicable semantics.

In the following two subsections languages on the PIM and PSM level will be described.

3.1 Platform Independent Web Rule Languages

In the following, three prominent platform independent Web Rule languages are introduced.

RuleML
The Rule Markup Language (RuleML, `www.ruleml.org`) is a markup language developed to express a family of Web rules in XML for deduction, rewriting, and reaction, as well as further inferential, transformational, and behavioral tasks. It is defined by the Rule Markup Initiative (`www.ruleml.org`), an open network of individuals and groups from both industry and academia that was formed to develop a canonical Web language for rules using XML markup and transformations from and to other rule standards/systems. It develops a modular, hierarchical specification for different types of rules comprising facts, queries, derivation rules, integrity constraints (consistency-maintenance rules), production rules, and reaction rules (Reaction RuleML, `http://reaction.ruleml.org`), as well as tools and transformations from and to other rule standards/systems. Datalog RuleML is defined over both data constants and individual constants with an optional attribute for IRI (URI) webizing. Atomic formulas have n arguments, which can be positional terms or, in Object-Oriented Datalog, slots (F-logic-like key→term pairs); OO Datalog also adds optional types and RDF-like oids/anchors, via IRIs (Boley, 2003). Inheriting all of these Datalog features, Hornlog RuleML adds positional or slotted functional expressions as terms. In Hornlog with equality, such uninterpreted (constructor-like) functions are complemented by interpreted (equation-defined) functions. This derivation rule branch is extended upward

towards First Order Logic, has subbranches with Negation-As-Failure, strong-Negation, or combined languages, and is parameterized by 'pluggable' built-ins.

SWRL

The Semantic Web Rule Language (SWRL, `www.w3.org/Submission/SWRL/`) is defined as a language combining sublanguages of the OWL Web Ontology Language (OWL DL and Lite) with those of the Rule Markup Language (Unary/Binary Datalog). The specification was submitted to W3C in May 2004 by the National Research Council of Canada, Network Inference (since acquired by webMethods), and Stanford University in association with the Joint US/EU ad hoc Agent Markup Language Committee. Compared to Description Logic Programs (DLP) [50], a slightly earlier proposal for integrating description logic and Horn rule formalisms by an overlapping authoring team, SWRL takes the opposite integration approach: DLP can be seen as the intersection of description logic and Horn logic; SWRL, as roughly their union. For DLP, the resulting rather inexpressive language corresponds to a peculiar looking description logic imitating special rules. It is hard to see the DLP restrictions, which stem from Lloyd-Topor transformations, being either natural or satisfying. On the other hand, SWRL retains the full power of OWL DL, but adds rules at the price of undecidability and a lack of complete implementations, although the SWRL Tab of Protege has become quite popular (`http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab`). Rules in SWRL are of the form of an implication between an antecedent (body) conjunction and a consequent (head) conjunction, where description logic expressions can occur on both sides. The intended interpretation is as in classical first-order logic: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold.

SWRL [53] is a homogeneous approach combining rules with ontoligies with relatively high complexity bounds for the ontology reasoning part (due to the fact that standard rule engines are not optimized for DL reasoning). In general, the works on combining rules and ontologies can be basically classified into two basic approaches: *homogeneous and heterogeneous integrations*. Starting from the early Krypthon language [20] among the heterogeneous approaches, which hybridly use DL reasoning techniques and tools in combination with rule languages and rule engines are e.g., CARIN [60], Life [2], Al-log [38], non-monotonic dl-programs [42] and r-hybrid KBs [85]. Among the homogeneous approaches which combine the rule component and the DL component in one homogeneous framework sharing the combined language symbols are e.g., DLP [50], KAON2 [69] or SWRL [53]. Both integration approaches have pros and cons and different integration strategies such as reductions or fixpoint iterations are applied with different restrictions to ensure decidability. These restrictions reach from the intersection of DLs and Horn rules [50] to leaving full syntactic freedom for the DL component, but restricting the rules to *DL-safe rules* [69], where DL variables must also occur in a non DL-atom in the rule body, or *role-safe rules* [60], where at least one variable in a binary DL-query in the body of a hybrid rule must also appear in a non-DL atom in the body of the rule which never appears in the consequent of any rule in the program or to tree-shaped rules [51]. Furthermore, they can be distinguished

according to their information flow which might be *uni-directional* or *bi-directional*. For instance, in homogeneous approaches bi-directional information flows between the rules and the ontology part are naturally supported and new DL constructs introduced in the rule heads can be directly used in the integrated ontology inferences, e.g., with the restriction that the variables also appear in the rule body (safeness condition). However, in these approaches the DL reasoning is typically solved completely by the rule engine and benefits of existing optimized DL reasoners using, e.g. variants of tableau based algorithms, are lost. On the other hand, heterogenous approaches, benefit from the hybrid use of both reasoning concepts exploiting the advantages of both (using LP reasoning and tableaux based DL reasoning), but bi-directional information flow and fresh DL constructs in rule heads are much more difficult to implement. A more complete survey and discussion of the combination of rules and ontologies is given in chapter "OWL and Rules" of this lecture book. [1]

W3C RIF

The W3C Rule Interchange Format (RIF) Working Group [86] is an effort, influenced by RuleML, to define a standard Rule Interchange Format for facilitating the exchange of rule sets among different systems and to facilitate the development of intelligent rule based applications for the Semantic Web. For these purposes, RIF Use Cases and Requirements (RIF-UCR) have been developed. The RIF architecture is conceived as a family of languages, called dialects. A RIF dialect is a rule based language with an XML syntax and a well-defined semantics.

The W3C RIF recommendation defines the Basic Logic Dialect (RIF-BLD), which corresponds to a definite Horn rule language with equality. RIF-BLD has a number of syntactic extensions with respect to 'regular' Horn rules, including internationalized resource identifiers (IRIs) as identifiers for concepts, F-logic-like frames and slots, and a standard system of built-ins drawn from Datatypes and Built-Ins (RIF-DTB). RIF Core (RIF-Core) in the intersection of RIF-BLD and the Production Rule Dialect (RIF-PRD) influenced by OMG's PRR, which can then be further extended or supplemented by reaction rules. The connection to other W3C Semantic Web languages is established via RDF and OWL Compatibility (RIF-SWC). Moreover, RIF-BLD is a general Web language in that it supports the use of IRIs (Internationalized Resource Identifiers) and XML Schema data types. The RIF Working Group has also defined the Framework for Logic Dialects (RIF-FLD). RIF-FLD uses a uniform notion of terms for both expressions and atoms in a higher order logic (HiLog)-like manner.

In the following, the syntax and semantics of the basic logic dialect of RIF will be summarized.

Definition 56. *(**Alphabet**): The alphabet of the non-normative presentation language of RIF-BLD, which maps to the normative XML syntax of RIF, consists of*

- *a countably infinite set of **constant symbols** Const*
- *a countably infinite set of **variable symbols** Var*
- *a countably infinite set of **argument/slot names**, Arg*
- ***connective symbols** And, Or, and* :−

- ***quantifiers*** *Forall and Exists*
- *the symbols* $\rightarrow$, *External*, $=$, #, ##, *Import*, *Prefix*, *and Base*
- *the symbols Group and Document*
- *the auxiliary symbols* (,), [,], <, >, *and* ^^.

Constants in RIF are written as *literal*^^*symbolspace*, where literal is a sequence of Unicode characters and symbolspace is an identifier for a symbol space consisting of an identifier and a lexical space. Symbol spaces supported in RIF are

- **identifiers of Web entities**, where the lexical space consists of strings that syntactically are internationalized resource identifiers (IRIs), e.g., `http://www.w3.org/2007/rif#iri`
- **datatypes** supported by RIF, e.g. `http://www.w3.org/2001/XMLSchema#integer`
- **rif:local** which is used for function and predicate symbols that are local to a rule document.

Slot/argument names *Arg* and variables *Var* are unicode strings. Variables start with the symbol ?, e.g. $?x$. The symbol $\rightarrow$ is used in terms that have named arguments and in frame formulas. Equality in RIF is denoted by $=$. The symbols #, and ## are used in formulas that define class membership and subclass relationships. The symbol *External* defines an external atomic formula or a function term defined by a RIF built-in. The symbol *Document* is used to specify RIF-BLD documents. The symbol *Import* is used for importing documents, and the symbol *Group* is used to organize RIF-BLD formulas into rule sets.

Using the above alphabet the language of RIF-BLD is constructed as a set of formulas. The main building blocks that are used to construct formulas are **terms**. RIF-BLD defines several kinds of terms: **constants** and **variables**, **positional** (as in normal logic programs) and **named-argument** (slotted) terms (as in F-Logic), and additionally **equality**, **membership**, **subclass**, **frame**, and **external** terms.

Positional terms of the form $p(v_1...v_n)$ and unpositional named arguments $p(s_1 \rightarrow v_1...s_n \rightarrow v_n)$, where p is a (webized) predicate symbol, are atomic formula. Equality, subclass, membership, and frame terms are atomic formulas, too. $External(\varphi)$, where φ is an atomic formula, is an externally defined atomic formula.

The *condition language* of RIF-BLD constructs condition formula from atomic formula using **Conjunction**: $And(\varphi_1...\varphi_n)$ to build a conjunctive formula, **Disjunction**: $Or(\varphi_1...\varphi_n)$, to build disjunctive formula, and **Existentials**: $Exists ?V_1...?V_n(\varphi)$ to build existential formula.

Condition formulas are used inside the premises of rules in the RIF *rule language* dialects (RIF Core, RIF BLD and RIF PRD). In RIF-BLD definite horn rules are defined as **rule implications**: $\psi :- \varphi$ which are universally quantified $Forall ?V_1...?V_n(\psi :- \varphi)$. A set of rules is grouped in a **group formula**: $Group(\varphi_1...\varphi_n)$, where φ_i is either a universal fact, variable-free rule implication, variable-free atomic formula, or another group formula. Finally, RIF-BLD **document formula** are expressions of the form: $Document(directive_1...directive_n \Pi)$,

where Π is an optional group formula (the knowledge base of the RIF document) and the optional directives are

- import directive of the form $Import(iri)$ or $Import(iriprofile)$, where iri indicates the location of another RIF document to be imported plus and optional $profile$ for import.
- base directives of the form $Base(iri)$ defining syntactic shortcuts for expanding relative IRIs into full IRIs
- prefix directive of the form $Prefix(pv)$ defining a syntactic shortcut to enable a compact URI representation for $rif : iri$ constants.

Additionally, RIF-BLD allows every term and formula to be optionally preceded by an annotation of the form $(* \; id \; \varphi \; *)$, where id is a $rif : iri$ constant and φ is a formula.

The non-normative presentation syntax corresponds to the normative XML syntax of RIF-BLD which uses the element and attribute names listed below:

- **And**: conjunction
- **Or**: disjunction
- **Exists**: quantified formula for existentials, containing declare and formula roles
- **declare**: declare role, containing a Var
- **formula**: formula role, containing a FORMULA
- **Atom**: atom formula, positional or with named arguments
- **External**: external call, containing a content role
- **content**: content role, containing an Atom, for predicates, or Expr, for functions
- **Member**: prefix version of member formula #
- **Subclass**: prefix version of subclass formula ##
- **Frame**: Frame formula
- **object**: Member/Frame role, containing a TERM or an object description
- **op**: Atom/Expr role for predicates/functions as operations
- **args**: Atom/Expr positional arguments role, with fixed 'ordered' attribute, containing n TERMs
- **instance**: Member instance role
- **class**: Member class role
- **super**: Subclass super-class role
- **sub**: Subclass sub-class role
- **slot**: prefix version of Name/TERM → TERM pair as an Atom/Expr or Frame slot role, with fixed 'ordered' attribute
- **Equal**: prefix version of term equation '='
- **Expr**: expression formula, positional or with named arguments
- **left**: Equal left-hand side role
- **right**: Equal right-hand side role
- **Const**: individual, function, or predicate symbol, with optional 'type' attribute
- **Name**: name of named argument
- **Var**: serialized version of logic '?' variable

- **id**: identifier role, containing IRICONST
- **meta**: meta role, containing metadata as a Frame or Frame conjunction
- **Document**: document, containing optional directive and payload roles
- **directive**: directive role, containing Import
- **payload**: payload role, containing Group
- **Import**: importation, containing location and optional profile
- **location**: location role, containing IRICONST
- **profile**: profile role, containing PROFILE
- **Group**: nested collection of sentences
- **sentence**: sentence role, containing RULE or Group
- **Forall**: quantified formula for 'Forall', containing declare and formula roles
- **Implies**: prefix version of logic ':-' implication, containing if and then roles
- **if**: antecedent role, containing FORMULA
- **then**: consequent role, containing ATOMIC or conjunction of ATOMICs

Like RuleML, the XML syntax of RIF divides all XML tags into class descriptors starting with upper case letters, called *type tags*, and property descriptors starting with lower case letters, called *role tags*. [17]

The semantics of RIF-BLD is an adaptation of the standard semantics for Horn clauses. It is specified using general models.

Definition 57. *(**Semantic Structure**) A semantic structure, I, is a tuple of the form ¡TV, DTS, D, D_{ind}, D_{func}, I_C, I_V, I_F, I_{frame}, I_{NF}, I_{sub}, I_{isa}, $I_{=}$, $I_{external}$, I_{truth}¿, where D is a non-empty set of elements called the domain of I, and D_{ind}, D_{func} are nonempty subsets of D. D_{ind} is used to interpret the elements of Const that play the role of individuals and D_{func} is used to interpret the constants that play the role of function symbols. DTS denotes a set of identifiers for primitive datatypes as defined in RIF-DTB. I_C maps Const to D. I_V maps Var to D_{ind}. I_F maps D to functions $D*_{ind} \rightarrow D$ with $D*_{ind}$ being a set of all finite sequences over the domain D_{ind}. I_{NF} maps D to the set of total functions $SetOfFiniteSets(ArgNames \times D_{ind}) \rightarrow D$, where $ArgNames$ are named arguments. I_{frame} maps D_{ind} to total functions of the form $SetOfFiniteBags(Dind \times Dind) \rightarrow D$. I_{sub} is a mapping of the form $D_{ind} \times D_{ind} \rightarrow D$. I_{isa} is a mapping of the form $D_{ind} \times D_{ind} \rightarrow D$. $I_{=}$ is a mapping of the form $D_{ind} \times D_{ind} \rightarrow D$. I_{truth} is a mapping of the form $D \rightarrow TV$. Finally, $I_{external}$ is a mapping of symbols into Const described as external to fixed n-ary functions.*

RIF-BLD also defines a generic mapping from terms to D as follows:

- $I(k) = I_C(k)$, if k is a symbol in $Const$
- $I(?v) = I_V(?v)$, if $?v$ is a variable in Var
- $I(f(t_1...t_n)) = I_F(I(f))(I(t_1), ..., I(t_n))$
- $I(f(s_1 \rightarrow v_1...s_n \rightarrow v_n)) = I_{NF}(I(f))(< s_1, I(v_1) >, ..., < s_n, I(v_n) >)$
- $I(o[a_1 \rightarrow v_1...a_k \rightarrow v_k]) = I_{frame}(I(o))(< I(a_1), I(v_1) >, ..., < I(a_n), I(v_n) >)$
  Note, that in RIF $I(o[a \rightarrow b\ a \rightarrow b]) = I(o[a \rightarrow b])$.
- $I(c_1\#\#c_2) = I_{sub}(I(c_1), I(c_2))$
- $I(o\#c) = I_{isa}(I(o), I(c))$

- $I(x = y) = I_{=}(I(x), I(y))$
- $I(External(p(s_1...s_n))) = I_{external}(p)(I(s_1), ..., I(s_n))$.

The truth value of (non-document) formulas in RIF BLD is determined from the semantic structures by the following truth valuation.

Definition 58. *(**Truth Valuation**)The truth valuation $TVal_I$ is defined as follows:*

- *Positional atomic formulas:* $TVal_I(r(t_1...t_n)) = I_{truth}(I(r(t_1...t_n)))$
- *Atomic formulas with named arguments:* $TVal_I(p(s_1 \rightarrow v_1...s_k \rightarrow v_k)) = I_{truth}(I(p(s_1 \rightarrow v_1...s_k \rightarrow v_k)))$
- *Equality:* $TVal_I(x = y) = I_{truth}(I(x = y))$ *with* $I_{truth}(I(x = y)) = t$ *if* $I(x) = I(y)$ *and that* $I_{truth}(I(x = y)) = f$ *otherwise*
- *Subclass:* $TVal_I(sc\#\#cl) = I_{truth}(I(sc\#\#cl))$
- *Membership:* $TVal_I(o\#cl) = I_{truth}(I(o\#cl))$
- *Frame:* $TVal_I(o[a_1 \rightarrow v_1...a_k \rightarrow v_k]) = I_{truth}(I(o[a_1 \rightarrow v_1...a_k \rightarrow v_k]))$
- *Externally defined atomic formula:* $TVal_I(External(t)) = I_{truth}(I_{external}(t))$
- *Conjunction:* $TVal_I(And(c_1...c_n)) = t$ *if and only if* $TVal_I(c_1) = ... = TVal_I(c_n) = t$. *Otherwise,* $TVal_I(And(c_1...c_n)) = f$.
- *Disjunction:* $TVal_I(Or(c_1...c_n)) = f$ *if and only if* $TVal_I(c_1) = ... = TVal_I(c_n) = f$. *Otherwise,* $TVal_I(Or(c_1...c_n)) = t$.
- *Quantification:*
  - $TVal_I(Exists?v_1...?v_n(\varphi)) = t$ *if and only if for some* $TVal_{I*}(\varphi) = t$
  - $TVal_I(Forall?v_1...?v_n(\varphi)) = t$ *if and only if* $TVal_{I*}(\varphi) = t$, *where* $I*$ *is a semantic structure with the special mapping* $I*_V$ *which coincides with* I_V *on all variables except, possibly, on* $?v_1, ..., ?v_n$.
- *Rule implication:*
  - $TVal_I(conclusion{:}\text{-}condition) = t$, *if either* $TVal_I(conclusion) = t$ *or* $TVal_I(condition) = f$.
  - $TVal_I(conclusion{:}\text{-}condition) = f$ *otherwise.*
- *Groups of rules: If* Π *is a group formula of the form* $Group(\varphi_1...\varphi_n)$ *then*
  - $TVal_I(\Pi) = t$ *if and only if* $TVal_I(\varphi_1) = t, ..., TVal_I(\varphi_n) = t$.
  - $TVal_I(\Pi) = f$ *otherwise.*

Since RIF allows to import other documents which can have $rif : local$ constants, semantic multi-structures are introduced for the interpretation of documents. Semantic multi-structures are essentially similar to regular semantic structures, as defined above, but, in addition, they allow to interpret $rif : local$ symbols that belong to different documents differently.

The following logical entailment defines what it means for a set of RIF-BLD rules to entail another RIF-BLD formula, in particular entailment of RIF condition formulas.

Definition 59. *(**Models**) A multi-structure I is a model of a formula, φ, written as $I \models \varphi$, iff $TVal_I(\varphi) = t$.*

Definition 60. *(**Logical Entailment**) Let φ and ψ be formulas, then φ entails ψ, written as $\varphi \models \psi$, if and only if for every multi-structure I for which both $TVal_I(\varphi)$ and $TVal_I(\psi)$ are defined, $I \models \varphi$ implies $I \models \psi$.*

For a more detailed account of RIF and RIF BLD we refer to the W3C recommendation [86] and [19].

The following subsection 3.3 exemplifies how platform independent rule languages such as RuleML and RIF are mapped into platform specific rule languages such as Prova.

3.2 Prova - A Platform Specific Web Rule Language

Prova (`http://www.prova.ws/`) is both a (Semantic) Web rule language and a highly expressive distributed (Semantic) Web rule engine which supports complex reaction rule based workflows, rule based complex event processing, distributed inference services, rule interchange, rule based decision logic, dynamic access to external data sources, Web Services, and Java APIs. Prova follows the spirit and design principles of the W3C Semantic Web initiative and combines declarative rules, ontologies and inference with dynamic object-oriented programming and access to external data sources via query languages such as SQL, SPARQL, and XQuery. One of the key advantages of Prova is its separation of logic, data access, and computation as well as its tight integration of Java, Semantic Web technologies and enterprise service-oriented computing and complex event processing technologies.

Semantically Prova provides the expressiveness of serial Horn logic with a linear resolution for extended logic programs (SLE resolution) and with several extra logical features which will be described in the following subsequent subsections. Syntactically Prova builds on top of the ISO Prolog syntax (ISO Prolog ISO/IEC 13211-1:1995), but it extends it syntactically and semantically. The following diagram 3 gives an overview on the Prova 3 language structure and its main language elements.

The basic syntactic structures of the Prova language are rules (head :- body), facts (rule heads with no body), and goals (rules with no head). Prova supports atomic terms (constants and variables) and complex terms (functions internally represented as lists). Constants in Prova can be simple strings starting with lower case letters (e.g. const) or text in single or double quotes (e.g. "Constant 1"), numeric data (e.g. 12, -300L), as well as fully qualified static or instance fields in Java objects (e.g. java.lang.Double(1.3)) or (Description Logic) individuals of ontology concepts (e.g. 10^^math:Percentage). Variables start with upper case letters (e.g. X). They can be typed (e.g. Integer.X) and assume the type of the assigned constant (e.g. $X = 1$). Like in Prolog anonymous variables begin with underscore (_). Special global variables and constants have names starting with '$' (e.g. $Counter). Complex terms in Prova are functions which can be equally represented as generic lists, where the first head element is the function operator and the list tail are its arguments, e.g. $f(X, Y)$ can be equally represented as $[f, X, Y]$ or $[f|R]$ (which binds the list tail to the variable R). Prova supports positional literals as in Prolog, e.g. $p(arg_1, ..., arg_n)$ as well as unpositional

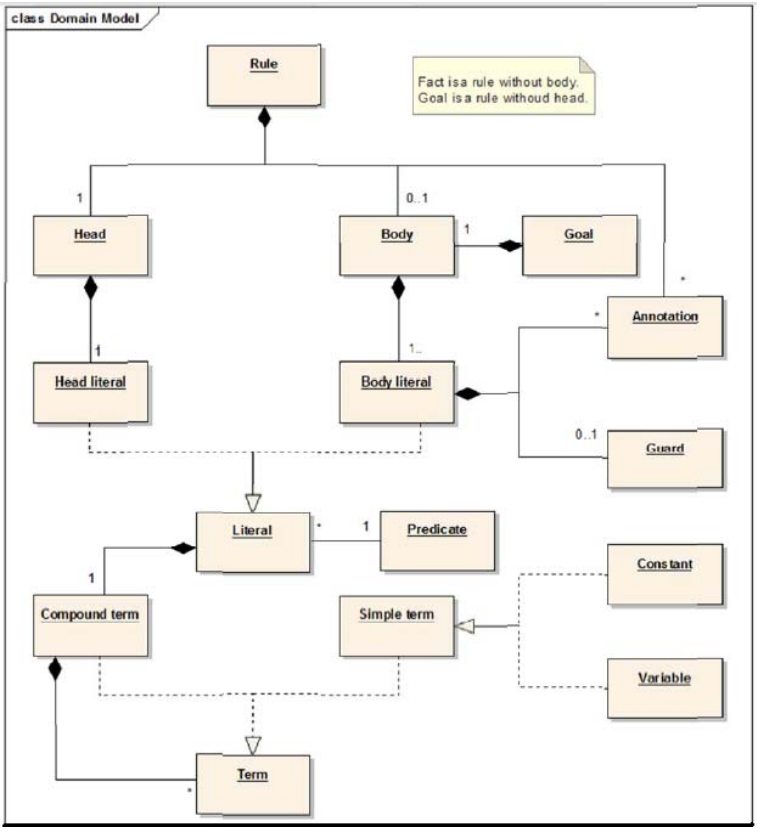

Fig. 3. Main Prova 3 Language Elements

slotted literals, as in slotted and object-oriented logics such as F-Logic, e.g. $p(slot_1 -> arg_1, ..., slot_n -> arg_n)$. In the following subsection we show how RIF and RuleML syntactically maps to Prova. Prova distinguishes between for all quantified *solve* goals and existential *eval* goals. For solve goals, for all successful inference all assignments for the variables in the goal predicate satisfying the query are handed back. For eval goals, the engine executes an exhaustive existential search of the rules and facts until no more backtracking is possible. Additionally, the built-in meta-predicate *derive* allows to define (sub) goals dynamically with the predicate symbol unknown until run-time, e.g. $p(F) :- derive([F|Args])$. where the variable F is assigned the function name at runtime.

3.3 Mapping from RIF to RuleML and Prova

This section by means of examples shows how RIF can be mapped into RuleML and Prova (Prolog). These examples largely correspond to the partial mappings defined for Datalog RuleML and the RIF-Core subset of RIF BLD [18]. While RIF only supports neutral constants (Const), RuleML supports specialized constant

RIF	RuleML	Prova
<Const type="&xs;string"> ABC </Const>	<Data xsi:type="xs:string"> ABC </Data>	"ABC"
<Var>x</Var>	<Var>x</Var>	X
<Expr> <op> <Const type="&rif;iri"> &func;f </Const> </op> <args ordered="yes"> <Var>X</Var> </args> </Expr>	<Expr> <Fun iri="func:f" per="value"/> <Var>X</Var> </Expr>	func:f(X)
<Atom> <op> <Const type="&rif;iri"> &cpt;discount </Const> </op> <args ordered="yes"> <Var>cust</Var> <Var>prod</Var> <Var>val</Var> </args> </Atom>	<Atom> <Rel iri="cpt:discount"/> <Var>cust</Var> <Var>prod</Var> <Var>val</Var> </Atom>	cpt:discount(Cust,Prod,Val)
<Equal> <left> <Var>X</Var> </left> <right> <Var>Y</Var> </right> </Equal>	<Equal oriented="yes"> <Var>X</Var> <Var>Y</Var> </Equal>	X=Y
<Member> <instance> <Const type="&rif;iri"> &ppl;Adrian </Const> </instance> <class> <Const type="&rif;iri"> &ppl;Person </Const> </class> </Member>	<Ind iri="ppl:Adrian" type="ppl:Person"/>	ppl:Adrian^^ppl:Person
<External> <content> <Expr> <op> <Const type="&rif;iri"> &rifb;numeric-add </Const> </op> <args ordered="yes"> <Const type="&xs;integer"> 1 </Const> <Const type="&xs;integer"> 1 </Const> </args> </Expr> </content> </External>	<Expr> <Fun iri="rifb:numeric-add" per="value"/> <Data xsi:type="xs:integer"> 1 </Data> <Data xsi:type="xs:integer"> 1 </Data> </Expr>	1+1
<Atom> <op> <Const type="&rif;iri"> &ex;gold </Const> </op> <slot> <Name>customer</Name> <Var>Customer</Var> </slot> </Atom>	<Atom> <Rel iri="ex:gold"/> <slot> <Ind>customer</Ind> <Var>Customer</Var> </slot </Atom>	ex:gold({customer->Customer})

terms which distinguish data constants (Data) from individuals/instances object constants (Ind). RIF does not support a multi-sorted logic with type definitions as in RuleML (@type attribute). The special member built-in in RIF (Member) can be used to define instances of classes which can be interpreted as an explicit type definition. External functions (built-ins) in RIF are restricted to the predefined RIF datatypes and built-ins (DTB) library which can be reused in RuleML together with other external built-in libraries (e.g. from SWRL, XPath etc.). Unpositional named arguments as well as positional arguments are supported by both RIF and RuleML and can be mapped into positional terms in Prova like in Prolog standard logic programs or into unpositional slotted terms.

In the following some of the extra logical extensions of Prova will be introduced.

3.4 Access to External Data, Type Systems and Procedural Attachments

Prova follows the spirit and design of the W3C Semantic Web initiative and combines declarative rules, ontologies and inference with dynamic object-oriented programming and access to external data sources and type systems. Therefore, Prova assumes not just a single universe of discourse, but several domains, so called sorts (types) which are interpreted in a multi-sorted logic. The extension of the signature and the typed variables of the language alphabet with sorts (aka types) is defined as follows.

Definition 61. *(**Multi-sorted Signature**) The multi-sorted signature S of Prova is defined as a tuple $\langle \overline{T}, \overline{P}, \overline{F}, arity, \overline{c}, sort \rangle$ where:*

1. *$\overline{P}$ is a finite sequence of predicate symbols $\langle P_1, .., P_n \rangle$.*
2. *$\overline{F}$ is a finite sequence of function symbols $\langle F_1, .., F_m \rangle$*
3. *For each P_i respectively each F_j, $arity(P_i)$ resp. $arity(F_j)$ is a non-zero natural number denoting the arity of P_i resp. F_i.*
4. *$\overline{c} = \langle c1, .., c_o \rangle$ is a finite or infinite sequence of constant symbols,*
5. *and, $\overline{T} = \{T_1, .., T_n\}$ is a set of sort/type symbols called sorts.*

The function sort associates with each predicate, function or constant its sorts:

- *if c is a constant, then $sort(c)$ returns the type T of c.*
- *if p is a predicate of arity k, then $sort(p)$ is a k-tuple of sorts $sort(p) = (T_1, .., T_k)$ where each term t_i of p is of some type T_j, i.e., $t_i : T_j$.*
- *if f is a function of arity k, then $sort(f)$ is a $k+1$-tuple of sorts $sort(f) = (T_1, .., T_k, T_{k+1})$ where $(T_1, .., T_k)$ defines the sorts of the domain of f and T_{k+1} defines the sorts of the range of f*

Prova supports the following three basic types of sorts

1. *primitive sorts are given as a fixed set of primitive data types such as integer, string, etc.*

2. *function sorts are complex sorts constructed from primitive sorts* $T_1 \times ... \times T_n \rightarrow T_{n+1}$ *and other complex sorts defined in the external type alphabet*
3. *Boolean sorts are a (predicate) statement of the form* $T_1 \times ... \times T_n$

Definition 62. *(**Multi-sorted Logic**) Prova's multi-sorted logic associates which each term, predicate and function a particular sort:*

1. *Any constant or variable* t *is a term and its sort* T *is given by* $sort(t)$
2. *Let* $f(t_1, .., t_n)$ *be a function then it is a term of sort* T_{n+1} *if* $sort(f) = \langle T_1, .., T_n, T_{n+1} \rangle$*, i.e.,* f *takes argument of sort* $T_1, .., T_n$ *and returns arguments in sort* T_{n+1}*.*

The intuitive meaning is that a predicate or function holds only if each of its terms is of the respective sort given by *sort*.

The alphabet of the Prova language builds on top of the standard ISO Prolog syntax standard, but further extends it. For typing each variable X_j in the multi-sorted alphabet of the Prova language is associated with a specific sort $sort(X_j) = T_i$, written as $X_j : T_i$, where X_j is a variable and T_i is a type sort associated with the variable. That is, the extended Prova language considers external sort/type alphabets. The combined signatures of the Prova rule language and the external type languages form the basis for combined hybrid knowledge bases and the integration of external type systems into the rule system.

Definition 63. *(**Type alphabet**) An external type alphabet* $\overline{T}$ *is a finite set of monomorphic sort/type symbols built over the distinct set of terminological class concepts of a (external type) language.*

Definition 64. *(**Combined Signature**) A combined signature* $\overline{S}$ *is the union of all its constituent finite signatures:* $\overline{S} = \langle S_1 \cup .. \cup S_n \rangle$

The type systems considered in Prova are order-sorted (i.e., with sub-type relations):

Definition 65. *(**Order-sorted Type System**) A finite order-sorted type system* TS *comes with a partial order* $\leq$*, i.e.,* TS *under* $\leq$ *has a greatest lower bound* $glb(T_1, T_2)$ *for any two types* T_1 *and* T_2 *having a lower bound at all. Since* TS *is finite also a least upper bound* $lub(T_1, T_2)$ *exists for any two types* T_1 *and* T_2 *having an upper bound at all.*

Definition 66. *(**Combined Knowledge Base**) The combined knowledge base of a typed Prova* $\overline{KB} = \langle \Phi, \Psi \rangle$ *consists of a finite set of (order-sorted) type systems / type knowledge bases* $\Psi = \{\Psi_1 \cap .. \cap \Psi_n\}$ *and a typed Prova KB* Φ*.*

The combined signature is the union of all constituent signatures, i.e., each interpretation of a Prova rule program has the set of ground terms of the combined signature as its fixed universe.

Definition 67. *(**Extended Herbrand Base**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a typed combined Prova KB P. The extended Herbrand base of P, denoted $\overline{B}(P)$, is the set of all ground literals which can be formed by using the predicate/function symbols in the combined signature with the ground typed terms in the combined universe $\overline{U}(P)$, which is the set of all ground typed terms which can be formed out of the constants, type and function symbols of the combined signature.*

The grounding of the combined KB is computed wrt the composite signature.

Definition 68. *(**Grounding**) Let P be a typed (combined) Prova KB and $\overline{c}$ its set of constant symbols in the combined signature. The grounding ground(P) consists of all ground instances of all rules in P w.r.t to the combined multi-sorted signature which can be obtained as follows:*

- *The ground instantiation of a rule r is the collection of all formulas $r[X_1 : T_1/t_1, .., X_n : T_n/t_n]$ with $X_1, .., X_n$ denoting the variables and $T_1, .., T_n$ the types of the variables (which must not necessarily be disjoint) which occur in r and $t_1, .., t_n$ ranging over all constants in $\overline{c}$ wrt to their types.*
- *For every explicit query/goal $Q[X_1 : T_1, .., X_m : T_m]$ to the type system, being either a fact with one or more free typed variables $X_1 : T_1, .., X_m : T_m$ or a special built-in Prova query literal rdf(...) with variables as arguments in the triple-like query, the grounding ground(Q) is an instantiation of all variables with constants (individuals) in $\overline{c}$ according to their types.*

Using equalities Prova assumes a notion of default inequality for the combined set of individuals/constants which leads to a default unique name assumption:

Definition 69. *(**Default Unique Name Assumption**) Two ground terms are assumed to be unequal, unless equality between the terms can be derived.*

The interpretation I of a typed Prova program P then is a subset of the extended Herbrand base $\overline{B}(P)$.

Definition 70. *(**Multi-sorted Interpretation**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a combined KB and $\overline{c}$ its set of constant symbols. An interpretation I for a multi-sorted combined signature $\overline{S}$ consists of*

1. *a universe $|M| = T_1^I \cup T_2^I \cup .. \cup T_n^I$, which is the union of the types (sorts), and*
2. *the predicates, function symbols and constansts/individuals $\overline{c}$ in the combined signature, which are interpreted in accordance with their types.*

The assignment function σ from the set of variable $\overline{X}$ of P into the combined universe $\overline{U}(P)$ must respect the sorts/types of the variables (in order-sorted type systems also subtypes). That is, if X_i is a variable of type T, then $\sigma(X) \in T^I$. In general, if ϕ is a typed predicate or function in Φ and σ an assignment to the interpretation I, then $I \models \phi[\sigma]$, i.e., ϕ is true in I when each variable X of ϕ is substituted by the values $\sigma(X)$ wrt to its type. Since the assignment to constant and function symbols is fixed and the domain of discourse corresponds one-to-one with the constants $\overline{c}$ in the combined signature $\overline{U}(P)$, it is possible to identify an interpretation I with a subset of the extended Herbrand base: $I \subseteq \overline{B}(P)$.

The assignment function in Prova is given as a query from the rule component to the type system, so that there is a separation between the inferences in a type system and the rule component. Moreover, explicit queries to a type system (Java or Semantic Web) defined in the body of a rule, e.g., procedural attachments, built-ins or ontology queries (special *rdf* query or free DL-typed facts) are based on this hybrid query mechanism.

Definition 71. ***(Semantic Multi-Structure Model)*** *Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a combined KB of a typed Prova program P.*
An interpretation I is a model of an untyped ground atom $A \in \overline{KB}$ or I satisfies A, denoted $I \models A$ iff $A \in I$.
I is a model for a ground typed atom $A : T \in \overline{KB}$, or I satisfies $A : T$, denoted $I \models A : T$, iff $A : T \in I$ and for every typed term $t_i : T_j$ in A the type query $T_j = sort(t_i)$, denoting the type check "is t_i of type T_i", is entailed in $\overline{KB}$, i.e., $\overline{KB} \models T_i = sort(t_i)$ (note, in an order sorted type system subtypes are considered, i.e., t_i is of the same or a subtype of T_j).
I is an interpretation of an ground explicit query/goal Q to the type system Ψ if $\Psi \models Q$.
I is a model of a ground rule $r : H \leftarrow B$ iff $I \models H(r)$ whenever $I \models B(r)$. I is a model a typed program P (resp. a combined knowledge base $\overline{KB}$), denoted by $I \models P$, if $I \models r$ for all $r \in ground(P)$.

Informally, a typed Prova knowledge base consists of rules with logic programming literals which have typed terms and a set of external (order-sorted) type systems in which the types (sorts) are defined over their type alphabets. An external type system might possibly define a complete knowledge base with types/sorts (Java classes or T-Box in DL) and individuals associated with these types (Java object instances of the classes or A-box in DL). Restricted built-in predicates and procedural attachment predicates or functions which construct or return individuals of a certain type (boolean or object-valued) are also considered to be part of the external type system(s), i.e., part of the external signature. The combined signature is then the union of the two (or more) signatures, i.e., the combination of the signature of the rule component and the signatures of the external type systems / knowledge bases combining their type alphabets, their functions and predicates and their individuals.

The operational semantics of typed Prova is implemented as hybrid polymorphic order-sorted unification. [71] In contrast to other hybrid (DL-typing) approaches which apply additional constraint literals as type guards in the rule body and leave the usual machinery of resolution and unification unchanged, the operational semantics for prescriptive types in Prova's typed logic is implemented by an order-sorted unification. Here the specific computations that are performed in the typed language are intimately related to the types attached to the atomic term symbols. The order-sorted unification yields the term of the two sorts (types) in the given sort hierarchy. This ensures that type checks apply directly during typed unification of terms at runtime enabling ad-hoc polymorphism of variables leading e.g., to different optimized rule variants and early constrained search trees. Thus,

the order-sorted mechanism provides higher level of abstraction, providing more compact and complete solutions and avoiding possibly expensive backtracking.

Prova provides support for two external order-sorted type systems, namely *Java* class hierarchies and ontological type systems (e.g. OWL or RDFS ontologies) respectively *Description Logic* knowledge bases.

Description Logic Type Systems / Ontologies. An external type systems supported by Prova are Semantic Web ontologies (Description Logic KBs) represented e.g. in RDFS or OWL. That is, the combined signature $\overline{S}_{DL}$ consisting of the finite signature S of the rule component and the finite signature(s) S_i of the ontology language(s).

The type alphabet TS is a finite set of monomorphic type symbols built over the distinct set of terminological atomic concepts $\overline{T}$ in a Semantic Web ontology language Σ^{DL}, i.e., defined by the atomic classes in the T-Box model.

Note, that restricting types to atomic concepts is not a real restriction, because for any complex concept such as $(T_1 \sqcap T_2)$ or $(T_1 \sqcup T_2)$ one may introduce an atomic concept T_3 in the T-Box and use T_3 as atomic type instead of the complex concept. This approach is also reasonable from a practical point of view since dynamic type checking must be computationally efficient in order to be usable in an order-sorted typed logic with possible very large rule derivation trees and many typed unification steps, i.e., fast type checks are crucial during typed term unification. We assume that the type alphabet is fixed (but arbitrary), i.e., no new terminological concepts can be introduced in the T-Box by the rules at runtime. This ensure completeness of the domain and enables static type checking on the used DL-types in Prova programs at compile time (during parsing the Prova script).

The set of constants/individuals $\overline{c}$ is built over the set of individual names in Σ^{DL}, but Prova do not fix the constant names and allow arbitrary fresh constants (individuals) (under default UNA) to be introduced in the head of rules and facts of the rule base. However, new individuals which are introduced in rules or facts apply locally within the scope of the rules in which they are defined, i.e., within a local reasoning chain; in contrast to the individuals defined in the A-box model of the type system which apply globally as individuals of a class. DL-typed terms in Prova are defined as follows:

Definition 72. *(**DL-typed Terms**) A DL-type is a terminological concept/class defined in the DL-type system (T-Box model). A typed DL-typed Prova term is denoted by the relation $t^{\wedge\wedge}T$ where t is the term and T is the DL-type of term.*

The type ontologies are typically provided as Web ontologies (RDFS or OWL) where types and individuals are represented as resources having an webized URI. Namespaces can be used to avoid name conflicts and namespace abbreviations facilitate are more readable language.

```
% A customer gets 10 percent discount, if the customer is a gold customer

discount(X^^business:Customer, 10^^math:Percentage) :-
   gold(X^^business:Customer).

% fact with free typed variable acts as instance query on the ontology A-box
gold(X^^business:Customer).
```

Free DL-typed variables are allowed in facts. They act as free instance queries on the ontology layer, i.e., they query all individuals of the given type and bind them to the typed variable.

Java Type System, Procedural Attachments and Built-Ins. For external Java type systems, the combined multi-sorted signature $\overline{S}_{Java}$ uses the fully qualified order-sorted Java class hierarchy as type symbols. In order to type a variable with a Java type the fully qualified name of the Java class to which the variable should belong must be specified as a prefix separated from the variable by a dot ".".

```
java.lang.Integer.X       variable X is of type Integer
java.util.Calendar.T      variable T is of type Calendar
java.sql.Types.STRUCT.S    variable S is of SQL type Struct
```

To sense the environment and trigger actions, query data from external sources such as databases, call external procedural code such as Enterprise Java Beans, and receive / send messages from / to other agents or external services, Prova provides a set of built-in functions and additionally can dynamically instantiate any Java object and call its API methods at runtime.

Java objects, as instances of Java classes, can be dynamically constructed by calling their constructors or static methods using extra logical procedural attachments. The returned objects, might then be used as individuals/constants that are bound by an equality relation (denoting typed unification equality) to appropriate variables, i.e., the variables must be of the same type or of a super type of the Java object.

A procedural attachment is a function that is implemented by an external procedure (i.e., a Java method). They are used in Prova to dynamically call external procedural methods during runtime, i.e., they enable the (re)use of procedural code and allow dynamic access to external data sources and tools using their programming interfaces (APIs). They are a crucial extension to traditional logic programming, combining the benefits of object-oriented languages (Java) with declarative rule based programming, e.g., in order to externalize mathematical computations such as aggregations to highly optimized procedural code in Java or use query languages such as SQL by JDBC to select and aggregate facts from external data sources.

Definition 73. *(**Procedural Attachments**) A procedural attachment is a function or predicate whose implementation is given by an external procedure. Two types of procedural attachments are distinguished:*

- ***Boolean-valued attachments** (or **predicate attachments**) which call methods which return a Boolean value, i.e., which are of Boolean sort (type).*
- ***Object-valued attachments** (or **functional attachments**) which are treated as functions that take arguments and return one or more objects, i.e., which are of a function sort.*

Functional Java attachments have a left-hand side with which the results (the returned object(s)) of the call are unified by a unification equality relation =,

e.g., $C = java.util.Calendar.getInstance()$. If the left-hand side is a free (unassigned) variable the latter stores the result of the invocation. If the left-hand side is a bound variable or a list pattern the unification can succeed or fail according to the typed unification and consequently the call itself can succeed or fail. List structures are used on the left-hand side to allow matching of sets of constructed/returned objects to specified list patterns. A predicate attachment is assumed to be a test in such a way that the call succeeds only if a true Boolean variable is returned. Static, instance and constructor calls are supported in both predicate and functional attachments depending on their return type. Constructor calls follow the Java syntax with the fully qualified name of the class and the constructor arguments, e.g., $X = java.lang.Long(123)$. Static method calls require fully qualified class names to appear before the name of the static method followed by arguments, e.g., $Z = java.lang.Math.min(X, Y)$. Instance methods are mapped to concrete classes dynamically based on the type of the variable, i.e., the method of a previously bound Java object is called. They require a variable before the name of an instance method followed by the arguments, e.g., $S = X.toString()$.

```
add(java.lang.Integer.In1,java.lang.Integer.In2,Result):-
  Result = java.lang.Integer.In1 + java.lang.Integer.In2.
```

The first rule takes two Integer variables $In1$ and $In2$ as input and returns the result which is bound to the untyped variable $Result$. Accordingly, a query $add(1, 1, Result)?$ succeeds with an Integer object 2 bound to the $Result$ variable, while a query $add("abc", "def", Result)?$ will fail.

It is important to note, that Java objects can be bound to variables and their methods can be dynamically used as procedural attachment functions anywhere during the reasoning process, i.e., in other rules. This enables a tight and highly expressive integration of external object oriented functions into declarative agent's rules' execution.

Definition 74. ***(Built-in Predicates or Functions)*** *Built-in predicates or functions (built-ins) are special restricted procedural attachment predicate resp. function symbols in the Prova language for concrete domains, e.g., integers or strings, that may occur in the body of a rules.*

Examples are $+$, $=$, $assert$, $bound$, $free$ etc. For instance, Prova provides a rich library of built-ins for query languages such as SQL, SPARQL, and XQuery:

```
File Input / Output
   ..., fopen(File,Reader), ...
XML (DOM)
   document(DomTree,DocumentReader) :-  XML(DocumenReader),...
SQL
   ... ,sql_select(DB,cla,[pdb_id,"1alx"],[px,Domain]).
RDF
   ...,rdf(http://...,"rdfs",Subject,"rdf_type","gene1_Gene"),...
XQuery
 ..., XQuery = 'for $name in StatisticsURL//Author[0]/@name/text()
   return $name', xquery_select(XQuery,name(ExpertName)),...
SPARQL
   ...,sparql_select(SparqlQuery,...
```

The following rule uses a SPARQL query built-in to access an RDF Friend-of-a-Friend (FOAF) profile published on the Web. The selected data is assigned to variables which can be used within an agent's rule logic, e.g. to expose the agent's contact data.

```
exampleSPARQLQuery(URL,Type) :-
 QueryString = ' PREFIX foaf:
             PREFIX rdf:
             SELECT ?contributor ?url ?type
             FROM
             WHERE {
                   ?contributor foaf:name "Bob DuCharme" .
                   ?contributor foaf:weblog ?url .
                   ?contributor rdf:type ?type . } ',
 sparql_select(QueryString,url(URL),type(Type)).
```

Note, that the structures in Java type systems are usually not considered as interpretations in the strict model-theoretic definition, but are composite structures involving several different structures whose elements have a certain inner composition. However, transformations of composite structures into their flat model theoretic presentations is in the majority of cases possible. From a practical point of view, it is convenient to neglect the inner composition of the elements of the universe of a structure. These elements are just considered as "abstract" points devoid of any inherent meaning. This structural mapping between objects from their interpretations in the Java universe to their interpretation in the rule system ignoring finer-grained differences that might arise from the respective definitions is given by the following isomorphism.

Definition 75. *(Isomorphism) Let I_1, I_2 be two interpretations of the combined signature $\overline{S} = \{T_1, .., T_n\}$, then $f_{\cong} : |M_1| \rightarrow |M_2|$ is an isomorphism of I_1 and I_2 if $f_{\cong}$ is a one-to-one mapping from the universe $|M_1|$ of I_1 onto the universe $|M_2|$ of I_2 such that:*

1. *For every type T_i, $t \in T_i^{I_1}$, iff $f_{\cong}(t) \in T_i^{I_2}$*
2. *For every constant c, $f_{\cong}(c^{I_1}) \cong c^{I_2}$*
3. *For every n-ary predicate symbol p with n-tuple $t_1, .., t_n \in |M_1|$, $\langle t_1, .., t_n \rangle \in p^{I_1}$ iff $\langle f_{\cong}(t_1), .., f_{\cong}(t_n) \rangle \in p^{I_2}$*
4. *For every n-ary function symbol f with n-tuple $t_1, .., t_n, \in |M_1|$, $f_{\cong}(f^{I_1}(t_1, .., t_n)) \cong f^{I_2}(f_{\cong}(t_1), .., f_{\cong}(t_n))$*

For instance, in Prova an isomorphism between Boolean Java objects and their model-theoretic truth value is defined, which makes it possible to treat boolean-valued procedural attachments as conditional body literals in rules and establish a model-theoretic interpreation as defined above between the Java type system and the model-theoretic semantics of the typed logic of the rule component. Other examples are String objects which are treated as standard constants in rules, i.e., the Java String object maps with the untyped theory of logic programming. Primitive datatype values, from the ontology resp. XML domain (XSD datatypes) can be mapped similarly.

3.5 Modularization, Scopes and Guards

To capture the distributed, open structure of Web based rule bases and enable scoped queries on explicitly closed parts of open and distributed knowledge, Prova supports principles of information hiding and modularization, which makes it easier to maintain and manage (distributed) rule sets.

Metadata Based Modularization and Module Imports/Updates. Prova has a flexible approach towards modularization of the knowledge base which allows constructing metadata based views on the knowledge base, so called *scopes*. Therefore, Prova extends the rule language to a labelled logic programming rule language (LLP) with metadata annotations such as rule labels, module (rule sets in rule bases) labels and arbitrary other (Semantic Web) annotations (e.g., Dublin Core author, date etc). These metadata annotations are used to manage the rules and facts in the knowledge base.

In analogy to the multi-sorted extension for types, the meta-data extension of the Prova language is defined over a combined signature $\overline{S}$ which is the union of the signature of the rule language and the signatures of the used metadata vocabularies (e.g. Dublin Core).

Definition 76. *(**Combined Signature with Metadata Annotations**) The combined metadata annotated signature $\overline{S}$ is defined as a tuple $\langle \overline{T}, \overline{P}, \overline{F}, arity, \overline{c}, sort, meta \rangle$ where $\overline{P}$ is the union of the predicate symbols define in the signature of the core Prova rule language and the metadata predicate symbols (denoting metadata key properties) defined in the signature(s) of the metadata vocabularie(s) and $\overline{c}$ is the union of constant symbols defined in the rule signature and in the metadata signature(s) (denoting metadata values). meta is a special unary function which returns the assigned metadata.*

To explicitly annotate clauses in a Prova program P with an additional set of metadata labels a general 1-ary built-in function @ is introduced in the Prova language.

Definition 77. *(**Metadata Annotation Labels**) The special 1-ary built-in function @ is a partial injective labelling function that assigns a set of metadata annotations m (property-value pairs) to a clause cl in P, e.g.*

$@(L_1, .., L_n)\ H :- B$

where L_i are a finite set of unary positive literals (positive metadata literals) which denote an arbitrary metadata property(value) pair, e.g., $@label(rule1)$.

The implicit form $@(L_1), .., @(L_n)\ H :- B$ of the metadata function expresses that $@(H :- B) = L_1, .., L_n$. The explicit @() annotation is optional, i.e., a Prova program P without metadata annotated clauses coincides with a standard unlabelled logic program.

Clauses in Prova are treated as objects in KB having an unique *object id* (*oid*) which might be user-defined, i.e., explicitly defined by a metadata annotation $@label(oid)\ H :- B$ or system-defined i.e., all rules are automatically "labelled"

with an auto-incremented *oid* (an increasing natural number) provided by the system at compile time. Rules and facts might be bundled to clause sets, so called *modules*, which also have an object id, the module oid. By default the module oid is the URI or full document name of the Prova script which contains the module. But the module oid might also be user-defined $@src(moduleoid)$. All clauses (rules and facts) defined in a module are automatically annotated with the module oid $@src(moduleoid)\ H :- B$. The oids are used to manage the knowledge in the (distributed) knowledge base, e.g., to import a rule set from an URI which is then used as the module oid or remove a module from the KB by its oid. Beside oids arbitrary other semantic annotations such as Dublin Core data might be specified in the @ annotation function.

```
@label(r1) @dc:author("Adrian") @dc:date(2006-11-12)
   p(X):-q(X).
@label(f1)
   q(1).
```

The example shows a rule with rule label $r1$ and two additional Dublin Core annotations $dc : author("Adrian")$ and $dc : date(2006-11-12)$ and a fact with fact label $f1$. Since there is no explicitly user-defined module oid in the meta-data labels, the default module oid for both clauses is the URI or document name of the Prova script in which they are defined, e.g. $@src("http://prova.ws/example1.prova")$.

In Prova it is possible to consult (import/load) distributed rulebases from local files, a Web address, or from incoming messages transporting a rulebase. Furthermore, Prova supports update built-ins such as *assert* and *retract*.

```
%load from a local file
:- eval(consult("organization2009.prova")).
% import from a Web address
:- eval(consult("http://ruleml.org/organization2010.prova")).
```

The imported rulebases are managed as modules in the knowledge base, which are uniquely identified by their source object id $src(moduleOID)$. Since multiple nested imports are possible, modules might be nested, i.e. a module denoting a rule base (e.g. a Prova script) might consist of several nested submodules (e.g. sets of rules and facts).

Similar to imports of external type systems and built-ins (procedural attachments) which query and compute external data, the semantics for modules in Prova is defined over the combined knowledge base of the modules, an extended state based Herbrand Base and semantic multi-structures.

Definition 78. ***(Combined Knowledge Base)*** *The combined knowledge base of a modular Prova* $\overline{KB} = \langle \Phi, \Psi \rangle$ *consists of a finite set of modules* $\Psi = \{\Psi_1 \cap .. \cap \Psi_n\}$ *and an initial primary Prova KB* Φ.

Prova supports knowledge updates which import modules (*consult*) and add or remove clauses (*assert*, *retract*). Each update leads to a new knowledge state of the combined KB.

Definition 79. *(**Knowledge State**) A knowledge state represents the combined knowledge base KB_k at this particular state, where $k \in \aleph$.*

Note that according to the modularized logic in Prova a state, i.e., a combined knowledge base KB_k, might consist of nested submodules, each having an unique ID (the module oid). Intuitively, a state represents the union of all clauses stored in all modules in the combined knowledge base.

An update is then a transition which adds or removes facts and/or rules and changes the knowledge base. That is, the KB transits from the initial state KB_1 to a new state KB_2. We define the following notion of positive (assert) and negative(retract) transition:

Definition 80. *(**Positive Update Transition**) A positive update transition, or simply positive update, to a knowledge state KB_k is defined as a finite set $U^{pos}_{oid} := \{r_N : H : -B, fact_M : A\}$ with A an atom denoting a fact, $H : -B$ a rule, $N = 0, .., n$ and $M = 0, ..m$ and oid being the update oid which is also used as module oid to manage the knowledge as a new module in the KB. Applying U^{pos}_{oid} to KB_k leads to the extended state $KB_{k+1} = \{KB_k \cup U^{pos}_{oid}\}$. Applying several positive updates as an increasing finite sequence $U^{pos}_{oid_j}$ with $j = 0, .., k$ and $U^{pos}_{oid_0} := \emptyset$ to KB_0 leads to a state $KB_k = \{KB_0 \cup U^{pos}_{oid_0} \cup U^{pos}_{oid_1} \cup ... \cup U^{pos}_{oid_k}\}$.*

That is a state KB_k is decomposable in the previous knowledge state $k-1$ plus the update: $KB_k = \{KB_{k-1} \cup U^{pos}_k\}$. We define $KB_0 = \{\emptyset \cup U^{pos}_{oid_0}\}$ and $U^{pos}_{oid_0} = \{KB$: the set of rules and facts defined in the program $P\}$, i.e., importing the initial Prova program P from a Prova script document is the first update leading to the knowledge state KB_1.

Likewise, We define a *negative update transition* as follows:

Definition 81. *(**Negative Update Transition**) A negative update transition, or for short a negative update, to a knowledge state KB_k is a finite set $U^{neg}_{oid} := \{r_N : H : -B, fact_M : A\}$ with $A \in KB_k$, $H : -B \in P$, $N = 0, .., n$ and $M = 0, ..m$, which is removed from KB_k, leading to the reduced program $KB_{k+1} = \{KB_k \setminus U^{neg}_{oid}\}$.*

Applying arbitrary sequences of positive and negative updates leads to a sequence of KB states $KB_0, .., KB_k$ where each state KB_i is defined by either $KB_i = KB_{i-1} \cup U^{pos}_{oid_i}$ or $KB_i = KB_{i-1} \setminus U^{neg}_{oid_i}$. In other words, KB_i, i.e., the set of all clauses in the KB at a particular knowledge state i, is decomposable in the previous knowledge state plus/minus an update, whereas the previous state consists of the state $i-2$ plus/minus an update and so on. Hence, each particular knowledge state can be decomposed in the initial state KB_0 and a sequence of updates. Although an update might insert more than one rule or fact, i.e., insert or remove a complete module, it nevertheless is treated as an elementary update, a so called bulk update, which transits the current knowledge state to the next state in an elementary transition: $\langle KB_i, U^{pos/neg}_{oid}, KB_{k+1}\rangle$. Intuitively, one might think of it as a complex atomic update action which performs all knowledge inserts resp. removes simultaneously.

Elementary updates have both a truth value, i.e. they may succeed or fail, and a side effect on the knowledge base leading to the transition of the knowledge state. The extended Herbrand Base is defined on the notion of knowledge states and transitions from one state to another.

Definition 82. *(**Extended State based Herbrand Base**) Let P be the combined KB at a particular knowledge state KB_k. The extended Herbrand base of P, denoted $\overline{B}(P)$, is the set of all ground literals which can be formed by using the predicate/function symbols in the combined signature with the ground typed terms in the combined universe $\overline{U}(P)$, which is the set of all ground typed terms which can be formed out of the constants, type and function symbols of the combined signature of KB_k.*

Definition 83. *(**Modular semantic multi-structure**) A modular multi-structure I is model of a modular program P (resp. the knowledge state KB_k of the combined knowledge base $\overline{KB}$), denoted by $I \models P$, if $I \models c$ for all clauses $c \in ground(P)$, where $I \models c$ is a usual multi-sorted model for providing the interpretation of Prova clauses.*

Accordingly, all queries to a Prova program apply on the extended resp. reduced transition knowledge state of the program, i.e., the truth valuation of a goal G depends on its model at the current knowledge state KB_k, denoted by $TVal_{KB_k \models G}(G)$.

Based on this modular knowledge state transition semantics and the metadata based control of the knowledge state updates which are treated as modules in the combined KB, Prova provides supports for transactional updates, where failing sequences of knowledge updates can be rolled back by removing the associated modules from the combined Prova KB. In the non-transactional style updates in (serial) Prova rules are not rolled-back to the original state if the derivation fails and the system backtracks. Typically this "weak" non-transactional semantics is intended when external Prova script are imported (consult) or new rule sets are added (assert) as modules. That is, independently, of whether the particular derivation in which the update is performed fails from some reason the update transition to the next knowledge state subsists and is not rolled back in case of failures.

Scoped Reasoning. The metadata annotation of rules/facts and rule sets (modules) enables scoped (meta) reasoning with the semantic annotations. The metadata can act as an explicit scope for constructive queries (creating a view) on the knowledge base. For instance, the metadata annotations might be used to constrain the level of generality of a scoped goal literal to a particular module, i.e., to consider only the set of rules and facts which belong to the specified module.

Definition 84. *(**Scoped Literal**) A scoped literal is of the form $@\overline{C}\ L$ where L is a positive or negative literal and $@\overline{C}$ is the scope definition which is a set of one or more metadata constraints. Scoped literals are only allowed in the body of a rule.*

Informally, the semantics of scoped literals allows to explicitly close the domain of discourse to certain parts of the KB.

Definition 85. *(**Metadata based Scope**) Let $\overline{KB}$ be a combined KB consisting of a set of submodules $\overline{KB} = \{KB_1 \cup .. \cup KB_k\}$. The scope KB' of a scoped literal $@\overline{C}\ L$ is the set of clauses $KB' = \{m'_1 cl_1, .., m'_n cl_n\} \in \overline{KB}$, where for all clauses $cl_i(m'_i) \in \overline{KB'}$ its set of metadata annotations m'_i satisfy the scope constraints $\overline{C}$ of the scoped literal L, i.e., $m'_i \models \overline{C}$.*

Accordingly, a scope (aka constructiv view) is constructed by one or more metadata constraints, e.g., the module oid $@src(URI/Filename)$ or Dublin Core values $@dc : author(...)$.

Definition 86. *(**Closure**) Let $\overline{KB}$ be a combined KB. The closure of $\overline{KB}$, denoted $Cl(\overline{KB})$, is defined by $\overline{KB}$ plus all modules KB_k which are in the scope of any scoped literal in $\overline{KB}$.*

A scoped literal $@\overline{C}\ L$ is closed if each rule in $\overline{KB}$ which unifies with the literal L is also closed, i.e., its body literals are closed in $Cl(\overline{KB})$.

Intuitively, this means that the closure of a Prova program depends on the scopes of the literals in the bodies of its rules. Obviously, if one of the subsequently used goal literals in a proof attempt is open, i.e., without a scope, the closure expands to the open KB.

Definition 87. *(**Scoped Semantics**) Given a scoped $\overline{KB}'$, where all literals are scoped with closure $Cl(\overline{KB}')$, the truth value of a scoped literal $@\overline{C}\ L$ depends on the partial model of the clauses of $\overline{KB}'$ wrt the scope definition $\overline{C}$, i.e., $I_{partial^{\overline{C}}}(\overline{KB}') \models L$.*

Syntactically the scope definitions use the syntax of Prova metadata annotations.

```
@label(rule1) r1(X):-q(X).
@label(rule2) r2(X):-q(X).
@label(rule3) p1(X):-
            @label(rule1) r1(X). % scoped goal literal
q(1).

:-solve(p1(Y)).
```

The example shows three metadata annotated rules. They query $p1(Y)$ will return only one solution with $Y = 1$, since the subgoal $r1(X)$ of $rule3$ applies only in the scope of the rule with label $rule1$, but not on $rule1$ and $rule2$, which would be the case if there would be no scope constraint defined for the subgoal.

Prova allows variables in the scope definitions which are bound to the annotated metadata values. The following example shows the definition of a scope, that constraints the application of the subgoal $r2(X)$ on the rule with label $rule3$ and on the module with source name $AgentRole1.prova$.

```
% get module label
r1(X,Y):-
   @src(Y) @label(rule3)
   r2(X).
:-solve(r1(X,"AgentRole1.prova")).
```

Guards. In addition to scopes Prova supports literal *guards* which act as additional pre-condition constraints.

Guards in Prova are syntactically specified in the Prova rule language using brackets after the goal literal. The model-theoretic semantics of guards is like for goal literals, however in the proof-theoretic semantics guards act like pre-conditions before the proofs of the standard goal literals starts.

For instance, the following rule makes decisions on the basis of rules which haven been authored by different persons and only applies those rules from trusted authors.

```
%simplified decision rules of an agent
@author(dev22) r2(X):-q(X).
@author(dev32) r2(X):-s(X).
q(2).
s(-2).

% for simplicity this is a fact, but could be also a complex rule
% which computes the trust value from the reputation value of dev22
trusted(dev22).

% Author dev22 is trusted but dev32 is not, so one solution is found: X=2
p1(X):-
 @author(A)
 r2(X) [trusted(A)].

% for all query
:-solve(p1(X1)).
```

This example uses metadata annotations on rules for the head literals $r2/1$ and a scopes on the literal $r2(X)$ in the body of the rule for $p1(X)$. Since variable A in $@author(A)$ is initially free, it gets instantiated from the matching target rule(s). Once A is instantiated to the target rule's $@author$ annotation's value ($dev22$, for the first $r2$ rule), the body of the target rule is dynamically non-destructively modified to include all the literals in the additional guard $trusted(A)$ before the body start, after which the processing continues. Since $trusted(dev22)$ is true but $trusted(dev32)$ is not, only the first rule for predicate r2 is used and so one solution $X1 = 2$ is returned by $solve(p1(X1))$.

3.6 Prova Serial Horn Rules for Messaging

For communication between distributed rule agents Prova supports special built-ins for asynchronously sending and receiving event messages within serial Horn rules. The main language constructs of messaging reaction rules are: *sendMsg* predicates to send messages, reaction *rcvMsg* rules which react to inbound messages, and *rcvMsg* or *rcvMult* inline reactions in the body of messaging reaction rules to receive one or more context-dependent multiple inbound event messages:

```
sendMsg(XID,Protocol,Agent,Performative,Payload |Context)
rcvMsg(XID,Protocol,From,Performative,Paylod|Context)
rcvMult(XID,Protocol,From,Performative,Paylod|Context)
```

Here, *XID* is the conversation identifier (conversation-id) of the conversation to which the message will belong. *Protocol* defines the communication protocol.

Agent denotes the target party of the message. *Performative* describes the pragmatic envelope for the message content. A standard nomenclature of performatives is, e.g., the FIPA Agents Communication Language (ACL). *Payload* represents the message content sent in the message envelope. It can be a specific query or answer or a complex interchanged rule base (set of rules and facts). For instance, the following rule snippet shows how a query is sent to an agent via an Enterprise Service Bus (esb) and then an answer is received from this agent.

```
...
sendMsg(Sub_CID,esb,Agent,acl:query-ref, Query),
rcvMsg(Sub_CID,esb,Agent,acl:inform-ref, Answer),
...
```

Interchanged messages besides the conversation's metadata and payload also carry the pragmatic context of the conversation such as communicative situations / acts, mentalistic notions, organizational and individual norms, purposes or individual goals and values. The payload of incoming event messages is interpreted with respect to the local conversation state, which is denoted by the conversation id, and the pragmatic context, which is given by a pragmatic performative. For instance, a standard nomenclature of pragmatic performatives, which can be integrated as external (semantic) vocabulary/ontology, is e.g., defined by the Knowledge Query Manipulation Language (KQML) (Finin et al. 1993), by the FIPA Agent Communication Language (ACL), which gives several speech act theory based communicative acts, or by the Standard Deontic Logic (SDL) with its normative concepts for obligations, permissions, and prohibitions. Depending on the pragmatic context, the message payload is used, e.g. to update the internal knowledge of the agent (e.g., add new facts or rulebases), add new tasks (goals), or detect a complex event pattern (from the internal event instance sequence). For instance, the following example shows a reaction rule that sends a complete rule base, which is loaded from a local *File* to an agent service *Remote* using JMS as transport protocol.

Example 2

```
% Upload a rule base read from File to the host
% at address Remote via JMS
upload_mobile_code(Remote,File) :-
   % Opening a file returns an instance
   % of java.io.BufferedReader in Reader
   fopen(File,Reader),
   Writer = java.io.StringWriter(),
   copy(Reader,Writer),
   Text = Writer.toString(),
   % variable SB will encapsulate the whole content of File
   SB = StringBuffer(Text),
   % send the complete rule base to the receiver agent "Remote"
   sendMsg(XID,jms,Remote,acl:inform,consult(SB)).
```

The corresponding receiving reaction rule of the remote agent is:

```
% wait for incoming messages with pragmatic context $acl:inform$
rcvMsg(XID,jms,Sender,acl:inform,[Predicate|Args]):-
   % derive the message payload, i.e. consult the received rule set to the internal KB
   derive([Predicate|Args]).
```

This rule receives incoming JMS based messages with the pragmatic context $acl : inform$ and derives the message content, i.e. consults the received rule base to the local knowledge base of the remote agent. It is important to note that via the conversation id several reaction rule reasoning processes might run in parallel, local to their conversation flows. Inactive reactions (conversation partitions) are removed from the system, e.g. by timeouts. Self-activations by sending a message to the receiver "self" are possible. With the pragmatic performatives it is possible to implement different coordination and negotiation protocols. For instance, if an agent does not understand the semantics of the interchanged message payload, it can inform the sender about this, using, e.g., the $acl : not - understood$ performative, so that the sender can additionally send the semantic information, e.g. a pointer to the ontology that defines the concepts of the payload, and the receiving agent can import this ontology to its internal knowledge base.

By using messaging reaction rules a Prova rule engine can be deployed as a distributed rule inference service, e.g. in the Rule Responder agent architecture [72], or e.g. as an OSGI component enabling massive parallelization of Prova agent nodes in grid/cloud environments and (smart) devices (e.g. RFID networks) which communicate via event messages.

4 Conclusion

Rule based systems have been investigated comprehensively in the realms of declarative logic programming and expert systems in the past decades. Logic programming has been a very popular paradigm and one of the most successful representatives of declarative programming in general. It is based on solid and well-understood theoretical concepts and has been proven to be very useful for rapid prototyping and describing problems on a high abstraction level. In recent years rule based technologies have experienced a remarkable come back namely in two areas: business rules processing, and reasoning in the context of the (Semantic) Web. The first trend is caused by the need to accelerate the slow and expensive software development life cycle. The vision of treating application logic as declarative business rules is particularly interesting for businesses with rapidly changing business logic. The second trend is related to the Semantic Web initiative of the W3C. The vision is that intelligent Semantic Web agents with their rule-based decision and reaction logic are capable of processing the cross referenced, machine processable knowledge on the Web in a platform independent manner. They are able to infer new knowledge and make intelligent, possibly pro-active and self-autonomous decisions and reactions. Emerging standards for rules operating in the context of the Semantic Web include RuleML (and SWRL) and the new W3C RIF recommendation.

A general rule markup language such as RuleML or RIF covers many different rule types and rule families. Their syntax builds on well establish Web data representation standards such as XML, RDF, URIs/IRIs etc. Some of the language families such as classical production rules historically only define an operational semantics, while other rule families such as logical rules are based on a model-theoretic and/or proof-theoretic semantics. An open research question is whether

there exists a unifying semantic framework for all different rule types. Work in this direction is pursued, e.g. in the RIF Framework for Logic Dialects (RIF FLD) and in Reaction RuleML for reaction rule and complex event processing semantics. However, since there is no general consensus on one particular semantics for all expressive rule languages, an exclusive commitment to one particular semantics for a Web rule language should be avoided (even in well-researched fields such as logic programming several semantics such as well-founded semantics and answer set semantics are competing). Nevertheless, for certain subfamilies a preferred semantics can still be given and semantic mappings between rule families be defined.

Another crucial extension to the classical theory of rule-based logic programming in modern Web rule engines such as Prova is that they include practical language constructs which might not (yet) have a standard formal semantics based on classical model-theoretic logic. For instance, procedural calls to external (object) functions, operational systems, data sources and terminological descriptions, are often vital to deal with practical real-world settings of distributed Web applications. Recent research, e.g. in Prova, is done on adopting such practical language constructs without a standard formal semantics but with a non-standard extra logical one which allows for a hybrid knowledge representation. Further examples of useful practical constructs are the annotation of rules and rule sets with additional metadata such as rule qualifications, rule names, module names, Dublin Core annotations, etc., which eases, e.g., the modularization of rules into rule sets (bundling of rules), the creation of constructive views over internal and external knowledge (scoped reasoning), as well as the publication and interchange of rules / rule sets on the Web (rule messaging). Advanced rule qualifications such as validity periods or rule priorities might for example safeguard dynamic updates (e.g. the incorporation of interchanged rules into the existing rule base), where conflicts are resolved by rule prioritizations. Although these extra logical features have no direct formalization in first order logic, the benefits for a practical rule-based Web system, which needs to cope with large problem sizes and which needs to efficiently interoperate with existing systems and data sources on the Web, prevail. The hybrid KR design which allows the integration of external vocabulary types, methods and data into rule execution combines the benefits of declarative and imperativ (object-oriented) programming and helps to overcome typical problems of declarative programming, e.g., wrt to computational efficiency of certain tasks. While there is a risk that these concessions to non-standard semantics might endanger the benefits of formal semantics for the overall rule language, they turn out to be a crucial means to avoid limitations of standard rule representations. The rule component will rarely run in isolation, but interact with various external components, hence call for functionalities such as efficient object-oriented, relational/SQL-style, and RDF data retrieval and aggregation methods that are common in modern Web information systems.

Another domain of research is the engineering and maintenance of large rule-based applications, where the rules are serialized and managed in a distributed manner, and are interchanged across domain boundaries. This calls for support of verification, validation and integrity testing (V&V&I), e.g., by test cases that

are written in the same rule markup language and are stored and interchanged together with the rule program [70].

Acknowledgements. This work has been partially supported by the "Inno Profile-Corporate Semantic Web" project funded by the German Federal Ministry of Education and Research (BMBF) and the BMBF Innovation Initiative for the New German Länder - Entrepreneurial Regions.

References

1. Krisnadhi, F.M.A.A., Hitzler, P.: Owl and rules. In: 7th International Summer School 2011 - Tutorial Lectures. LNCS, Springer, Heidelberg (2011)
2. Ait-Kaci, H., Podelski, A.: Towards the meaning of life. In: Małuszyński, J., Wirsing, M. (eds.) PLILP 1991. LNCS, vol. 528, pp. 255–274. Springer, Heidelberg (1991)
3. Alferes, J., Damasio, C., Pereira, L.M.: Slx: a top-down derivation procedure for programs with explicit negation. In: Bruynooghe, M. (ed.) International Logic Programming Symp., pp. 424–439 (1994)
4. Alferes, J.J., Damasio, C., Pereira, L.M.: A logic programming system for nonmonotonic reasoning. J. of Automated Reasoning 14(1), 93–147 (1995)
5. Apt, K.: Logic programming. In: Leeuwen, J.v. (ed.) Handbook of Theoretical Computer Science, vol. B, ch. 10, pp. 493–574. Elsevier, Amsterdam (1990)
6. Apt, K., Blair, H.: Logic Programming and Negation: A Survey. J. of Logic Programming 19(20), 9–71 (1994)
7. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In: Minker, J. (ed.) Foundations of Deductive Databases, pp. 89–148. Morgan Kaufmann, San Francisco (1988)
8. Apt, K., Emden, M.H.: Contributions to the theory of logic programming. J. of ACM 29(3), 841–862 (1982)
9. Baral, C., Gelfond, M.: Logic programming and knowledge representation. J. of Logic Programming 19, 20, 73–148 (1994)
10. Baral, C., Lobo, J., Minker, J.: Generalized well-founded semantics for logic programs. In: Stickel, M.E. (ed.) International Conference on Automated Deduction. Springer, Heidelberg (1990)
11. Baral, C., Lobo, J., Minker, J.: Generalized disjunctive well-founded semantics for logic programs. Annals of Math and Artificial Intelligence 11(5), 89–132 (1992)
12. Baral, C., Subrahmanian, V.S.: Dualities between alternative semantics for logic programming and non-monotonic reasoning. In: Int. Workshop of Logic Programming and Non-Monotonic Reasoning, pp. 69–86. MIT Press, Cambridge (1991)
13. Beeri, C., Ramakrishnan, R.: On the power of magic. The Journal of Logic Programming 10, 255–299 (1991)
14. Bidoit, N., Legay, P.: Well!: An evaluation procedure for all logic programs. In: Int. Conf. on Database Theory, pp. 335–348 (1990)
15. Bol, R.: Tabulated resolution for the well-founded semantics. Journal of Logic Programming 34(2), 67–109 (1998)
16. Bol, R., Degerstedt, L.: Tabulated resolution for well founded semantics. In: Intl. Logic Programming Symposium (1993)

17. Boley, H.: Object-oriented ruleML: User-level roles, URI-grounded clauses, and order-sorted terms. In: Schröder, M., Wagner, G. (eds.) RuleML 2003. LNCS, vol. 2876, pp. 1–16. Springer, Heidelberg (2003)
18. Boley, H.: RIF RuleML Rosetta Ring: Round-Tripping the Dlex Subset of Datalog RuleML and RIF-Core. In: Governatori, G., Hall, J., Paschke, A. (eds.) RuleML 2009. LNCS, vol. 5858, pp. 29–42. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-04985-9
19. Boley, H., Kifer, M.: A guide to the basic logic dialect for rule interchange on the web. IEEE Trans. on Knowl. and Data Eng. 22, 1593–1608 (2010)
20. Brachman, R.J., Gilbert, P.V., Levesque, H.J.: An essential hybrid reasoning system: Knowledge and symbol level accounts for krypton. In: Int. Conf. on Artificial Inelligence (1985)
21. Brass, S., Dix, J.: Characterizations of the disjunctive wellfounded semantics: Confluent calculi and iterated gcwa. Journal of Automated Reasoning (1997)
22. Brass, S., Dix, J.: Characterizations of the disjunctive well-founded semantics. Journal of Logic Programming 34(2), 67–109 (1998)
23. Brass, S., Dix, J., Zukowski, U.: Transformation based bottom-up computation of the well-founded model. Theory and Practice of Logic Programming 1(5), 497–538 (2001)
24. Brewka, G.: Well-founded semantics for extended logic programs with dynamic preferences. Journal of Artificial Intelligence Research 4, 19–36 (1996)
25. Bry, F.: Negation in logic programming: A formalization in constructive logic. In: Karagiannis, D. (ed.) IS/KI 1990 and KI-WS 1990. LNCS, vol. 474, pp. 30–46. Springer, Heidelberg (1991)
26. Bry, F.: Query evaluation in recursive databases: bottom-up and top-down reconciled. Data and Knowlege Engineering 5, 289–312 (1990)
27. Chen, J., Kundu, S.: The strong semantics for logic programs. In: Proceedings of the 6th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC (1991)
28. Chen, W., Swift, T., Warren, D.S.: Efficient top-down computation of queries under the well-founded semantics. J. of Logic Programming 24(3), 161–199 (1995)
29. Chen, W., Warren, D.S.: A goal-oriented approach to computing well-founded semantics. In: Intl. Conf. and Symposium on Logic Programming (1992)
30. Chen, W., Warren, D.S.: Query evaluation under the well-founded semantics. In: Proceedings of Symp. on the Principles of Database Systems (1993)
31. Chen, W.: Query evaluation in deductive databases with alternating fixpoint semantics. ACM Transactions on Database Systems 20, 239–287 (1995)
32. Cherchago, N., Hitzler, P., Hölldobler, S.: Decidability under the well-founded semantics. In: Marchiori, M., Pan, J.Z., Marie, C.d.S. (eds.) RR 2007. LNCS, vol. 4524, pp. 269–278. Springer, Heidelberg (2007)
33. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and DataBases, New York, pp. 293–322 (1978)
34. Dix, J.: A framework for representing and characterizing semantics of logic programs. In: Nebel, B., Rich, C., Swartout, W. (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR 1992), pp. 591–602. Morgan Kaufmann, San Mateo (1992)
35. Dix, J.: A classification-theory of semantics of normal logic programs: Ii. weak properties. Fundamenta Informaticae XXII(3), 257–288 (1995)
36. Dix, J.: Semantics of logic programs: Their intuitions and formal properties. an overview. In: Fuhrmann, A., Rott, H. (eds.) Essays on Logic in Philosophy and Artificial Intelligence, pp. 241–327. DeGruyter, Berlag-New York (1995)

37. Doets, K.: From Logic to Logic Programming. MIT Press, Camebridge (1994)
38. Donini, F.M., Lenzerini, M., Nardi, D., Schaerf, A.: A hybrid system with datalog and concept languages. In: Ardizzone, E., Sorbello, F., Gaglio, S. (eds.) AI*IA 1991. LNCS (LNAI), vol. 549, pp. 88–97. Springer, Heidelberg (1991)
39. Dung, P.M.: Negation as hypotheses: An abductive foundation for logic programming. In: 8th Int. Conf. on Logic Programming, MIT Press, Cambridge (1991)
40. Dung, P.M.: An argumentation semantics for logic programming with explicit negation. In: 10th Logic Programming Conf., MIT Press, Cambridge (1993)
41. Dung, P.M., Kanchansut, K.: A natural semantics of logic programs with negation. In: 9th Conf. on Foundations of Software Technology and Theoretical Computer Science, pp. 70–80 (1989)
42. Eiter, T., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. In: KR 2004 (2004)
43. Emden, M.H., Kowalski, R.: The semantics of predicate logic as a programming language. JACM 23, 733–742 (1976)
44. Fitting, M.: A kripke-kleene semantics of logic programs. Journal of Logic Programming 4, 295–312 (1985)
45. Fitting, M.: Well-founded semantics, generalized. In: Int. Symposium of Logic Programming, pp. 71–84. MIT Press, San Diego (1990)
46. Fitting, M.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer, Heidelberg (1996)
47. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) 5th Conference on Logic Programming, pp. 1070–1080 (1988)
48. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: ICLP 1990, pp. 579–597. MIT Press, Cambridge (1990)
49. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
50. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: International World Wide Web Conference, ACM, New York (2003)
51. Heymans, S., Van Nieuwenborgh, D., Hadavandi, E.: Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 392–407. Springer, Heidelberg (2005)
52. Hitzler, P., Seda, A.K.: Mathematical Aspects of Logic Programming Semantics. Studies in Informatics. Chapman and Hall/CRC Press (2010)
53. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: Swrl: A semantic web rule language combining owl and ruleml (2004), `http://www.w3.org/submission/swrl/` (accessed January 2006)
54. Hu, Y., Yuan, L.Y.: Extended well-founded model semantics for general logic programs. in koichi furukawa, editor, In: Int. Conf. on Logic Programming, Paris, pp. 412–425 (1991)
55. Kemp, D.B., Srivastava, D., Stuckey, P.J.: Bottom-up evaluation and query optimization of well-founded models. Theor. Comput. Sci. 146, 145–184 (1995)
56. Khamsi, M.A., Misane, D.: Fixed point theorems in logic programming. Ann. Math. Artif. Intell. 21(2-4), 231–243 (1997)
57. Kowalski, R., Kuehner, D.: Linear resolution with selection function. Artifical Intelligence 2, 227–260 (1971)
58. Kunen, K.: Negation in logic programming. Journal of Logic Programming 4, 289–308 (1987)

59. Leitsch, A.: The Resolution Calculus. Springer, Heidelberg (1997)
60. Levy, A., Rousset, M.-C.: A representation language combining horn rules and description logics. In: European Conference on Artificial Intelligence, ECAI 1996 (1996)
61. Lifschitz, V.: Foundations of declarative logic programming. Principles of Knowledge Representation. CSLI publishers (1996)
62. Lloyd, J.W.: Foundations of logic programming, 2nd extended edn. Springer, New York (1987)
63. Lobo, J., Minker, J., Rajasekar, A.: Foundations of disjunctive logic programming. MIT Press, Cambridge (1992)
64. Lonc, Z., Truszcynski, M.: On the problem of computing the well-founded semantics. Theory and Practice of Logic Programming 1(5), 591–609 (2001)
65. Marek, V.W.: Autoepistemic logic. Journal of the ACM 38(3), 588–619 (1991)
66. McCarthy, J.: Circumscription - a form of non-monotonic reasoning. Journal of Artificial Intelligence 13(1-2), 27–39 (1980)
67. Minker, J.: An overview of nonmonotonic reasoning and logic programming. Journal of Logic Programming 17(2-4), 95–126 (1993)
68. Morishita, S.: An extension of van gelder's alternating fixpoint to magic programs. Journal of Computer and System Sciences 52, 506–521 (1996)
69. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. Journal of Web Semantics 3(1), 41–60 (2005)
70. Paschke, A.: Verification, validation, integrity of rule based policies and contracts in the semantic web. In: 2nd International Semantic Web Policy Workshop (SWPW 2006), Athens, GA, USA, November 5-9 (2006)
71. Paschke, A.: A typed hybrid description logic programming language with polymorphic order-sorted dl-typed unification for semantic web type systems. CoRR, abs/cs/0610006 (2006)
72. Paschke, A., Boley, H., Kozlenkov, A., Craig, B.L.: Rule responder: Ruleml-based agents for distributed collaboration on the pragmatic web. In: ICPW, pp. 17–28 (2007)
73. Pereira, L.M., Alferes, J.J.: Well founded semantics for logic programs with explicit negation. Proceedings of ECAI 1992 (1992)
74. Pereira, L.M., Alferes, J.J., Aparicio, J.N.: Adding closed world assumptions to well founded semantics. In: Fifth Generation Computer Systems, pp. 562–569 (1992)
75. Przymusinska, H., Przymusinski, T.C.: Weakly perfect semantics for logic programs. In: 5th International Conference and Symposium on Logic Programming, pp. 1106–1121 (1988)
76. Przymusinska, H., Przymusinski, T.C.: Weakly stratified logic programs. Fundamenta Informaticae 13, 51–65 (1990)
77. Przymusinski, T.C.: Perfect model semantics. In: 5th Int. Conf. and Symp. on Logic Pro- gramming, pp. 1081–1096. MIT Press, Cambridge (1988)
78. Przymusinski, T.C.: Every logic program has a natural stratification and an iterated fixed point model. Proceedings of ACM Symp. on Principles of Database Systems, 11–21 (1989)
79. Przymusinski, T.C.: On the declarative and procedural semantics of logic programs. Journal of Automated Reasonig 5, 167–205 (1989)
80. Przymusinski, T.C.: Non-monotonic reasoning vs. logic programming: A new perspective. In: Partridge, D., Wilks, Y. (eds.) The Foundations of Artifical Intelligence - A Sourcebook, Cambridge University Press, London (1990)

81. Przymusinski, T.C.: The well-founded semantics coincides with the three-valued stable semantics. Fundamenta Informaticae 13, 445–463 (1990)
82. Przymusinski, T.C.: Stable semantics for disjunctive programs. New Generation Computing 9, 401–424 (1991)
83. Rajasekar, A., Lobo, J., Minker, J.: Weak generalized closed world assumption. Journal of Automated Reasonig 5(3), 293–307 (1989)
84. Reiter, R.: A logic for default reasoning. Journal of Artificial Intelligence 13, 81–132 (1980)
85. Riccardo, R.: On the decidability and complexity of integrating ontologies and rules. Journal of Web Semantics 3(1) (2005)
86. RIF. W3c rif: Rule interchange formant (2010), `http://www.w3.org/2005/rules/` (accessed october 2010)
87. Robinson, J.: A machine-oriented logic based on the resolution-principle. JACM 12(1), 23–41 (1965)
88. Ross, K.: A procedural semantics for well-founded negation in logic programs. Journal of Logic Programming 13(1), 1–22 (1992)
89. Ross, K.: Modular stratification and magic sets for datalog programs with negation. Journal of the ACM 41(6), 1216–1266 (1994)
90. Sacca, D., Zaniolo, C.: Partial models and three-valued models in logic programs with negation. In: Workshop of Logic Programming and Non-Monotonic Reasoning, Washington D.C, pp. 87–104. MIT Press, Cambridge (1991)
91. Schlipf, J.: Formalizing a logic for logic programming. Annals of Mathematics and Artificial Intelligence, 5, 279–302 (1992)
92. Shen, Y.-D., Yuan, L.-Y., You, J.-H.: Slt-resolution for the well-founded semantics. Journal of Automated Reasoning 28(1), 53–97 (2002)
93. Shepherdson, J.C.: Negation in logic programming. In: Minker, J. (ed.) Foundations of Deductive Databases, pp. 19–88. Morgan Kaufmann, San Francisco (1988)
94. Shepherdson, J.C.: Unsolvable problems for sldnf resolution. J. of Logic Programming, 19–22 (1991)
95. Stuckey, P.J., Sudarsham, S.: Well-founded ordered search: Goal-directed bottom-up evaluation of well-founded models. The Journal of Logic Programming 32(3), 171–205 (1997)
96. Tamaki, H., Sato, T.: Old resolution with tabulation. In: 3rd Int. Conf. on Logic Programming, London, pp. 84–98 (1986)
97. Teusink, F.: A proof procedure for extended logic programs. In: ILPS 1993. MIT Press, Cambridge (1993)
98. Ullman, J.D.: Principles of Database and Knowlegebase Systems, vol. 2. Computer Science Press, Rockville (1989)
99. Van Gelder, A.: The alternating fixpoint of logic programs with negation. In: 8th ACM SIGACT-SIGMOND-SIGART Symposium on Principles of Database Systems, pp. 1–10 (1989)
100. Van Gelder, A.: The alternating fixpoint of logic programs with negation. Journal of Computer and System Sciences 47(1), 185–221 (1993)
101. Van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. JACM 38(3), 620–650 (1991)
102. You, L.H., Yuan, L.Y.: Three-valued formalization of logic programming: is it needed. In: Proceedings of 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 172–182. ACM Press, New York (1990)