Ten Theses on Logic Languages for the Semantic Web

François Bry[1] and Massimo Marchiori[2]

[1] University of Munich, Germany
http://pms.ifi.lmu.de/
[2] University of Venice, Italy, and W3C
http://www.w3.org/People/Massimo/

Abstract. This articles discusses the logic, or logic-based, languages required for a full deployment of the Semantic Web. It presents ten theses addressing
1. the kinds of logic languages needed,
2. data and data processing,
3. semantics, and
4. engineering and rendering issues.

The views reported about in this article have been presented at the *W3C Workshop on Rule Languages for Interoperability* (27-28 April 2005, Washington, D.C., USA, http://www.w3.org/2004/12/rules-ws/).

1 Languages

Thesis 1 (Diversity) *The Semantic Web requires logic languages of different kinds:*

1. *three kinds of reasoning, or deductive, languages, viz.*
   (a) *constructive rules (or views),*
   (b) *normative rules (or integrity constraints),*
   (c) *descriptive specifications (or ontologies),*
2. *and reactive rules.*

Constructive rules,[3] called 'views' in databases, specify how to derive new data from data already available. Constructive rules typically involve data selection and grouping. Constructive rules are often, but not always, expressed as implications of the form `new-data` ⇐ `query`. Examples of constructive rules are SQL views, Datalog or pure Prolog clauses,[4] and XSLT templates. Queries after XQuery can be seen as constructive rules with intertwined query and new-data parts. CSS rules can also be seen as constructive rules: CSS selectors are a kind of queries, declaration-blocks (or {}-blocks) specify how new, styled, data are constructed. RDFS semantic rules are further examples of constructive rules.

[3] The name stresses that consequences from such rules can be drawn in constructive logic, i.e. without relying on excluded middle or refutation.
[4] I.e. Prolog clauses without imperative predicates.

Inference rules[5] used in specifying proof systems, are also constructive rules (*cf. infra* Thesis 8).

Normative rules, called 'integrity constraints' in databases, express conditions that data must fulfill, e.g. ISBN numbers uniquely characterize books, and that must be checked when data are updated. Data schemas, especially tree grammars in their various disguises, e.g. DTD, XML Schema, RelaxNG, etc., express normative rules.[6] Normative rules can be expressed as denials and evaluated like constructive rules. A denial is a rule of the form `false` $\Leftarrow$ `query` where the head `false`, or `error(...)`, etc., denotes a violation of a requirement `req` and the denial's body `query`expresses a negation of this requirement, i.e. `query` $\equiv \neg$`req`. E.g. the following denial expresses that ISBN numbers uniquely characterize book titles: `error(ISBN)` $\Leftarrow$ `book(Title1, ISBN)` $\wedge$ `book(Title2, ISBN)` $\wedge$ `Title1` $\neq$ `Title2`.

Descriptive specifications specify data types and relationships between data types without necessarily referring to actual data. They are used in software specifications, data schemas, and ontologies. They are often expressed in logics[7] corresponding to classical logic fragments with *restricted quantifications* of the forms $\forall x : s\ F[x]$ and $\exists x : s\ F[x]$ restricting the variable x to some sort, class, entity, etc. s. Such quantifications can be expressed in classical logic as $\forall x\ s(x) \Rightarrow F[x]$ and $\exists x\ s(x) \wedge F[x]$, resp. using a conveniently defined unary predicate symbol s.

It is worth noting that, in many cases, the distinction between normative rules (integrity constraints) and descriptive specifications (ontologies) subtly depends on the use. Consider a system of rules expressing some regulation, e.g. under which conditions students are allowed to register for courses. In drawing conclusions from the regulation, or in verifying that it is consistent or non-redundant, the regulation is used as a descriptive specification – certain forms of reasoning such as excluded middle and refutation make sense and might even be indispensable. In verifying that student registrations to courses enforce the regulation, the regulation is used as integrity constraint – excluded middle and refutation do not make sense.[8]

Reactive rules specify how a data store can be modified depending on the current state of the store and, in some languages, on events. Reactive rules commonly have one of the forms `if condition then action` and `on event if condition then action`. Rules of the first kind are called *production rules*,[3] rules of the second, *ECA* (short for *Event-Condition-Action*) *rules*. In production and ECA rules, `condition` is an (atomic or compound) query to the data

[5] E.g. modus ponens: If both A and $A \Rightarrow B$ are provable, then B is provable.

[6] However, variables in grammars differ from logic variables, since different occurrences of a same grammar variable represent different data instances.

[7] E.g. sorted logics and description logics.

[8] One might object that Prolog, or a Prolog-like proof-system, can used for integrity checking, integrity constraints been expressed as denials, and that the proof method of Prolog, SLD resolution, is a refutation method. In fact, as opposed to general resolution, SLD resolution can be re-expressed in constructive logic [8], i.e., without referring to refutation.

store similar to a body of a constructive or normative rule, and `action` is an atomic (i.e. single) or compound update of the data store (typically consisting of insertions, removal, and/or changes in a data item). In an ECA rule, `event` denotes an *event query*, i.e. a query to events received so far. An event query can be atomic, i.e. refer to a single event, or compound, i.e. refer to composite events. In the following, the condition of a production or ECA rule is called *standard query* so as to stress its similarity with the body of a constructive or normative rule.[9]

Thesis 2 (Negation) *Non-monotonic negation*[10] *is the negation of choice for constructive rules (views), normative rules (integrity constraints), and reactive rules. Monotonic negation may, but must not, be offered in constructive, normative, and reactive rules. Monotonic negation is the negation of choice for descriptive specifications (ontologies).*

Non-monotonic negation, *cf.* [7] for selected articles, is the negation of choice for constructive rules (views) because data constructions depends on both, available and non-available data. Since normative rules can be expressed as constructive rules (*cf. supra* Thesis 1), non-monotonic negation is also the negation of choice for normative rules. Non-monotonic negation is the negation of choice for reactive rules, too, for both 'event queries' (i.e. the `event` parts of ECA rules) and 'standard queries' (i.e. the `condition` parts of production or ECA rules) refer to the presence or absence of data, events resp.

Monotonic negation is the negation of choice for descriptive specifications because descriptive specifications do not refer to actual data, e.g. the flights listed in a time table, but instead to meta-level specifications, e.g. conditions flights must fulfill, the negation needed in descriptive specifications does not have to refer to the absence or non-availability of such data.

Recall (*cf. supra* Thesis 1) that the same rule can be used as a normative specification (integrity constraint) or descriptive specification (ontologie). As a consequence, the choice of a negation semantics, monotonic or non-monotonic, does not necessarily depend on the syntax of negation.

Thesis 3 (Coherency and Inter-Operability) *Inter-operable logic languages of the various kinds should be striven for. Inter-operability is sustained by the following forms of coherency: syntax coherency, rendering coherency, reasoning coherency, and explanation coherency.*

Syntax coherency means that expressions from different languages with similar meanings are expressed similarly. *Rendering coherency* means that expressions from different languages are (visually or verbally) rendered (*cf. infra* Thesis 10) similarly, possibly using the same rendering methods or tools. *Reasoning coherency* means that similar forms of reasoning applied on different languages,

[9] [13] further discusses how constructive and reactive rules, called 'passive' and 'active' resp., relate.

[10] The negation used in concluding that flights not mentioned in a time table do not exist.

e.g. for deriving new data using constructive rules, for computing the closure of RDF specifications, or for checking normative rules, are performed using similar reasoners. Reasoning coherency is desirable both for programmers and language design, and implementation. An important aspect of reasoning coherency is to have a common semantics for non-monotonic negation in constructive, normative, and reactive rule languages. *Explanation coherency* means that similar forms of reasoning are explained, by explanation tools, relaying on similar explanation paradigms.

2 Data and Data Processing

Thesis 4 (Data Distribution and Versatility, and Meta-Level Reasoning) *A logic language for the Semantic Web must access data everywhere on the Web; be 'data versatile', i.e. capable of accessing data and meta-data in any common Web Semantic Web format – especially XML, RDF, Topic Maps, and OWL, as well as the formats of Semantic Web logic languages –, and capable of some forms of meta-level reasoning*

There has already been a number of pleas in favour of data versatile query languages, e.g. [19].

Meta-level reasoning poses interesting, but not impossible, challenges. Meta-level reasoning has bad reputation among Computational Logicians, however, conveniently, e.g. constructively, restricted, *cf.* [6] meta-level reasoning is semantically as safe, and practically as useful as higher-order functions in Functional Programming. Note that meta-level reasoning is already present, though in a limited form, on the Semantic Web: RDF Schema, the "RDF Vocabulary Description Language", is itself an RDF Vocabulary for describing terms in an RDF vocabulary.

Thesis 5 (Reasoning Paradigms) *Constructive and normative rules (views and integrity constraints) should be evaluable by both forward chaining*[11] *and backward chaining*[12]*, backward chaining being the reasoning paradigm of choice. Descriptive specifications (ontologies) call for (non-constructive) reasoning, including excluded middle*[13]*, non-contradiction*[14] *and refutation*[15]*. The reasoning paradigms of Semantic Web logic languages should support grouping, aggregation, theory reasoning, and non-monotonic negation.*[16]

On the Web, forward chaining is well-suited only for well-defined and closed sets of Web sites. Queries referring directly, or indirectly (through sub-queries triggered by constructive rules at queried Web sites) to a set of Web sites that

[11] Also called bottom-up reasoning.
[12] Also called top-down reasoning.
[13] At least one of A and $\neg A$ is true.
[14] At most one of A and $\neg A$ is true.
[15] If under the assumption A, a contradiction, i.e. B and $\neg B$ for some B, can be derived, then $\neg A$ is proven.
[16] Preferably with a semantics understandable without PhD in Logic!

cannot be statically[17] recognized, cannot be evaluated by forward chaining. Indeed, with such queries, forward chaining would require to compute intermediate results from all possible Web sites.Thus, on the web, backward chaining is the reasoning paradigm of choice for constructive and normative rules.

Theory reasoning, a term coined after Mark Stickel's 'theory resolution' [20], denotes enhancing a general purpose reasoning method with special reasoners where convenient, e.g., reasoning on bank accounts with a basic arithmetic 'theory reasoner' instead of the Peano axioms of Arithmetic.

Thesis 6 (Event Processing) *Event broadcasting is undesirable on the Web. Events can be exchanged between Web sites using a push, or a pull model. Pushed events can be sent as data streams, calling for streamed query evaluation methods. Evaluating event queries, e.g. the event parts of ECA rules, calls for event driven query evaluation methods.*

On the Web, events can not be broadcasted, i.e. indiscriminately sent to all sites, because this would result in too high a traffic. Events can be exchanged on the Web sites via either push, i.e. events are sent by the emitters to specific recipients, or pull methods, i.e. each site publishes the events it emits, together with the event's recipients, on a 'blackboard' which is repeatedly queried by the potential recipient sites. Such queries are called *continuous*. With the push model, event can be sent as 'data streams' [4]. Continuous queries [22, 1, 17, 18], data streams [4], and event queries [5, 2] require specific query evaluation methods.

3 Semantics

Thesis 7 (Declarative Semantics) *Logic languages for the Semantic Web, except reactive rule languages, should have declarative semantics defined as 'Tarski-style model theories'.*

Tarski-style models [12], i.e., the models of classical logic, are expressed in terms of so-called 'valuation functions' that are defined recursively on a formula's structure. They make possible to evaluate a formula independently of other formulas. Therefore, they are easy to understand, and they do not require complex operational semantics.[18]

Production and ECA rules amount to *imperative programming*, hence they are inherently not amenable to declarative semantics. However, (1) declarative semantics are possible and desirable for the 'standard query' and 'event query' languages used in production or ECA rules languages, and (2) a formal semantics amenable to reasoning on production and ECA rule programs is possible (and desirable!).

[17] I.e. before query evaluation.

[18] Note that most declarative semantics for non-monotonic negation that do not assume stratified, or stratifiable, rules, e.g. the stable [11] and well-founded [10] semantics, do *not* have Tarski-style model theories.

Thesis 8 (Operational Semantics) *The operational semantics of a logic language is conveniently expressed with constructive and normative rules. Backtracking is useful for a fine tuning of proof construction in implementing logic languages.*[19]

The operational semantics of a logic language or reasoner is usually and conveniently expressed in terms of inference rules of the form:

$$\frac{\texttt{Premise}_1 \ldots \texttt{Premise}_n}{\texttt{Conclusion}}$$

Inference rules can be seen as constructive rules in a meta-language specifying proofs for formulas of the object-level language. Thus, a constructive rules are subjacent to (the procedural semantics of) *every* rule language and reasoners. This observation has led to successful uses of the run-time system [21] of Prolog or of the Prolog language itself [14] for implementing efficient theorem provers. Normative rules, too, are convenient in specifying the procedural semantics of rule languages and reasoners for expressing constraints on the proof, or search, space. Reactive rule can be convenient in implementing logic languages and reasoners.[20]

4 Engineering and Rendering

Thesis 9 (Language Engineering) *Logic languages for the Semantic Web should be referentially transparent, strongly closed, have Web formats, and modern type systems.*[21] *The specification of abstract machines should be striven for.*

Referential transparency, i.e. within a same declaration scope two occurrences of a same expression have the same meaning, is desirable because it is *the* trait of declarativity. *Closure*, i.e. the data returned by a program are like, e.g. have formats similar to, the data accessed by programs in the same language. *Strong closure* means that the data returned by a program can be further processed by this same program. Strong closure is desirable because it eases structuring programs in sub-programs. *Web formats*, especially XML formats such as RuleML formats, are desirable for rule languages because they eases inter-changing programs on the Web, e.g., for Web services applications. *Abstract data types* and

[19] Backtracking is however undesirable as a programming concept for high-level logic languages like the logic languages needed on the Semantic Web because it destroys the language's declarativity. The operational paradigm(s) desirable for a Semantic Web logic languages can be equivalently called 'backtracking-free logic programming' or 'set-oriented functional programming'. It is worth noting almost of the query languages proposed for RDF are of this kind.

[20] Since constructive and reactive rule languages can be used in specifying and implementing logic languages and reasoners, some claim that a single language of such a kind would be sufficient for the Semantic Web. This amounts to claiming that only one single, e.g., imperative, programming language could be sufficient for developing software.

[21] I.e., type systems supporting abstract data types and offering static type checking, parametric polymorphism, and modules.

static type checking are desirable for Semantic Web reasoning and reactive languages as they are for any other programming languages: *"Well typed programs do not go wrong."* [16] *Abstract machines* are desirable because they are essential for wide-spreading languages.

Thesis 10 (Visual and Verbal Rendering) *Logic languages for the Semantic Web should have visual and verbal renderings.*

Declarative languages are especially well-suited to visual rendering and visual rendering is very appealing to potential users of logic languages for the Semantic Web, as the many systems for graphical rendering and/or visualization of business rules amply demonstrate.

Programs used on the Web and Semantic Web should be *verbalizable*, i.e. the rules or formulas they consist of should be expressible in a controlled language [15, 9], i.e. in a non-ambiguous language resembling natural language. Rules, e.g. expressing policy specifications and trust, verbalized in a controlled language would considerably help wide-spreading the (verbal as well as non-verbal forms of the) languages they are expressed in.

Acknowledgments. The ideas expressed in this article have been significantly influenced by the research project REWERSE (Reasoning on the Web with Rules and Semantics, `http://rewerse.net`). The authors thank their colleagues of REWERSE for many fruitful discussions on the subject of this article. This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (*cf.* `http://rewerse.net`).

References

1. Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 2001.
2. James Bailey, François Bry, and Paula-Lavinia Pătrânjan. Composite Event Queries for Reactivity on the Web. In *Proc. 14th Int. World Wide Web Conference*, 2005.
3. Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming.* Addison-Wesley, 1985.
4. François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. The XML Stream Query Processor SPEX. In *Proc. 21st Int. Conf. on Data Engineering (ICDE)*, 2005.
5. Franois Bry and Paula-Lavinia Pătrânjan. Reactivity on the Web: Paradigms and Applications of the Language XChange. In *Proc. 20th Annual ACM Symp. Applied Computing (SAC)*, 2005.
6. Weidong Chen, Michael Kifer, and David Scott Warren. HILOG: A Foundation for Higher-Order Logic Programming. *Jour. of Logic Programming*, 15(3):187–230, 1993.
7. Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski., editors. *Selected Papers from the Non-Monotonic Extensions of Logic Programming.* LNCS 1216. Springer-Verlag, 1996.

8. K. Doets. *From Logic to Logic Programming*. MIT Press, 1994.
9. Norbert E. Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English – Not Just Another Logic Specification Language. In *Proc. 8th Int. Workshop (LOPSTR)*, LNCS 1559. Springer-Verlag, 1999.
10. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Jour. ACM*, 38(3):620–650, 1991.
11. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. Int. Conf. and Symp. Logic Programming*, 1988.
12. Jerome Keisler. *Handbook of Mathematical Logic*, chapter Fundamentals of Model Theory, pages 47–103. North-Holland, 1989.
13. Rainer Manthey. Active and Passive Rules in Database Systems: How do They Relate. In *Proc. 1st Workshop on Advances in Databases and Information Systems*, 1994.
14. Rainer Manthey and François Bry. SATCHMO: A Theorem Prover Implemented in Prolog,. In *Proc. 9th Conf. on Automated Deduction*, 1988.
15. Massimo Marchiori and Janne Saarela. Query + Metadata + Logic = Metalog. In *Proc. QL '98, The Query Languages Workshop*, 1998. http://www.w3.org/TandS/QL/QL98/.
16. Robin Milner. Fuly Abstract Models of Typed λ-Calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.
17. Benjamin Nguyen, Serge Abiteboul, Gregory Cobena, and Mihai Preda. Monitoring XML Data on the Web. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, 2001.
18. Sandeep Pandey and and Soumen Chakrabarti Krithi Ramamritham. Monitoring the Dynamic Web to Respond to Continuous Queries. In *Proc. 12th Int. World Wide Web Conference*, 2003.
19. Jonathan Robie. The Syntactic Web: Syntax and Semantic on the Web. In *Proc. XML Conf. and Exposition*, 2001.
20. Mark E. Stickel. Automated Deduction by Theory Resolution. *Jour. of Automated Reasoning*, 1(4):333–355, 1985.
21. Mark E. Stickel. A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Computer. *Jour. of Automated Reasoning*, 1988.
22. Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 1992.