Engaging Prolog with RDF

Borys Omelayenko
Department of Computer Science
Vrije Universiteit, De Boelelaan 1081, 1081hv,
Amsterdam, the Netherlands
borys@cs.vu.nl

Abstract

Prolog has been often used to represent the axioms and inference over RDF data models often by converting all the data to plain-text Prolog facts and programs. In this paper we present the PRoDeF infrastructure for using Prolog for inferencing over RDF data on the Web by representing Prolog programs in RDF, allowing them to be distributed over the Web and even incomplete, and represent reasoning results in a form suitable for further automatic processing.

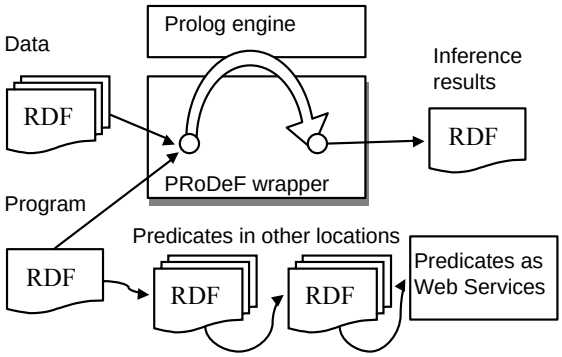


Figure 1: The infrastructure.

1 Introduction

RDF and RDF Schema lack the means for representing axioms and rules, which are still necessary to build any kind of applications and different approaches originating from different motivations and requirements have been proposed. The same time Prolog [Bratko, 1990] has been extensively used to inference over data models represented in RDF,[1] however, mostly RDF and Prolog have been connected in an ad-hoc manner, primarily by converting everything to plain-text Prolog facts and programs.

We intend to build the infrastructure depicted in Figure 1, where a special wrapper connects the Prolog engine to the Web. It parses a Prolog program represented in RDF and downloads the RDF data modules to be processed with the program. The program itself is distributed to several locations and the predicates used in one location may be defined in other places. The predicates, especially resource-critical or performing some specific function, may even be implemented in other languages and accessible as web services. Finally, the inference results are represented in RDF for further automatic processing.

We believe that such an infrastructure should possess the following basic properties:

1. The language should have clear semantics, sufficient and stable tool support, and existing expertise in terms of available literature, courses, and skills
2. The Prolog programs should be available on the Web in a distributed manner possibly decomposed into pieces and represented in RDF according to a certain RDF Schema
3. RDF data should be interpreted in Prolog
4. Program execution should assume that some parts of the program may require some time to download from different locations or being not available at the moment
5. A clear algorithm for converting plain-text Prolog programs to and from their RDF representation should be provided
6. The reasoning results should be represented in RDFand should allow updating the data
7. The rule language should allow representing the constraints at the RDF Schema level and smoothly link them to instance data .

We try to meet these requirements in the Prolog wrapper discussed in this Chapter.

1.1 Where are the Limits of Ontology Languages?

The ontology languages for the Semantic Web incorporate certain means for representing axioms that can be then used without any additional rule language.

RDF and RDF Schema contain the axioms needed to form the object-attribute language for representing the conceptual models: rdfs:subClassOf and rdfs:subPropertyOf are used to organize classes

[1] http://www.google.com/search?q=using+prolog+rdf

and properties into hierarchies, `rdfs:domain` and `rdfs:range` specify the attachment of properties to classes. This set of axioms is often difficult to use in practice, e.g. the conjunctive semantics of multiple occurrences of `rdfs:domain` or `rdfs:range` means that a property may be attached to an intersection of one or more classes, and not a union (disjunctive semantics). This poses some problems whenever a property has to be attached to several classes.

In OWL RDF Schema is extended and several groups of axioms are introduced. These are:

Equality axioms `sameClassAs`, `samePropertyAs`, `sameIndividualAs` and `differentIndividualFrom` to denote that two classes, properties or individual are equivalent;

Property characteristics are introduced to define property characteristics `inverseOf`, `TransitiveProperty`, `SymmetricProperty` to define inverse, transitive and symmetric properties; `allValuesFrom` and `someValuesFrom` to define property range restrictions;

Cardinality constraints of properties.

This set of axioms allows modelling numerous frequently needed constraints. For example the `bossOf` relation is often used to represent organizational structures. To model its transitivity in RDF Schema one needs to create and interpret a special rule, while it can be directly modelled in OWL with the `TransitiveProperty` property. However, in many organizations the set of bosses of an employee who can actually give him the orders is limited to two levels: the immediate boss and his/her immediate boss, and not a single step further. This axiom can not be modelled in OWL directly and a rule is needed to model this two-steps transitivity.

Another sort of examples include the axioms using value constraints, e.g. to classify some offers according to price where cheap offers would assume `0 EUR < price < 500 EUR`.

1.2 Rule Languages for RDF

Many applications require rules and axioms that can not be directly represented in RDF Schema or OWL. However, RDF and RDF Schema do not possess any rule language, that is caused by numerous difficulties that are expected in standardization of such a language at present time. However, this need has been widely understood in the Semantic Web community and several approaches for such a rule language have been proposed.

Triple [Sintek and Decker, 2001] is proposed as an RDF query and inference language, providing full support for resources and their namespaces, models represented with sets of RDF triples, reification, RDF data transformation, and an expressive rule language for RDF. The language is intended to be used with a Horn-based inference engine.

The RuleML[2] initiative aims at defining a shared rule markup language to specify forward (bottom-up) and backward (top-down) rules in XML. The language being developed within the initiative is essentially an XML serialization for the rules, and it specifies the rules in a generic form of a head and a body consisting of atomic predicates with parameters that can be also interpreted in Prolog. RuleML is probably the only rule language for RDF that defines an RDF syntax for the rules themselves.

However, there are several differences in the goals pursued in RuleML and PRoDeF. These are:

RDF facts. RuleML focuses at a universal representation of the rule on the (Semantic) Web and thus makes no assumptions about the structure and arity of the facts, while any inference engine dealing with RDF naturally deals with binary RDF facts only;

Implementation. RuleML is not linked to a specific inference engine and thus needs to provide its own interpretation of the rules together with a linkage to inference engines. PRoDeF follows the opposite approach tightly connecting the RDF serialization to standard Prolog semantics. From the engine implementation side, PRoDeF relies on the decades-long experiences in making Prolog engines;

RDF Schema interpretation. The RDF serialization used in RuleML makes no commitment to RDF Schema and uses `rdf:Bag`s and `rdf:Seq`uences to represent the rules that are not representable in RDF Schema.

The RuleML initiative has been hosting a workshop on rule languages for the Semantic Web where a number of initiatives have been presented.[3]

Squish[4] also known as RDQL is somewhat similar to SQL but is further elaborated to query RDF triples. This similarity to SQL allows seamless integration with database back-ends. However, querying in Squish is bounded to plain RDF, without any support for RDF Schema or high-level languages.

The Sesame RDF querying engine [Broekstra *et al.*, 2002] uses the RDF Query Language RQL.[5] Similar to Squish, RQL statements contain the select-from-where construct, however, an RQL interpreter is supposed to understand RDF Schema axioms: transitivity of the subclass-of relation, its connection to `rdf:type`, etc.

A comparison of different RDF query languages is published on the W3C web site[6] together with query samples and may serve as an interesting information source.

The Object Constraint Language OCL[7] is the expression language for the Unified Modeling Language (UML) that allows specifying constraints about the objects, links, and property values of UML models. OCL is a pure expression language and any OCL expression is guaranteed not to change anything in the model. Whenever an OCL expression is evaluated, it simply delivers a value. OCL is a modelling lan-

[2] `http://www.dfki.uni-kl.de/ruleml/`

[3] `http://www.soi.city.ac.uk/~msch/conf/ruleml/`

[4] `http://swordfish.rdfweb.org/rdfquery/`

[5] `http://sesame.aidministrator.nl/publications/rql-tutorial.html`

[6] `http://www.w3.org/2001/11/13-RDF-Query-Rules/`

[7] `www.omg.org/docs/ad/97-08-08.pdf`

Language	Requirements from Section **??**				
	1:Exp.	2:Web	3:RDF	4:Exec.	6:Res.
Triple	–	–	+	–	–
RuleML	–	+/–	–	–	–
RQL	+	–	+	–	(planned)
Squish	–	–	–	–	–
PAL	+/–	–	–	–	–
OCL	+/–	–	–	–	–
Prolog	+	+⋆	+⋆	+⋆	+⋆

⋆ in PRoDeF

Table 1: An estimate of the popularity of each of the rule language proposals for RDF, as queried on 7 January 2003

Query string	Papers in CiteSeer	Google results
RDF and Triple	4	282
RDF and OCL	2	591
RDF and RuleML	11	803
RDF and Prolog	31	16,100

Table 2: An estimate of the popularity of each of the rule language proposals for RDF, as queried on 7 January 2003

guage rather than a programming language and it is not possible to write program logic or flow-control in OCL. As a side effect, not everything in OCL is promised to be directly executable.

The Protégé axiom language PAL[8] is used together with the Protégé editor[9] to specify knowledge base constraints. The syntax of PAL is a variant of the Knowledge Interchange Format (KIF) and it supports KIF connectives but not all of KIF predicates and statements. PAL is not really targeted towards RDF and RDF Schema.

The RDF parser for SWI Prolog is a very relevant and popular initiative on using Prolog with RDF. The parser is capable of converting RDF documents into Prolog facts and then utilize Prolog for reasoning, but it does not address RDF representation of Prolog programs themselves.

Table 1 represents a summary showing the features possessed by the languages in respect to the desired features listed in Section **??**. As we can see none of the languages fulfills all of them with RQL being the closest.

Interesting to mention the estimate of the popularity of the rule and query languages. We queried the Web for relevant documents as presented in Table 2. The table illustrates that Prolog has been frequently used with RDF, however, with a relatively small amount of publications made on that. Obviously, these results are a subject of various distortions and they do not indicate more than they do. However, it is obvious that none of the newly proposed languages has the tool support able to compete with the decades-long experience in Prolog tool development.

[8] `http://protege.stanford.edu/plugins/paltabs/pal-documentation/`

[9] `http://protege.stanford.edu/`

2 The Usage Scenario

Consider the prototypical scenario shown in Figure 2 that depicts an RDF document that has a certain constraint `a` attached to it with the `goal` property. It refers to the definition of `a` made in another file as a PRoDeF program. To verify the document over the constraint a Prolog parser needs to access the definition of `a`, that is, in turn, defined over `b` and `c`, where `c` is again defined in another file. In this way a parser needs to go along the `rdfs:isDefinedBy` links attached to the predicates and extract their definitions from different locations on the Web. At certain moment all the predicates would be collected, defined in terms of `l_triple` and `o_triple`'s, and the constraint can be verified.

3 The Ontology for PRoDeF

The ontology for PRoDeF represents the syntactic structure of Prolog programs and is depicted in Figure 3. In this ontology we do not try to represent the execution semantics of the programs, but treat program text as data and encode it as data, leaving its interpretation to a Prolog engine.

The modules of Prolog code are modelled with the class `PrologModule` that contains module's logical name represented with the `rdf:id` attribute and physical location of the module encoded with the `rdfs:isDefinedBy` attribute.[10] A module may export several predicates linked with the `export` property, call several directives, e.g. `consult`, as mentioned in the `calls` property, and contain rule `definitions`. `PrologModule`s are instantiated with RDF files with program code located somewhere on the Web.

The class `PredicateName` represents a predicate `name` that requires certain `numberOfParameters`. The `name` tag is targeted at a human user while the `rdf:id`'s of the `PredicateName` instances represent their identifiers used by the parser. The ontology includes several pre-defined instances of `PredicateName` reserved for `o_triple` and `l_triple` that correspond to the fact names reserved in PRoDeF to represent RDF data, and `bagof`, `setof` and `forall` that correspond to the special Prolog constructs. The `PredicateName`'s represent the predicate names without any connection to their possible use with different parameters.

These are represented with the `ClauseWithParameters` class that connects a `predicate` name to a list of `parameter`s. The parameters are organized as a list of instances of the `Parameter` class, where each instance corresponds to one `parameter` (a variable or a constant) and points to the next parameter in the list.

For example, an instance of `PredicateName` may look like the following:

```
<PredicateName rdf:id="MyPredicate"
name="myPredicate" numberOfParameters="3"/>
```

and correspond to `myPredicate/3`,[11] and an instance of `ClauseWithParameters` may look like this:

[10] `rdfs:isDefinedBy` belongs to RDF and RDF Schema and are not presented in the figure.

[11] Some of the conventions on encoding predicate names in Prolog that are lifted in PRoDeF as described later

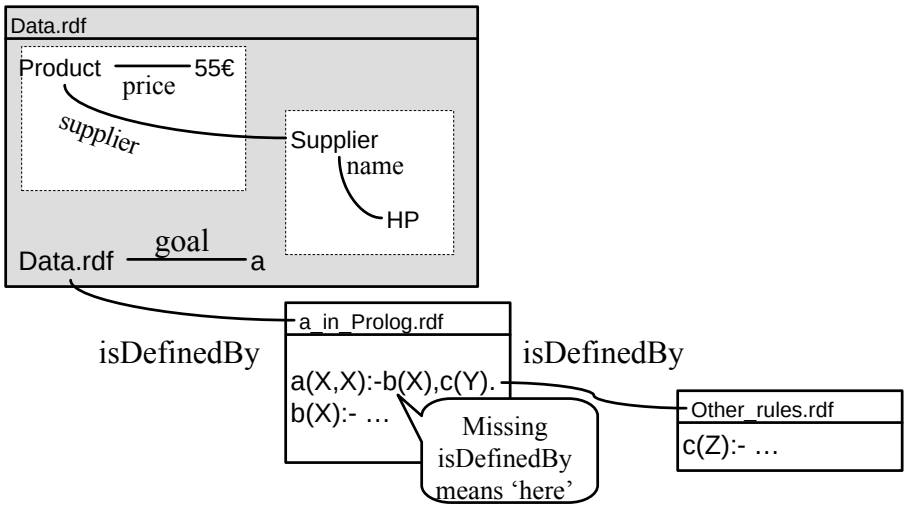


Figure 2: The usage scenario for PRoDEF: a data module named `Data.rdf` contains two objects: object `Product` with properties `Prince` and `supplier` linking it to the second object `Supplier` with property `name`. This data module has to comply the constraints represented by the `goal` predicate named `a`. In turn, `a` is defined in another file named `a_in_Prolog.rdf` as represented by the RDF Schema property `isDefinedBy`. The definition of `a` consists of two predicates `b` defined in the same file as `a`, and `c` defined in another file named `Other_rules.rdf`.

```
<ClauseWithParameters  rdf:id="CLAUSE03"
  predicate="MyPredicate"
  parameter="PAR01"/>
<Parameter rdf:id="PAR01"
  nextParameter="PAR02"
  parameter="VAR01"/>
<Parameter rdf:id="PAR02"
  parameter="CONST01"/>
<Variable rdf:id="A"
  name="A"/>
<Constant rdf:id="CONST00"
  value="This is a string constant"/>
```

and correspond to `myPredicate(A,'This is a string constant')`.

Similar to the predicates, the RDF data triples are modelled as the instances of `ClauseWithParameters`. Figure 5 shows the relation between the standard RDF modelling of RDF triples and the one used in PRoDEF. A certain triple `Product01, price, 55 EUR` (e.g. triple #23) is modelled in RDF with an instance of `rdf:Statement` with the property `rdf:Predicate` pointing to the property name, and in PRoDEF – with an instance of `ClauseWithParameters` with the property `predicate`. The `rdf:Subject` is modelled in PRoDEF with the first `Parameter` linked to `ClauseWithParameters` with the property `parameters`. The `rdf:Object` is modelled with the second `Parameter` linked to the previous one with the `nextParameter` property.

However, the modelling and interpretation of RDF triples is primarily done with the supporting tools and not by a human user. Accordingly, the different ways of modelling the triples in RDF and PRoDEF may not affect the utility of the ontology.

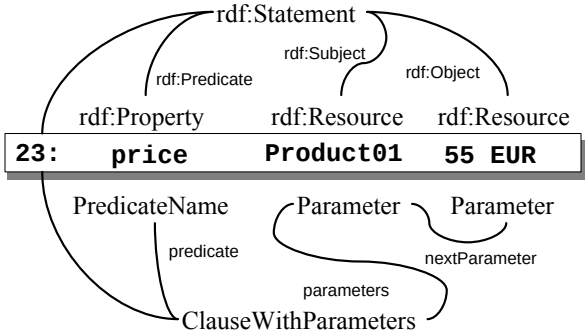


Figure 5: The way RDF statements are aligned to PRoDEF clauses.

Figure 4 illustrates how a piece of a Prolog program may be encoded in PRoDEF. The figure contains the sample code defining transitive `subClassOf` predicate and the tree illustrating its RDF representation in PRoDEF. The tree contains two branches: `RULE000` and `RULE001` corresponding to the two (disjunctive) definitions of `subClassOf` and point to the `PiR:predicate` name `subClassOf`. `RULE000` comes with the list `PAR000` of `PiR:parameters`, containing `PiR:parameter X` and `PiR:nextParameter Y`. The `PiR:body` of the rule consists of the clause `CLS000` pointing to the name `o_triple` and its parameters `X`, `Y`, and constant `'...#subClassOf'`. In a similar way `RULE001` has its `PiR:body` clause `CLS001` with `PiR:parameters X, Z`, and constant `'...#subClassOf'`, and the `PiR:nextClause CLS002` pointing to `PiR:predicate subClassOf`

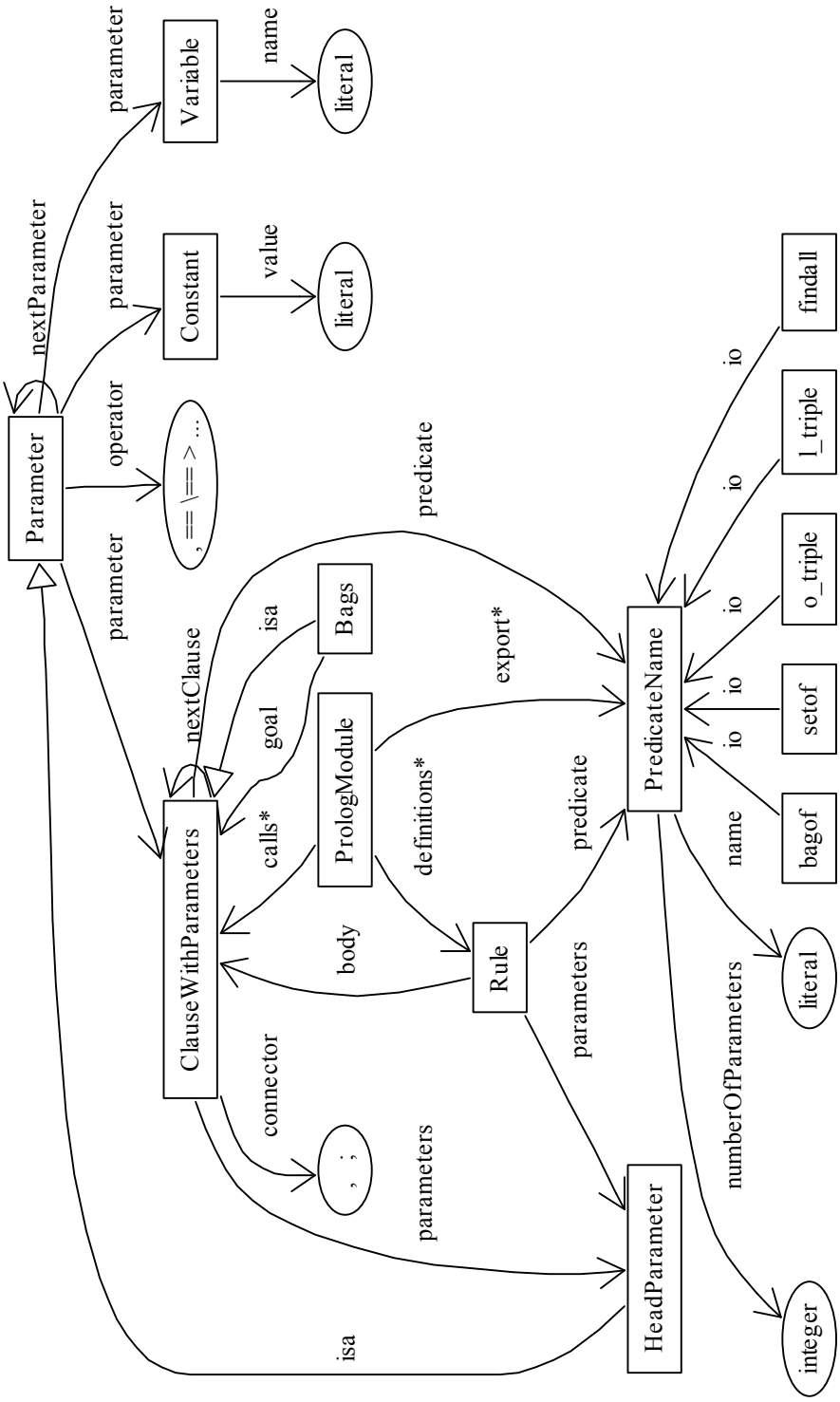
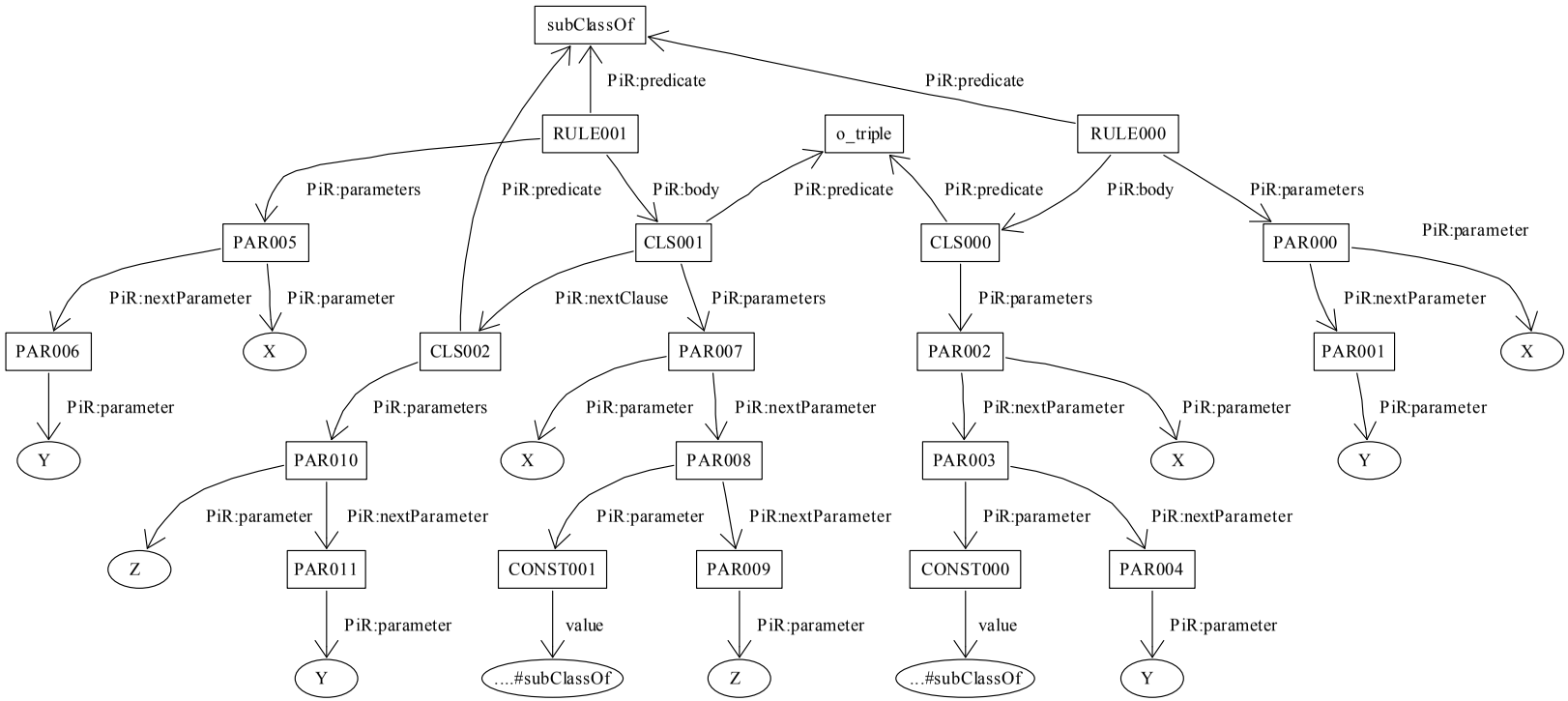


Figure 3: The ontology for representing Prolog programs in RDF




```
subClassOf(X,Y) :- o_triple(X,'http://www.w3.org/TR/1999/PR-rdf-schema-19990303#subClassOf',Y).
subClassOf(X,Y) :- o_triple(X,'http://www.w3.org/TR/1999/PR-rdf-schema-19990303#subClassOf',Z),subClassOf(Z,Y).
```

Figure 4: An example of a Prolog program being encoded in RDF

and parameters Z and Y.

3.1 RDF facts

The facts used by the rule language should not be abstract and disconnected but need to be grounded to RDF statements.

The facts in Prolog are represented with statements of an arbitrary arity indicating that one or more string-valued concepts are in a certain relation to each other.

For example, the statement `price('Product','55 EUR')` denotes that something called 'Product' is in relation 'price' to something called '55 EUR'.

In RDF the facts are represented with the RDF triples. Each triple of the form `(object,property,value)` denotes that two objects, `object` and `value` are in relation `property` to each other.

For example, the previous statement can be re-written as RDF triple `('Product',price,'55 EUR')`, which may be encoded in RDF/XML as the following:

```
<rdf:Description rdf:about="Product"
price="55 EUR"/>
```

In RDF only binary facts are allowed and a thus only binary facts need to be represented to Prolog. We interpreted them in a uniform way: each RDF triple `(object, property, value)` where `value` takes `rdfs:Literal` strings is represented with Prolog fact `l_triple(object,property,value)`, a triple with an `rdf:Resource` value is translated into Prolog fact `o_triple(object,property,value)`. No other facts are allowed.

This interpretation is similar to the one used in SWI-Prolog[12] where the result of importing an RDF file is a list of `rdf(Subject, Predicate, Object)` triples, where `Subject` is either a plain resource (an atom), or one of the terms `each(URI)` or `prefix(URI)` with the obvious meaning. `Predicate` is either a plain atom for explicitly non-qualified names or a term `NameSpace:Name`. If `NameSpace` is the defined RDF name space it is returned as the atom `rdf`. Finally, an `Object` is represented by its URI, a `Predicate` or a term has the format `literal(Value)` if they take literal values.

3.2 Names for Predicates and Variables

Historically, Prolog imposes certain constraints on the names for predicates and variables that originate from the plain-text encoding used in Prolog programs. In standard Prolog predicate names are represented with the identifiers starting with a small letter, and variable names start with a capital letter. This way of name encoding looks a bit archaic from the XML and RDF perspective.

We encode both predicates and variables as RDF objects whose `rdf:ID`'s correspond to their identifiers (or names). These objects are easily distinguished because they are defined as the instances of a certain class, either `PredicateName`, `Variable` or a `Constant`. Accordingly, we lift the restriction on the case for the first letter, and allow the use of different namespaces as the qualifiers to distinguish different predicates with the same names on the Web.

Opposite to the predicates, the variables are used only locally within a single rule definition and may not be accessed from the outside.

Predicate names make some sense only if there is a link to the location where they are actually defined. We use the `rdfs:isDefinedBy` property of a resource to denote the file where the predicate is actually defined. [**?**] defines `rdfs:isDefinedBy` as 'an instance of `rdf:Property` that is used to indicate a resource defining the subject resource. This property may be used to indicate an RDF vocabulary in which a resource is described' and thus is perfectly suitable for this purpose.

3.3 Namespaces

In XML and RDF namespaces are used as qualifiers for the names allowing two equivalent names to be distinguished globally by having different qualifiers. We use the namespaces as parts of predicate and variable names. Essentially the namespaces for the variables that are used only within the predicate definitions are not that important as for the predicates that may be accessed globally.

4 The Execution of PRoDeF Modules on the Web

The Prolog programs on the Web are executed in a different way than in classical Prolog systems and the inference results are produced for further automatic processing rather than direct human consumption.

4.1 Modules and Goals

Prolog programs are decomposed into modules that are imported by the engine by executing the directive `consult`. In the Web scenario the modules may well be distributed all over the Web and consulting a module would require prior downloading of the correspondent RDF file. Accordingly, instead of a local file name `consult` need to receive an URL of the module.

In PRoDeF each RDF data module contains two parts: RDF data itself and a possible annotation of the `rdf:RDF` tag with the special `pir:goal` property linking it to the goal description, a set of axioms that are applicable to the data module:

```
<rdf:RDF pir:goal="http://...goal.rdf">
here goes RDF data
</rdf:RDF>
```

The goal descriptions are special objects that define the axioms, the applicable data modules, and the interpretation of the axioms. It consists of (Figure 6):

`axioms` pointing to the axiom with the `name` property and its definition with the `rdf:isDefinedBy` property. The axioms are subclassed into `PositiveAxiom`s and `NegativeAxiom`s to specify whether the predicate is a positive test those results represent correct data, or a negative test that results in the incorrect data.

[12] `http://www.swi-prolog.org/packages/rdf2pl.html`

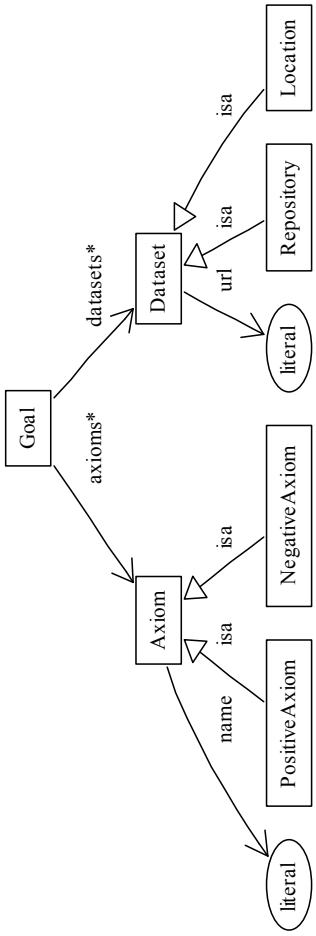


Figure 6: The structure of a goal.

`datasets` that are applicable to the axioms. Each dataset may consist of several locations pointed with their `url`s, or a query made in a repository.

Quite often it happens that a certain location with a piece of a program is not accessible at the moment. What should if the definition of a certain predicate can not be found? A possible solution path is to provide the Prolog engine with a parameter specifying server's behavior: to wait, to ignore the predicate, or to fail. This failure is then included in the reasoning results.

4.2 Reasoning Results

The Prolog engine and the wrapper return two types of information:

The list of failed locations that could not be accessed and the data or program modules could not be downloaded;

Prolog inference result: `success`, `failure`, `yes`, or `no`;

The list of solutions in form of tuples $(x_1, ..., x_n)$ that correspond to the goal predicate with arguments $(X_1, ..., X_n)$ returned in case of `success`.

We naturally represent the solutions as a bag of RDF objects, each of which contains n properties with the names $X_1, ..., X_n$ and the values $x_1, ..., x_n$.

If a goal is defined as a conjunction of several predicates $goal(X_1, ..., X_n) :- P_1(X_1, ..., X_m), ..., P_k(X_1, ..., X_m)$ where each P_i receives some or all of the n arguments of $goal$, then we may represent each resulting object as a set of objects $P_1, ..., P_k$, each of which corresponds to one of the predicates defining the goal. It may then make sense to explicitly represent these predicates in the reasoning results and process them further separately.

Accordingly, the Prolog interpreter receives a parameter 'detail level of the results' $1, 2, ..., \infty$ that specifies the number of objects representing each result, where ∞ forces the results to be fully decomposed. However, further elaboration of this scheme is rather a subject of further research.

5 Summary

In the paper we propose a solution for the problem of representing Prolog programs on the (Semantic) Web and dealing with distributed data modules in RDF.

A number of questions remain open:

- How the language should be restricted (or better to say, which extensions to Prolog should be prohibited). Primarily this refers to the problems of batch execution of the programs that may not use any graphic user interface nor console output;
- The definition of an interface between PRoDeF and the predicates implemented with the other languages and available as web services;
- A number of issues concerning distributed program execution remain open.

Extra information together with the ontologies, examples and occasional tool support is available at the PRoDeF homepage.[13]

References

[Bratko, 1990] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1990.

[Broekstra *et al.*, 2002] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Ian Horroks and James Hendler, editors, *Proceedings of the First International Semantic Web Conference (ISWC-2002)*, number 2342 in LNCS, pages 54–68, Sardinia, Italy, June 9-12 2002. Springer-Verlag.

[Sintek and Decker, 2001] Michael Sintek and Stefan Decker. TRIPLE - An RDF Query, Inference, and Transformation Language. In *Proceedings of the Workshop on Deductive Databases and Knowledge Management (DDLP-2001)*, October 20-22 2001.

[13] `http://www.cs.vu.nl/~borys/PiR/`