LMU

LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

KWRL - an RDF based rule language for annotations

Maximilian Kwapil
supervised by Jakub Kotowski and Prof. Dr. François Bry

Projektarbeit im Rahmen des Fortgeschrittenenpraktikum
WS 2008/2009, SS 2009, Abgabe: 03.12.2009

Erklärung

Hiermit versichere ich, dass ich diese Projektarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 03.12.2009 Maximilian Kwapil

Abstract

This project thesis presents KWRL - a rule language for annotations for the KiWi project. The thesis consists of two main parts. The first main part introduces RDF as the underlying language for the project and KWRL itself and general inference. Existing query and rule languages and RDF reasoners are then examined to see whether any of these are usable for the project.

The second main part deals with KWRL itself. It introduces the concept of annotations, defines the syntax of KWRL queries, constructions, rules and programs; it sketches the semantics for KWRL and finally specifies what needs to be done in further research.

Table of contents

Kapitel 1

Introduction

KWRL is short for 'KiWi [2] Rule Language'. As the title of the thesis suggests, it is a rule language based on RDF (resource description framework, [3]) and it is (primarily) for annotations.

In this thesis, all of these keywords (KiWi, RDF, rule language, annotations) and related concepts will be explained and KWRL itself will be introduced and defined on top of these concepts.

1.1 KiWi overview

Very simplified, KiWi itself is a semantic social wiki environment. This means that KiWi is...:

...a wiki A wiki is a special kind of website in that it is basically an online text editor for creating websites that anyone can view and edit. Wikis can be shared on the Internet (see Wikipedia) and/or can be used locally - for example on a company's intranet.

...a semantic wiki In a semantic wiki, users can enhance pages with metadata so as to give meaning (semantics) to the contents of their wiki. This can be for example things like GPS data for pictures, classification for animals or chemicals and virtually any extra information imaginable.

...a social wiki KiWi takes the social component of a wiki one step further as it encourages social behavior wherever possible.

KiWi will also include *annotations* which basically associate user-given metadata in the form of tags or RDF triples to wiki pages, or, more generally, content items. Users will be able to define rules over those annotations and RDF data stored in the KiWi system and a reasoner will be employed to evaluate those rules.

The system will have several features to provide the foundation for the social behavior in KiWi. The features include:

a version control system

origin tracking that is the tracking of the origin or sources of general (meta)data or even rules which can be rules and users respectively

tag-based recommendations as a feature for personalisation

This has been only a brief introduction to the KiWi project, for more details, definitions, use cases and examples see the KiWi vision paper [2].

1.2 Contents of the thesis

The next part of the thesis will show simple use cases that serve to motivate KWRL in the social wiki environment.

After that, RDF as the data for the database of KiWi is introduced which will consecutively serve as the database for KWRL, too. Several query languages and rule languages for RDF are then introduced and described along with general inference methods that are also used by a few selected reasoners which are surveyed to see whether they are suitable for the project.

The main part and last part finally deals with the introduction and definiton of KWRL with its syntax and preliminary semantics. The concept of an annotation will be properly defined and the rule language will be based upon it, leaving as future work the implementation of KWRL and a reasoner to go along with it.

1.3 Use cases - software development

This part serves to motivate rule engines in general and the rule language KWRL.

Consider a software development team working in a wiki-friendly company or research facility. The team has set up a wiki and the team members share all kinds of information on it, including pages for meetings, special events, deadlines and so on which are stored as content items in their KiWi system. The pages can be annotated by each team member and each of the member has an own content item by which he is described himself.

Note that the example annotations and rules are just sketches of the high-level KWRL syntax and the actual, complete syntax and structure is only properly defined in the KWRL chapter itself. For the use cases, an annotation is an association between metadata and content items where content items are wiki pages and metadata can be the following: atomic tags (represented by a label), structured tags (which is an atomic tag that has one or more atomic tags as arguments) and RDF triples.

Use case 'event reminder'

Although the team members already have numerous years of work experience, they do not always remember important upcoming events such as meetings or congresses in time. In order to avoid confusion and prevent hasty preparations, the system is set up to remind the team members several days before each such event by creating an annotation stating that the event is upcoming on the page describing the event.

Assuming that the current date is always part of the knowledge database and the unit[1] for the arithmetics is days, a simple rule like this could be used for such a reminder:

```
(:today :is ?x), (stag:'eventOn'(?y), ci:'kiwici:planpage',...)
WHERE ?y = ?x+5 ->
(tag:+'upcomingEvent', ci:'kiwici:planpage',...)
```

which will create an annotation to warn about an upcoming event if the current day is x and an important event is 5 days past x.

In this rule, *(:today :is ?x)* is an RDF triple with a variable *?x*, the *(stag:'eventOn'(?y), ci:'kiwici:planpage',...)* is an annotation with a structured tag as its metadata, *WHERE ?y = ?x+5* is a WHERE clause similar to SQL (structured query language) WHERE clauses and *(tag:+'upcomingEvent', ci:'kiwici:planpage',...)* is the resulting annotation containing an atomic tag as metadata.

The resulting annotation about an upcoming event can be searched for by a query program and the respective event can be displayed for the user in a way that they can define for themselves.

Use case 'calling in sick'

A particular team member, called 'Bob', has not been feeling very well the past few days. Although he knows that an important meeting is coming up on friday, he knows that he will need the time to get better so he calls in sick. He logs on to the team wiki, goes to the personnel page and annotates the page with the following data:

```
(stag:'isSick'('Bob', 'monday', 'friday'), ci:'kiwici:personnelpage',...)
```

Here, the structured tag takes three arguments, specifying who is sick and from which day to which other day the person is sick.

Now there is a rule in the system:

```
(stag:'isSick'(?p, ?f, ?t), ci:'kiwici:personnelpage',...),
(?d :is :day) WHERE ?f <= ?d <= ?t
-> (stag:'isSick'(?p, ?d), ci:'kiwici:adminpage',...)
```

stating that a person is sick on each day from the first to the last day of the time span. Again, RDF data, structured tags and a WHERE clause are used to express this.

The reasoner infers with the data from Bob that

```
(stag:'isSick'(Bob, monday), ci:'kiwici:adminpage',...), ...,
(stag:'isSick'(Bob, friday), ci:'kiwici:adminpage',...)
```

i.e. Bob is sick everyday from monday to friday inclusively.

It is further expressed that sickness is a reason for absence:

```
(stag:'isSick'(?p, ?d),...) -> (stag:'absent'(?p, ?d),...)
```

Another reason for absence is the attendence of a remote conference:

[1] Note that although units are important for this kind of rules, the topic must still be worked out in the future

```
(stag:'conference'(?c, ?d), ci'kiwici:planpage',...),
(?c :is :remote), (?p :attends ?c) ->
(stag:'absent'(?p, ?d), ci:'kiwici:adminpage',...)
```

This is not relevant for Bob though as he has no conference to attend in the near future.

New project

As soon as Bob feels healthy enough to go to work again after his sickness, he hears that a big meeting discussed a new project that the team could take after their current project.

The new project must be an important one, since its page is already annotated such that the project should be reviewed.

The following rules have been written to aid the managers when faced with a new project:

```
(tag:+'euproject', ci:?p,...) -> (tag:+'interesting', ci:?p,...)

(tag:+'trustworthy', ci:?p,...), (tag:+'bigfunds', ci:?p,...)
-> (tag:+'interesting', ci:?p,...)

(tag:+'interesting', ci:?p,...), (?x :inChargeOf 'kiwici:projects')
-> (stag:'review'(?x), ci:?p,...)
```

A project that is officially sanctioned and sponsored by the EU is considered interesting enough. If a project is otherwise trusthworty (for example the team has already worked with the offering party in the past and was satisfied) and has a good funding to work with then it is considered interesting also.

Finally, all interesting projects, i.e. those fulfilling at least one of the above conditions, should be reviewed by whoever is in charge for projects in general.

1.4 Preliminaries

At this point, it should be clear that KWRL heavily relies on the ideas of the semantic web and rule engines to infer on and use the provided information in form of annotations.

As part of the semantic web and language for (meta)data representation, RDF is used by KWRL. Since the basic structure of RDF resembles simple natural language grammar, it can easily be used to formulate annotations for any kind of web content and due to its possible serialization in XML, it can be processed very easily. There are also some worked-out ontologies available built on RDF/S[2] for common real world models.[3]

Therefore, we will now have a look at RDF itself [3], (RDF) rules, inferencing and what existing RDF technology we can use for our goals.

[2]RDF in combination with RDFS

[3]e.g. Dublin Core (http://www.dublincore.org) for media metadata (author, publication date, language etc)

1.4.1 RDF

RDF is a framework for the description of resources.

A resource is anything which has a URI[4].

The essential building block of RDF data is an RDF *triple* or *statement* which is of the form *(s, p, o)* where s is the subject, p the predicate or property and o is the object of the statement.

Assume U is a set of URIs, L is a set of literals[5] and B is a set of blank nodes - basically anonymous placeholders for URIs or literals.

Then any RDF triple is an element of the set
$\{(U \cup B) \times U \times (U \cup L \cup B)\}$.

That is a subject can be either a URI or a blank node, a predicate must be a URI and an object can be a URI, a literal or a blank node.

A set of RDF triples is also called an RDF *graph* with the subjects and objects as nodes and the predicates as the (labeled) edges connecting them.

The set of all resources found in an RDF graph is called the vocabulary of the graph. There are also some predefined resources that can be used for basic statements called the RDF vocabulary, among others the predicate *rdf:type* to state that s is of the type o, or the resources *rdf:Statement*, *rdf:subject*, *rdf:predicate* and *rdf:object* which are self-explaining.

Further details go outside the scope of this thesis, for a good description of RDF see the RDF Primer [4].

RDF Schema

RDF Schema (often abbreviated RDF/S) extends the RDF vocabulary with some more predefined resources and predicates, introducing amongst other things the concept of classes and the vocabulary to describe the relations between classes, as well as descriptions of domains and ranges of predicates.

Additional predefined vocabulary by RDF/S includes:

rdfs:Class e.g. (ex:pet rdf:type rdfs:Class)

rdfs:subClassOf e.g. (ex:dog rdfs:subClassOf ex:pet)

rdfs:domain (predicate): if (s rdfs:domain o) then if (x s y) it follows that (x rdf:type o).

rdfs:range (predicate): if (s rdfs:range o) then if (x s y) it follows that (y rdf:type o).

Note that well-known ontologies build on RDF/S, such as OWL - the web ontology language [5].

However, as RDF/S is not immediately relevant to the thesis, it will not be discussed or explained further.

Refer to [6] for detailed information about RDF/S.

[4] uniform resource identifier

[5] There are different kinds of literals in RDF but this is out of scope of this thesis

1.4.2 RDF model theory

With the basic structure and syntactical concepts of RDF/S given, the RDF model theory (RDF MT) provides semantics and thus enables programs to use logic to query and infer on any RDF graph and triples.

Just like in formal logic an interpretation must be defined for the given vocabulary - in this case for the vocabulary of a given RDF graph. That interpretation, I, defines several sets and mappings between them, mainly the set IR of all given (RDF/S) resources, the set IP of properties - more or less defining the possible predicates of the statements - and an extension IEXT which maps IP into the powerset of $IR \times IR$ inducing a binary relation.

Basically, this binary relation says that for any interpretation I a given triple (s, p, o) is true iff s, p and o are indeed elements of their respectable sets defined by the interpretation and $< s, o >$ is in IEXT(p).

Apart from that, the model theory deals with the treatment of URIs and literals, e.g. saying a statement is false if it contains an ill-typed literal. Blank nodes (:_) are considered existential variables making its containing statement true if their exists at least one item in the vocabulary that when inserted for the blank node makes the statement true.

Furthermore, RDF MT defines the entailment between graphs: Let G and H be RDF graphs. Then G entails H if every interpretation that satisfies G also satisfies H. It immediately follows that every graph entails all its sub-graphs. This is called *simple entailment*.

There are also RDF-entailment, RDF/S-entailment and D-entailment (for datatype) that combine the simple entailment with the RDF and RDF/S vocabulary to give interpretations to RDF graphs, RDF/S graphs and datatypes. These combinations imply certain inference rules that are given and described in the section 1.4.5. See [7] for a complete description of the RDF model theory and the RDF/S semantics.

1.4.3 Rules on RDF data

Considering the logic aspects of RDF model theory, we can define rules on RDF graphs by checking for the existence of certain triples in their antecedents as described in the model theory.

Blank nodes can again be seen as existential quantifiers hence many rule languages for RDF data adopt its syntax.

As seen in the use cases, we demand user definable (i.e. rules not implied by the RDF MT) and declarative (i.e. rules that have statements instead of executable code as its consequence) rules.

We will call rule languages that are defined on RDF data *RDF rule languages*.

In a later section, several RDF rule languages that exist will be described.

1.4.4 Rule engines, inference

The inferencing itself is a task of the used rule engine which can have different algorithms implemented for it.

Two common inference methods that these algorithms are based on will be described next, namely forward and backward chaining.

Note that the examples are based on the use cases but have been modified to better illustrate how the inference methods work.

Forward chaining

Forward chaining is typically used for inferencing not immediately caused by a query. A new item added to the knowledge or rule base triggers the chaining process which just checks the existing data and rules for matches with the new piece of data or rule from the beginning of the chain, eventually producing new inferred data on its way or not.

[8] describes three algorithms for forward chaining: (semi-)naive forward chaining, the Rete algorithm and the magic templates algorithm.

The intuitive naive approach as applied to RDF statements and rules:

Consider the set of statements S and the set of rules R. Try to apply every element from S to every element from R, join the set of resulting inferred statements with S and recurse with the new set S' until a fixpoint is reached, i.e. no new statements are inferred. This approach is of course very simple and at the same time very inefficient. The most time-consuming and optimizable part of the algorithm consists of the recursion which is done with all statements combined, both the new and the old statements.

The *semi-naive* approach deals with this by avoiding to infer the same statement twice by e.g. the application of a rule twice when no data has changed in between the time of the applications.

Algorithm 1 Simple example program for (semi-)naive forward chaining

$interestingProject \leftarrow euproject$
$review(X) \leftarrow interestingProject, inCharge(X)$

$euproject.$
$inCharge(Anna).$

When applied to this example program, both naive chaining methods would infer the statements interestingProject and review(Anna).

The semi-naive algorithm, however, would, after the first application of the rules, only use the new fact *interestingProject* to fire the second rule only in the second run, thus reaching the fixpoint without inferencing *interestingProject* twice.

The Rete algorithm [9] makes use of another optimization technique: As can be seen in the use cases, rule bodies can contain instances of the same atom(s) in their bodies. This means that for these algorithms, those shared atoms are computed every time an affected rule is evaluated.

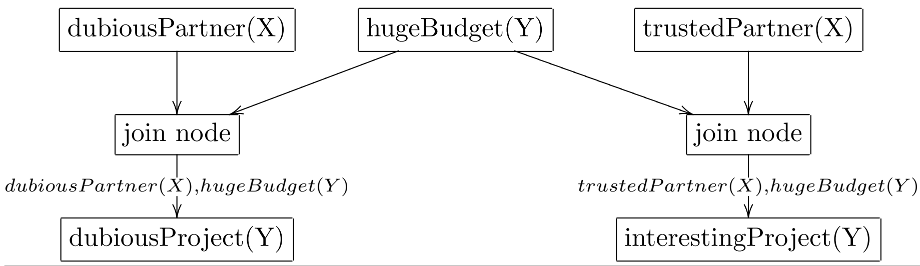
Rete solves this problem by creating two connected networks, an alpha network for the data and a beta network for the rules, where the beta network partly consists of join nodes that store i.a. these shared rule body atoms. A very short example of the beta network and the joining of rule body atoms is given in figure 2.

For an example of Rete being used for a big number of production rules, see [10].

The magic templates algorithm combines forward chaining with backward chaining which will be described next. The algorithm first rewrites the original

Algorithm 2 Very small example part of a Rete beta network and its correspondent rules

$dubiousProject(Y) \leftarrow dubiousPartner(X) \wedge hugeBudget(Y)$
$interestingProject(Y) \leftarrow trustedPartner(X) \wedge hugeBudget(Y)$



rules to an adorned version by using the fact that the query that should be answered by the program is known beforehand and thus the rules can be optimized so that the query can be answered more efficiently. For a detailed description of the magic templates algorithm, see [11], for an example see [8].

Backward chaining

Backward chaining is usually done by programs when answering queries given by the user. It is called 'backward' because when a statement has to be proven (a query has to be checked), there must be (an) assumption(s) and possibly rules that lead to data supporting that statement or to prove it wrong if a way cannot be found and so, the inferencer goes through the imagined graph of rules and data backwards.

According to [8], there are three basic algorithms for backward chaining: SLD resolution, its optimized version OLDT and the generalized BFP.

The Prolog way of backward chaining is called SLD-resolution. Following is a rather informal description of the algorithm, for an extensive formal one see [12]. With a specific query given, the (non-deterministic) algorithm tries to find a substitution of the variables in the program so that the program entails the query with the substitutions applied. For this, an SLD tree is constructed with the goal (modified query) as the root node and each edge representing the application of a possible rule for the goal with exactly one variable resolved resulting in a node that specifies the new goal for further resolutions. This is done until all branches of the tree are computed and the branches resulting in an empty goal define the correct answers to the query. It is important to note that, as the SLD tree can have infinite branches, some search strategies, such as the Prolog depth-first search, may loop indefinitely and may never find the answer(s) represented by the leaves of the finite branches.

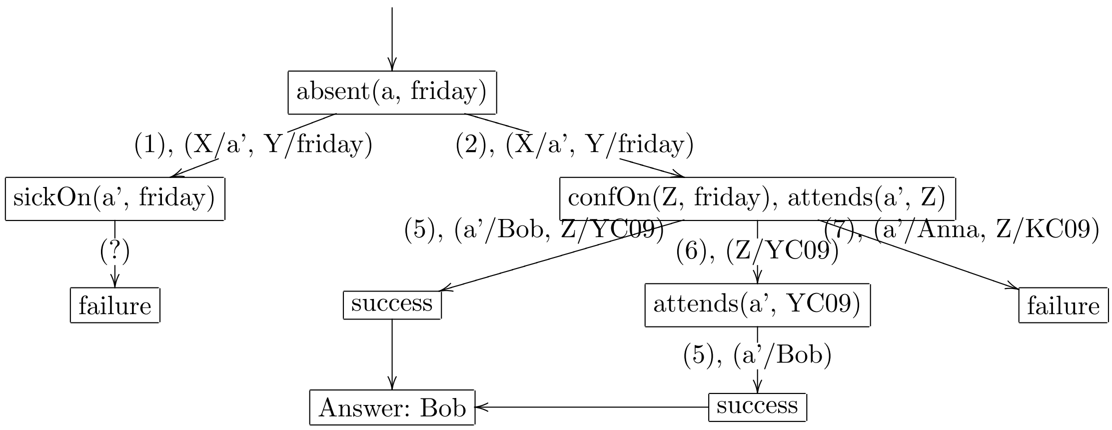
The OLDT-resolution (see also [13]) extends SLD-resolution by means of tabling. An OLDT structure consists of an SLD tree and two tables: a lookup table and a solution table. The main idea is that an SLD tree is constructed as described above but solutions for atoms are written in the solution table and can be referenced in the lookup table. The basic algorithm consists of three repeating steps:

Abbildung 1.1: Simplified example of an SLD tree

$query : absent(Answer, friday)$

1) $absent(X, Y) \leftarrow sickOn(X, Y)$.
2) $absent(X, Y) \leftarrow conferenceOn(Z, Y) \wedge attends(X, Z)$.

3) $sickOn(Bob, monday)$.
4) $attends(Bob, YC09)$.
5) $conferenceOn(YC09, friday)$.
6) $attends(Anna, KC09)$.
7) $conferenceOn(KC09, saturday)$.



1. Resolution: Add a new goal to the SLD tree.

2. Classification: Classify new node as solution, look-up or non-tabled goal.

3. Table update: Update solution or look-up table accordingly.

With this method, OLDT avoids most of the loops that are possible in SLD and thus has better termination properties. For a very good example see the transitive closure example in [8].

An interesting method that combines backward and forward chaining is the backward fixpoint procedure (BFP) [14]. It describes a backward chaining meta-interpreter that performs forward chaining on the object level. Four meta-predicates are defined of which three perform the computation of new facts in a forward chaining manner but only if a relevant query has been issued for those facts. The BFP can be seen as a generalization of several algorithms, f.e. the SLD-resolution.

Now, considering RDF data and rules, it would be only logical to state RDF data as triple(s,p,o) and thus a query in the BFP meta interpreter would be written as query(triple(s,p,o)). This leaves us with nested terms - in this case a predicate (triple) inside another (query) - which have to be 'unnested' to be properly evaluated by the actual implemented rule engine. One such procedure of 'unnesting' could be the kind of specialization described in [8]: applied to

Algorithm 3 Pseudocode of the BFP meta interpreter

A fact is only derived if a query has been issued by the user or the meta interpreter itself, there must of course be a rule with the fact as its head and the rule body must be fully evaluated.

$fact(Q) \leftarrow query(Q) \wedge rule(Q \leftarrow B) \wedge evaluate(B, \emptyset)$.

The meta predicate of evaluate takes two arguments, the left one represents the part of the rule body which is already evaluated (i.e. looked up) and the right part represents the part which has yet to be evaluated. The following couple of rules describe the left-to-right evaluation of the atoms of the rule bodies, creating subqueries on the way.

$query(B_2) \leftarrow evaluate(B_1, B_2) \wedge B_2 \neq (C_1 \wedge C_2)$
$query(B_2) \leftarrow evaluate(B_1, B_2 \wedge B_3)$

Finally, the evaluate predicate is defined.

$evaluate(\emptyset, B) \leftarrow query(Q) \wedge rule(Q \leftarrow B)$
$evaluate(B_1, B_2) \leftarrow evaluate(\emptyset, B_1 \wedge B_2) \wedge fact(B_1)$
$evaluate(B_1 \wedge B_2, B_3) \leftarrow evaluate(B_1, B_2 \wedge B_3) \wedge B_1 \neq \emptyset \wedge fact(B_2)$
$evaluate(B, \emptyset) \leftarrow fact(B)$
$evaluate(B_1 \wedge B_2, \emptyset) \leftarrow evaluate(B_1, B_2) \wedge B_1 \neq \emptyset \wedge fact(B_2)$

our example, the predicate query(triple(s,p,o)) would be rewritten to query-triple(s,p,o).

Now if we want to only query RDF triples, there is no need to have any other specialization of the query predicate. Thus, the query predicate could be replaced entirely by the query-triple predicate and, in the most simple example (applied to a ground RDF graph[6]), the query-triple predicate could just be evaluated like this:

$query - triple(s, p, o) \leftarrow triple(s, p, o)$

1.4.5 RDF/S inference

The RDF/S entailment rules (see RDF model theory) are briefly summarized in [7] chapter 7. They are listed here again (notation changed slightly) for completeness and reference.

Consider now an RDF graph G and triples in the form (sss ppp ooo). The rules are of the form (rule name: if G contains these triples → add these triples).

RDF rules

```
rdf1: (uuu aaa yyy) -> (aaa rdf:type rdf:Property)

rdf2: (uuu aaa lll) -> (_:nnn rdf:type rdf:XMLLiteral)
```

where lll is a well-typed XML literal and _:nnn is a blank node. These rules state that a predicate in an RDF triple is always of type rdf:Property and that a triple with a well-typed XML literal as its object implies that there is some node in the RDF graph that is of type rdf:XMLLiteral.

[6]An RDF graph with no blank nodes, i.e. an RDF graph with definite statements only

RDFS rules

```
rdfs1: (xxx aaa lll) -> (_:nnn rdf:type rdfs:Literal)
where lll is again a well-typed literal and _:nnn a blank node
```

This rule extends *rdf2* as it generalizes it to include all literals.

```
rdfs2: (aaa rdfs:domain xxx), (uuu aaa yyy) -> (uuu rdf:type xxx)

rdfs3: (aaa rdfs:range xxx), (uuu aaa vvv) -> (vvv rdf:type xxx)
```

These two rules describe the implications of the domain and range predicates. If a predicate p has a domain x and a range y, then for any triple (s p o) s must be of type x and o must be of type y.

```
rdfs4a: (uuu aaa xxx) -> (uuu rdf:type rdfs:Resource)

rdfs4b: (uuu aaa vvv) -> (vvv rdf:type rdfs:Resource)
```

A subject and an object of an RDF triple must be resources.

```
rdfs5: (uuu rdfs:subPropertyOf vvv),
(vvv rdfs:subPropertyOf xxx) -> (uuu rdfs:subPropertyOf xxx)

rdfs6: (uuu rdf:type rdf:Property) -> (uuu rdfs:subPropertyOf uuu)

rdfs7: (aaa rdfs:subPropertyOf bbb), (uuu aaa yyy) -> (uuu bbb yyy)
```

Here, the definiton of the subPropertyOf property (rdfs7) including its reflexivity (rdfs6) and transitivity (rdfs5) is stated.

```
rdfs8: (uuu rdf:type rdfs:Class) -> (uuu rdfs:subClassOf rdfs:Resource)
```

Every rdfs:Class is a subclass of rdfs:Resource.

```
rdfs9: (uuu rdfs:subClassOf xxx), (vvv rdf:type uuu)
-> (vvv rdf:type xxx)

rdfs10: (uuu rdf:type rdfs:Class) -> (uuu rdfs:subClassOf uuu)

rdfs11: (uuu rdfs:subClassOf vvv),
(vvv rdfs:subClassOf xxx) -> (uuu rdfs:subClassOf xxx)
```

The subClassOf property is defined in the same way that the subPropertyOf property was defined (see above).

```
rdfs12: (uuu rdf:type rdfs:ContainerMembershipProperty)
-> (uuu rdfs:subPropertyOf rdfs:member)

rdfs13: (uuu rdf:type rdfs:Datatype)
-> (uuu rdfs:subClassOf rdfs:Literal)
```

Other

It should be noted that there are other rules besides those mentioned, some of which deal with the meaning of blank nodes as stated in the RDF MT section.

Inference optimization

The rules implied by the RDF MT (see above) can be used with all the chaining procedures described above. However, since the rules are known beforehand, inference optimization on them is possible.

[15] describes several tests on large test data with RDF/S inference, stating that a slightly optimized version of the semi-naive forward chaining method seems to be the best choice for the task.

[16] mentions an RDF optimization technique: finding the closures of transitive properties in RDFS such as rdfs:subClassOf or rdfs:subPropertyOf through labeling schemes, using examples for the subclass relationship.

Three such methods are explained for directed acyclic RDF/S transitivity graphs:

Bitvector labeling scheme Each class is labeled with a bitvector with the length of the number of classes involved in the graph that is considered. The bitvector of a specific class is assigned a '1' bit for the position (in the bitvector) of the class itself and for all its superclasses respectively. This means that for each additional class that is added to the graph, all bitvectors have to be extended by one bit.

Prefix labeling scheme Each class is labeled with a prefix that contains the prefixes of all ancestors plus its own.

Interval labeling scheme Each class is labeled with an interval that is contained in all its ancestors but is disjoint with all intervals of its siblings.

Furthermore, the paper describes the Rete algorithm (see forward chaining) and its usefulness for RDF/S reasoning.

1.4.6 Negation in general

Apart from the method of inferencing that is done, the issue of negation is an important one to be considered.

The most important kinds of negation are classical negation and non-monotonic negation.

Classical negation can be seen as the negation found in formal logic such as description logic. For a statement to be considered false, it must be explicitely stated as being false. This implies that any statement which is stated neither positively nor negatively can not be evaluated.

Non-monotonic negation demands that if a positive statement is queried and neither its negative (as in classical negation) counterpart nor it itself can be found, it is assumed that the positive statement is false. This is the case e.g. in DBMSs (database management systems) which have a query for a non-existing table row return an empty (false) result.

Regarding the use cases, both kinds of negation are conceivable. Classical negation might be used to explicitely state that a project is not well funded.

With non-monotonic negation, a project which is not stated to be an euproject could be claimed to just not be an euproject at all.

How can negation be applied to RDF reasoners?

Statements negated with classical negation can be treated like positive statements since the negation can be seen as a symbol which marks a statement as false.

Non-monotonic negation can be implemented like this: have the reasoner look for the positive triple of the queried one (f.e. (s,p,o)), if it is found return true (false if NOT (s,p,o) was asked) and false (true if NOT (s,p,o) was asked) otherwise because in that case, the statement must be either negated (i.e. by classical negation) or non-existing, returning false (/true) in both cases as described above.

It is important to see that RDF neither provides for classical negation nor for non-monotonic negation.

Classical negation in RDF data

There have been several propositions how to include classical negation in RDF data. One such proposition is 'negation by declaration'[7] which basically requires putting an explicit NOT in front of the predicates of the negated statements, resulting in statements of the form *NOT_p(s, o)*.

While negation by declaration is just a suggestion for future RDF considerations, a whole extension of RDF[8] with both weak (i.e. non-monotonic) negation and strong (i.e. classical) negation has been suggested, too.

In this report, two kinds of negation are considered: weak negation (non-monotonic negation) and a weak form of strong negation in the form of negative polarity of atomic tags as annotation metadata in KWRL. See sections 3.4.6 and 3.4.4 for more details.

[7] see `http://www.agfa.com/w3c/2002/02/thesis/thesis.pdf` chapter 4

[8] Extended RDF (ERDF); both notions out of the scope of this thesis

Kapitel 2

RDF query and rule languages

In order to have a complete reasoning system for annotations, the RDF data has to be queried and rules have to be defined on it.

Following will be an overview of existing query languages for RDF and rule languages that can be used to define rules on RDF data. Note that a detailed description of the languages exceeds the purpose of this thesis.

2.1 RDF query languages

RDF queries are used to extract data from RDF graphs. Many languages have been created for that purpose with different features and syntaxes. This section presents a short overview of the different existing approaches.

2.1.1 Based on XPath [1] syntax

As its name suggests, GraphPath has been built for general data in graphs and finding paths through it. Using RDF graphs seems to be an obvious possibility for its application. GraphPath features a rule syntax that can be used to build rules which in turn can be evaluated by its reasoner implementation in Python. However, its basic query language syntax, closely resembling that of XPath, is strictly not rule-based.

Versa [17] is a pure RDF query language and, like GraphPath, has XPath-like syntax. It also features a core built-in function library for arithmetic functions, list and string manipulation etc. RxPath [18] is even syntactically equal to XPath and also a pure query language.

2.1.2 Based on SQL syntax

Another syntax to query RDF data follows that of SQL[1].

RDQL [19] is a relatively old query language in the style of SQL which inspired several similar and more advanced languages after it. Its queries have the well-known 'SELECT FROM WHERE' form.

[1]structured query language, common query language for relational databases

SeRQL [20] serves as the underlying RDF query language for the RDF framework Sesame[2]. Besides its more advanced (than RDQL) functions and operators and support for blank nodes, SeRQL does support construction of RDF data with *construct queries*.

SPARQL [21] is the newest one of them and has been a W3 recommendation since 2008. It supports blank nodes/variables, the construction of RDF data by inducing sub-graphs corresponding to subqueries and temporary tables in SQL, several functions, several different types of output, optional query parts. Therefore, SPARQL is very advanced in its features and well-known and used for RDF queries.

2.2 RDF rule languages

RDF rules define how new RDF statements can be inferred when they are applied to an RDF graph. There are several RDF rule languages, often enough in RDF syntax themselves. Following are some chosen examples for different approaches to RDF rules.

The most prominent rule languages for RDF are N3 and Triple.

N3 [22] is a language in RDF syntax. Even though N3 was originally intended to be an alternative to the RDF/XML notation (thus Notation 3), it is possible to create simple RDF rules; since N3 also supports variables and logical quantification. An extra feature of N3 is nested statements, adding to its expressiveness. N3 is e.g. used by the forward chaining reasoner CWM which will be discussed in the inference section.

Triple [23] on the other hand defines its own syntax which combines querying and inferencing, featuring the notion of models (i.e. worlds) for RDF data, allowing for different rule sets in different models. There is a subset of Triple in RDF syntax which does not include quantifiers. Inference for Triple rules is officially realized by an XSB (prolog) implementation [3]. It should also be noted that Triple is not to be confused with NTriple which is a subset of N3.

Corese [24] uses the graph structure of RDF data and translates RDF graphs to CGs (Conceptual Graphs, see paper). Its RDF/XML syntax is wrapped around SPARQL syntax for queries and construction to form rules which can be optimized because of the CG structure and run by its own reasoner implementation. Corese even features the notion of sources for RDF triples and graphs[4].

According to the Semantic Web Stack, the official language for rules is RIF [25] (rule interchange format). As a general XML rule language, it offers RDF compatibility and should be used by a forward chaining reasoner. As a universal exchange language for rules, RIF is not primarily relevant but when such compatibility is needed we could later on write a converting tool from and to KWRL.

A rule language similar to RIF is RuleML [26] which can inter-operate with RDF, too.

RDFLog [27] extends datalog and features the arbitrary use of quantifiers for rules and the construction of blank nodes.

[2] `http://www.openrdf.org/`

[3] see homepage `http://triple.semanticweb.org/`

[4] compare to the section about named graphs in the KWRL chapter

2.3 RDF inference

There are many RDF reasoners exisiting today and a full survey of all of them would go beyond the scope of this thesis. The reasoners described in this thesis are representatives of four different categories of inference.

Two of those basic categories are backward and forward chaining (see 1.4.4).

2.3.1 Backward chaining

Backward chaining, as already stated, is mostly used for query answering but it is a viable option for inferencing, too. We will have a look at two inferencers in this category.

One such inferencer is GraphPath [28] which was already described above.

The other backward chaining inferencer is Euler [29] which, as is hinted at by its name, features Euler path detection to ensure better termination properties. Its official title also includes 'proof mechanism' stating that Euler is typically used for explanations of its backward reasoning. It is implemented in the most common programming languages and uses N3 (see above) as its underlying RDF query/rule language.

2.3.2 Forward chaining

Forward chaining is the most common approach for inferencing since the basic idea is that when new data is added to the RDF graph, the statements could induce new statements[5]. For this, there are different algorithms and some reasoners use the Rete algorithm or one of its variations (e.g. [30] [31]) to optimize the rule matching process.

CWM [32] is the forward chaining counterpart to Euler, endorsed by the W3C. Although not its primary purpose, it has been further developed to inference on RDF data and rules with N3 as an underlying language just like Euler.

Sesame, the open source RDF database framework, has a builtin RDFS reasoner and with OWLIM [33] it has an extension for user definable rules. OWLIM comes with predefined rule-sets, working on RDFS and a subset of OWL.

Bossam [34] is based, like many others, on the Rete algorithm and reads RuleML rules and RDF/OWL data. It is written in Java and also supports negation.

2.3.3 Forward and backward chaining

Just like CWM/Euler combines forward and backward reasoning, as we just saw, there are systems that use both chaining methods.

One good example is the Jena RDF framework. Apart from the Java framework itself for RDF, Jena includes both a forward and a backward chaining inferencer [35] which seem to be combineable just like the CWM/Euler are. It is not surprising that Jena's forward chaining inferencer also uses the Rete algorithm.

[5] f.e. by the rules implied by the RDF MT

2.3.4 Special cases: Translation to SQL/CG

Apart from the chaining methods, there are other methods to infer on RDF data and rules. Two variations are described here as a more detailed description of alternative methods would go beyond the scope of this thesis.

Well known for their database software, Oracle has some databases specifically built for RDF data storage. One Oracle product also includes an inferencer which can evaluate SQL-like (syntactically) rules. See [36] for a demonstration of their work.

The other special case is Corese [24] (see above). The RDF graph structure here is also translated, but not to SQL, but to a different graph structure, the conceptual graph. See the paper for details.

2.4 Summary

In this section the most relevant features of the reasoners that were mentioned are compared and the reasoners are discussed regarding their suitability for the project.

These features are:

Rule language syntax What rule language syntax does the reasoner use as input for the rules? Depending on the syntax that we will use, a conversion tool might be necessary.

Algorithm What type of algorithm is used for the inferencing process?

Semantics What kind of semantics is assumed for the rules and the inferencing?

User definable rules The rules should also be user definable as apparent in the use cases, RDF/S reasoning is not enough for our purposes.

Monotony/negation What kind of negation is supported and, related to that, is the logic assumed to be (non-)monotonic?

Debugging/output Another issue is the question of debugging the rule engine. It would be nice to have the reasoner output some kind of explanation for the inference process so that it can be seen where possible errors have occurred.

Implementation What language(s) is the reasoner implemented in?

RDF/S inference Is RDF/S inference already implemented in the reasoner? If not, it can still be implemented by user-defined rules.

Open source Is the software open source?

2.4.1 Backward chaining

	GraphPath	Euler
rule language syntax	similar to XPath	N3
algorithm	plain backward chaining	backward-forward-backward
semantics	declarative rules	declarative rules / N3Logic
user definable rules	yes	yes
monotony/negation	?	non-monotonic negation
debugging/output	no	yes
implementation	Python	Java(script), Python, C#, Prolog
RDF/S inference	no	no
Open source	yes (GPL)	yes (W3C)

GraphPath features an XPath-like structure and XPath-like queries. GraphPath uses declarative and user defined rules which is perfect for our purposes but lacks the output and RDF/S inference options. Furthermore, it is not clear what kind of monotony and/or negation GraphPath assumes, without any declaration, monotony and non-monotonic negation might be the most probable option.

Euler seems like it has many useful features, it uses N3 as a very common rule language and N3Logic[6] for the semantics, it supports declarative and user defined rules and it can even produce debugging information which we would also want. Its combination of chaining processes[7] once again raises the question of how to use and combine forward and backward chaining effectively. Euler assumes a monotonic logic and non-monotonic negation. RDF/S inference is not supported. Another interesting thing about Euler is its considerably wide range of implementations, making it relatively portable. Combined with CWM, it might be a powerful reasoner and, with its proof output, very helpful for developers and users.

2.4.2 Forward chaining

	CWM	OWLIM (Sesame)	Bossam
rule language syntax	RDF/N3	RDF	RuleML
algorithm	FOPL forward chaining	Total materialization	Rete
semantics	declarative rules / FOPL	declarative rules	declarative rules
user definable rules	yes	yes	yes
monotony/negation	non-monotonic negation	non-monotonic	strong and weak negation
debugging/output	some	no	yes
implementation	Python	Java	Java
RDF/S inference	no	yes	yes
Open source	yes (W3C)	yes (LGPL)	proprietary license

As already mentioned, CWM is interoperable with Euler and could thus provide a powerful reasoner using standard FOPL (first order predicate logic). Its features mostly correspond to those of Euler, with the main and obvious difference being that CWM uses a normal forward chaining algorithm.

OWLIM is mostly interesting for its usage in Sesame and its strategy of

[6] see `http://www.w3.org/DesignIssues/N3Logic`

[7] Prolog backward chaining, forward meta-level reasoning, backward proof construction

total materialization[8] which is claimed to be very efficient with some optimization. OWLIM, as its name already suggests, does infer on a subset of OWL thus encompassing RDF/S inference. Declarative and user defined rules are, again, supported, too. Interstingly enough OWLIM assumes a non-monotonic world which we do not necessarily want (see the preliminary and the extensions sections). It should be noted that there are two versions of OWLIM available. An open source version called SwiftOWLIM licensed under the LGPL and a commercial version called BigOWLIM.

Bossam is the only forward chaining reasoner inspected here which explicitely uses the Rete algorithm (see Preliminaries). Like OWLIM, it uses both RDF/S and user defined rules. Bossam also claims to support both non-monotonic negation and classical negation which probably comes with the usage of RuleML and the extension of RDF with negation as described in the paper 'RDF/RuleML interoperability'[37]. Written in Java, Bossam can also call Java objects from the rule heads and bodies thus making the rules more or less semi-declarative which is not needed in our project. A bit of output can be produced by an 'explain' feature.

2.4.3 Forward and backward chaining

	Jena
rule language syntax	Java rule objects
algorithm	Rete and tabled datalog (SLD-based)
semantics	Production rules
user definable rules	yes
monotony/negation	non-monotonic
debugging/output	yes
implementation	Java
RDF/S inference	yes
Open source	yes

Jena extends the Java idea of Bossam even more since all Jena rules are actually Java objects and thus they are not really declarative. The rules can be user defined however and Jena has various specialized reasoners implemented, an RDF/S reasoner is one of them. Like OWLIM, Jena assumes a non-monotonic world and like some other reasoners can print more or less helpful output. Of course the most interesting feature of Jena is the explicit combination of forward and backward chaining with Rete as the forward chaining method (described in the preliminaries section) and tabled datalog (similar to Prolog's SLD) as the backward chaining method. Jena thus remains an interesting system that might be capable of being extended for our purposes.

[8]practically a synonym of forward chaining

2.4.4 Other approaches

	Oracle 11g	Corese
rule language syntax	RDF translated to SQL	SpaRQL constructs as RDF/XML rules
algorithm	forward chaining (SQL joins and sorts)	Conceptual Graphs
semantics	(modified) fixpoint	declarative rules / fixpoint
user definable rules	yes	yes
monotony/negation	monotonic/no negation	monotonic
debugging/output	yes	yes
implementation	DB application	Java
RDF/S inference	yes	yes
Open source	no	yes

Both reasoners in this category deserve bonus points for creativity.

Oracle's 11g database features inference on both user defined and RDF/S rules in a monotonic environment (which illustrates the monotony example of databases) and explanations are also possible through queries. Although it assumes a fixpoint semantics, the inference process can be controlled by the user. A more detailed description of the system is beyond the scope of this thesis, however, Oracle's solution remains an interesting one for RDF reasoning.

Corese, on the other hand, uses SpaRQL constructs as rules on RDF data. Its algorithm will also not be further examinated in this thesis but it would be interesting to know whether it could be used for our reasoner. Other than that Corese seems to support all relevant features that we described at the beginning of this part. Furthermore, another relevant idea of Corese consists of the proposition to add sources to RDF triples and whole RDF documents which we would also want in our social context.

2.4.5 Overall

After looking at these chosen approaches for RDF reasoning, it appears that there are a few interesting reasoners that could potentially be extended and used for the project. Many reasoners even support declarative and user defined rules. However, none of these reasoners were written with the background of a semantic wiki in mind. For example, we would want to allow inconsistencies when useful, data and inference should also be traceable to the responsible user and many more (social) features should exist, in turn demanding uncommon features of the underlying reasoner such as f.e. reason maintenance. See the KiWi papers for more details.

Not having found any immediately usable results, we now specify a basic rule-based language for annotations of content items as hinted at in the use cases that could be extended to support those social features if paired with a suitable reasoner.

Kapitel 3

The rule language KWRL

KWRL (KiWi [2] Rule Language) is a rule language for reasoning with annotations. It is based on RDF, the formalism in which the annotations are expressed, and allows either direct referring to RDF triples or a higher-level, less technical form of annotations. In the KiWi system, annotations are stored in knowledge spaces.

3.1 Metadata

There are three kinds of metadata in KiWi - atomic tags, structured tags, and RDF triples. Both atomic tags and structured tags are internally represented in RDF (see later section) but this internal, admittedly technical representation, does not have to be seen by KWRL programmers.

3.1.1 Atomic tags

Atomic tags are identified with a unique URI. Each atomic tag has at least one label, i.e. the 'tag' in informal language. This allows for tag normalization, i.e. a tag can have more than one synonymic labels. (e.g. 42, fortytwo, forty-two) Its basic syntax is:

```
AtomicTag ::= URI | Label
```

Note that for the examples, only labels are used.

Examples: user, reviewed, bug, trick17

3.1.2 Structured tags

Structured tags in this thesis are simplified because the details of structured tags are not important for this report and thus, structured tags can be seen as 'atomic tags with arguments'. (See e.g. the respective part of [38] for what structured tags can be.) The arguments of a structured tag are atomic tags. The 'structured label' of a structured tag therefore has the form '$label(label_1, label_2, ..., label_n)$'. The list of arguments can be ordered or unordered; the syntax may be the same for these two options, however, see the semantics section for a discussion about the possible meanings of structured tags with either option. Furthermore, each structured tag has a URI with which it is identified. Each $label_i$ has a URI as well (the URI of the atomic tag which the label belongs to, therefore it possibly has to be disambiguated).

Examples:

Consider a wiki page that describes a proposal for the acquisition of new chairs for the software development team. Bob wants to vote for the acquisition of exactly 5 of these new chairs. He annotates the page with the structured tag vote(5).

Now consider a meeting of the team. The team goes through and discusses several pages describing bugs found in the recent software. On one of the pages of code, Bob notices that Alice and him were working on the particular piece of code, thus the page is annotated with the structured tags check(Alice) and check(Bob), meaning of course that both Alice and Bob should check that code. On the same page, a structured tag bug(34587) was annotated (possibly even by rule inference) stating that the bug found in the code has the (ticket) number 34587.

3.1.3 RDF triples

For RDF theory, see the preliminary section. To briefly recapitulate basic RDF triple construction: A resource is anything that has a URI. RDF triples are statements about resources in the form 'Subject Predicate Object', where subject and predicate are resources identified with their URI, object is either a resource or a literal. To allow for a more compact syntax, URIs can be specified using a namespace. Then RDF triples are expressed as 'ns1:subject ns2:predicate ns3:object' or 'ns1:subject ns2:predicate literal', where ns1, ns2, and ns3 are namespaces specified elsewhere and subject, predicate, object are name strings and literal is a string.

Note that in the examples, we use sketches of a KiWi RDF vocabulary, specifying relevant concepts for the project. For further information, see the respective reports.

Examples:

kiwi:pageXYZ rdf:type kiwi:ContentItem
ex:car skos:narrowerThan ex:vehicle

3.2 Knowledge spaces

3.2.1 Provenance, grouping and named graphs

There are two basic motivations for having the concept of a knowledge space in KWRL and KiWi: tracking of provenance and grouping of related triples.

Provenance, sometimes also refered to as lineage, is an often required feature of knowledge bases where the origin of data is important for example to establish trust. Recently, provenance has been recognized as an important missing feature in the area of Semantic Web [39] and it has been argued [40] for the benefit of adding a social dimension to the pure technical notion of provenance which has been endorsed by [41].

A rudimentary concept of provenance has been around Semantic Web for a long time in the form of *contexts* and *quads* which allow referring to a triple by assigning it a unique identifier. A more formal and standards-based approach was taken by Carroll et al. [42, 43] by defining so called *named graphs*. Recently, two extensions of named graphs have been proposed: networked graphs by Schenk and Staab [44] and RDF/S Graphsets by Pediaditis et al. [45]. Networked graphs allow for named graph definition by a combination of SPARQL queries and explicit listing of triples. RDF/S Graphsets are proposed to solve the problem of "triple ownership" in presence of RDFS reasoning and updates. An orthogonal approach to provenance in semantic web is offered by Ding et al. [46] in the form of RDF molecules. In the named graph approach it is mainly the application or the user who decides how to create named graphs. RDF molecules allow for automatic, "lossless" decomposition of RDF graphs into "molecules" which are connected-subgraphs of the original graph. For a comprehensive overview of state of the art in provenance and categorizations of the approaches see [41, 47].

In KiWi, the focus is not on provenance in general but on some specific issues regarding its social features and reasoning. In terms of [41], we are specifically interested in *data* provenance (as opposed to workflow provenance) which is concerned with creation of single pieces of data, why-provenance which *"represents the origins that were involved in calculating a single entry of a query result"*[41] and how-provenance which describes the role of origins in the calculation.

Grouping of related triples can be seen as a pre-requisite for provenance so that it is possible to avoid duplication of provenance data by storing it with each triple separately. Also, there is often a natural correspondence between a domain concept and an RDF graph which is mostly standalone. This is not unlike the correspondence of real-world concepts and its counterparts in an object-oriented language. However, for the grouping to be really useful, it should go beyond simple RDF graphs and should allow also nested groups. Such a concept of grouping would allow storing of provenance information on different "levels of detail" (a term for granularity of data item used by [47]). Moreover, grouping would help the KiWi application to work more on the domain-level, conceptually closer to users and to do so in a more interoperable way.

For example if there is a "tag" concept which the user intuitively understands and the KiWi application knows that the tag concept translates to a specific set of say six triples then deleting a tag translates to deleting the corresponding six triples. In this case KiWi uses its application-specific domain knowledge to process the update. In contrast, if the "tag" concept corresponded to a group then deleting a tag would translate to deleting the whole corresponding group whatever it consists of thus giving more power to the information publisher and alleviating the need of hard-coding domain specific knowledge.

We can distinguish two kinds of groups: "Java-package-like groups" and "subgraph-like groups". If a package-like group contained subgroups then members of the subgroups would not be seen as members of the supergroup. On the other hand, "subgraph-like groups" would behave like subgraphs - if a graph contains a subgraph then the graph contains also everything that the subgraph contains. Our aim is to avoid duplication, to provide a flexible "level of detail" and to be able to specify different rules for different groups. Therefore the subgraph semantics is more desirable – what is said about the encompassing group holds also for all its members and members of their members transitively thus it avoids the necessity to assert the fact for each of them and enables reasoning all triples not only the one directly contained in a group.

A kind of subgraph semantics is already provided as part of the named graph proposal [43] in the form of *rdfg:subGraphOf* and *rdfg:equivalentGraph* properties. An advantage of named graphs over other alternatives is their standards-based approach, wide acceptance and tool support: [48] presents NG4J – a named graph implementation based on Jena, SPARQL [49], the query language for RDF, has named graph support, also the NEPOMUK project uses and extends named graphs in an interesting way [50] for the use in the Social Semantic Desktop. On the other hand there has been some confusion around named-graphs nesting and the rdfg:subGraphOf relationship the result of which maybe was that the relationship with the proposed semantics has not been implemented in any tool yet (to the best of our knowledge).

3.2.2 Knowledge spaces in KWRL

Recall that in KiWi and KWRL, knowledge spaces are mainly used for tracking the provenance of triples and for grouping of related triples. Consider for example a knowledge space for all annotations, a knowledge space for all projects of the software development team or even knowledge spaces for each one of the projects.

Knowledge spaces in KWRL are modeled using named graphs. Named graphs are used as the initial, lowest-level grouping mechanism for RDF triples. Nesting is supported using the *rdfg:subGraphOf* property.[1] Named graphs introduce rigid names for RDF graphs, i.e. named graphs, once asserted, can not be renamed and thus two named graphs using the same name but containing different triples are inconsistent. This approach allows to distinguish between different copies of the same RDF graph. The advantage is that each of the copies may be associated with a different provenance information. The name of a graph is a unique URI. Named graph itself is a pair $\langle n, g\rangle$, where n is the name and g is the associated RDF graph. It is worth noting, that "the named graph itself, rather than the RDF graph it intuitively 'names', is the denotation of the name. We consider the RDF graph to be related to the named graph in a way analogous to that in which a class extension is related to a class in RDFS."[43] It is this intensional modelling that allows to distinguish between different copies of an RDF graph.

Using named graphs, a couple of taggings can be specified the following way (in TriG syntax):

```
:t1g {
 kiwi:ci1 kiwi:hasTagging kiwi:t1t
 kiwi:t1t kiwi:hasUser kiwi:u1
 kiwi:t1t kiwi:hasTag kiwi:t1
}

:t2g {
 kiwi:ci1 kiwi:hasTagging kiwi:t2t
 kiwi:t2t kiwi:hasUser kiwi:u2
 kiwi:t2t kiwi:hasTag kiwi:t2
```

[1] Please note, that there still is some confusion about the precise semantics as proposed in [43], therefore it needs to be verified and in the worst case replaced by a KiWi specific property.

```
}

:t1g rdfg:subGraphOf kiwi:taggingKnowledgeSpace
:t2g rdfg:subGraphOf kiwi:taggingKnowledgeSpace
```

The named graph $:t1g$ consists of triples of tagging $t1t$ by user $u1$ of a content item $ci1$. The named graph $:t2g$ consists of triples of tagging $t2t$ by user $u2$ of a content item $c1$. Both named graphs $:t1g$ and $:t2g$ are subgraphs of the $kiwi:taggingKnowledgeSpace$ named graph.

3.3 Annotations

Annotation in KWRL means an association of metadata with a content item (can be generalized). Precisely, annotation is an association between the following information:

metadata (that is one of atomic tag, structured tag, RDF triple)

content item the content item to which the annotation belongs

authors that is a list of CIs of type user (the user who created the annotation and all users who updated the annotation)

date of creation (plus possibly the date of last update, ...)

marker to distinguish between user-defined, rule-derived, and system annotations

agreements that is a list of user-agreement value pairs (agreement value is a number between 1 for agreement, -1 for disagreement, thus 0 is expressed non-interest)

tracking consisting of the user-origin and content-item-origin of the annotation, i.e. a set of sets of content items that the annotation transitively depends upon

knowledge space that is the KiWi knowledge space the annotation belongs to

We call the information that the association consists of its *attributes*. We define two types of attributes: *system* attributes and *common* attributes. Every attribute belongs to exactly one of these types.

System attributes can be updated only by the system. They are comparable to system attributes found in every day computing, f.e. EXIF data ([51]) for pictures taken by a camera or file attributes like date of creation, author etc. The system attributes in KWRL are the following:

- authors
- date of creation
- marker
- agreements

- tracking

Since only the system itself can create and modify them, the attributes can not be set by the user, only queried. See the annotation construction syntax part to see how those attributes are set by the system.

Common attributes are all attributes that are not system attributes. They can be accessed both by the system and the KWRL programmer. The common attributes are:

- metadata
- content item
- knowledge space

An annotation tuple is a tuple of keyword value pairs in the following form: *(*$attr_1$: $value_1$, $attr_2$: $value_2$, ..., $attr_n$: $value_n$*)* where $attr_i$ are the keywords representing the attributes and $value_i$ are their respective (literal) values as described above. Note that in the following sections, we often treat 'annotation' and 'annotation tuple' as the same if it is clear whether the abstract notion of an annotation or a concrete representation of it as a tuple is described.

See the following tables for a summary of the annotation (tuple) attributes and their properties:

Attribute name	Attribute type	Keyword
metadata	common	tag — stag — triple
content item	common	ci
authors	system	author
date of creation	system	date
marker	system	marker
agreements	system	agreements
tracking	system	origin
knowledge space	common	ks

Attribute name	Value
metadata	atomic/structured tag or RDF triple
content item	(URI of) a content item
authors	list of content items of type user
date of creation	a date
marker	'user' — 'derived' — 'system'
agreements	list of keyword value pairs with user:agreement (see above)
tracking	set of sets of (URIs of) content items
knowledge space	(URI of) a KiWi knowledge space

Note that for atomic tags, we add the polarity property as described in the reason maintenance report, so that we prefix negative (atomic) tags with a '-' symbol and we explicitely prefix the positive ones with a '+' symbol.

Syntactically, an annotation (tuple) is represented like this:

```
AnnotationTupleDef ::= "(" AttrsDef ")"
AttrsDef ::=  CommonAttrsDef "," SystemAttrsDef
  CommonAttrsDef ::= MetadataDef "," CIDef "," KnowledgeSpaceDef
    MetaDataDef ::= AtomicTagDef | StructTagDef | RDFTripleDef
```

```
      AtomicTagDef ::= "tag:" ("+" | "-") URI
      StructTagDef ::= "stag:" Label "(" LabelListDef ")"
        LabelListDef ::= Label {"," Label}
      RDFTripleDef ::= "triple:" URI URI (URI | Literal)
    CIDef ::= "ci:" URI
    KnowledgeSpaceDef ::= "ks:" URI
  SystemAttrsDef ::= AuthorsDef "," DateDef ","
                     MarkerDef "," AgreementDef "," OriginDef
    AuthorsDef ::= "author:" "(" AuthorListDef ")"
      AuthorListDef ::= URI {"," URI}
    DateDef ::= "date:" Literal
    MarkerDef ::= "marker:" Literal
    AgreementDef ::= "agreements:" "(" [AgreementListDef] ")"
      AgreementListDef ::= URI ":" Value {"," URI ":" Value}
    OriginDef ::= "origin:" "(" [OriginList] ")"
      OriginList ::= URI {"," URI}
```

Literal is a literal. Labels are literals with type string. URIs are enclosed by ', too. The date literal is yet undefined but may be taken from the (time)dates used by f.e. SQL. For the marker literal see the table above. The Value for agreements is specified (as seen above) by a numeric value between 1 and -1.

For all following syntax, we will use a syntax quite similar to the above one with the following modifications:

- Addition of variables (in the form of ?x)
- Using IDs instead of URIs for generality and extensibility (for now, every ID is a URI)
- Polarity optional for atomic tags (consider a query for *all* atomic tags, both positive and negative ones)
- Usage of labels for atomic tags for all examples

Examples for annotation tuples:

```
(tag:+'review', ci:'kiwici:TheProject', ks:'kiwi:ProjectSpace',
author:('kiwiusr:TheManagement'), date:'01-01-2009', marker:'user',
agreements:(), origin:())
```

The specified annotation tuple represents an annotation that has an atomic (positive) tag review attached to the content item of kiwici:TheProject. The attributes further specify that the annotation is stored in the kiwi:ProjectSpace knowledge space and has been created by the user kiwiusr:TheManagement on the first of January in 2009. The annotation is also purely user-defined. One interpretation of this annotation tuple would be that the management of the group using the system wants to say that the project (which the content item TheProject is about) should be reviewed. Note that no user has yet stated an opinion of his level of agreement with this annotation and that, instead of the atomic tag, a structured tag could be used to specify f.e. who exactly should review the project. Thus, the modified annotation tuple would read like this: (stag:'review'('Bob'), ...).

```
(stag:'knows'('Java'), ci:'kiwiusr:Bob', ks:'kiwi:UserSpace',
author:('kiwiusr:Alice'), date:'01-01-2009', marker:'derived',
agreements:('kiwiusr:Chuck':0.8),
origin:('kiwici:Programming','kiwiusr:Bob'))
```

This slightly more complicated annotation tuple represents an annotation that has a structured tag knows(Java) attached to the content item of kiwiusr:Bob. It is stored in the kiwi:UserSpace and has as author kiwiusr:Alice. This time, another user, kiwiusr:Chuck, has set an agreement value for this annotation and the system has derived an origin of the annotation as specified, thus having as marker derived. The date is the same as for the first example. As for a possible interpretation of this annotation: The common attributes simply specify that it is said that Bob knows Java. In this case, however, the annotation has an origin, i.e. the system derived this annotation from the content items about Programming and Bob himself as the user origin. Since the rule was written by Alice, she is set as the author of the overall annotation. The third specified user, Chuck, mostly agrees that Bob does indeed know Java, he does not agree a hundred percent though, perhaps because he thinks that Bob should learn more about certain fields of applications of Java.

3.4 Rule syntax - rule body

KWQL [52] [53] has been designed as the query language for the KiWi project.

It is important to note that KWQL is a rule-based query language over content items, querying text and metadata and featuring simple construction, whereas KWRL is a pure rule language over annotations (and RDF triples) assigned to content items with full construction of annotations and RDF triples.

KWRL is not a replacement for KWQL but as KWQL includes the notion of querying simple tags for content items, annotations in KWRL can be seen as an extension of simple tags for content items and thus KWRL can be seen as a rule language extending KWQL's notion of rules over content item metadata.

3.4.1 Annotation query

Consider now a multitude of annotations as described above in the KiWi system. It is feasible to imagine that one would want to search for one or more of these annotations, f.e. to see what projects should be reviewed or what user in the team knows what programming language according to the annotations.

We call a query for annotation(s) (tuples) stored in the (KiWi) system an *annotation query*. It is possible to query all annotation attributes (i.e. both system and common attributes) except for the marker attribute.[2]

3.4.2 Basic annotation query (BAQ)

The most simple type of annotation query consists of querying the annotations for only one of their attributes. We call this type of query a basic annotation query, or *BAQ* for short. Consider the review annotation from the examples.

[2]Consider the following rule sketch:
$(tag :?x\ marker : derived) -> \ (page : specialPage\ tag :?x)$.

An intuitive way to look for all annotations that have as metadata the review tag is tag:'review' and this is exactly one example of a BAQ. As will be seen in the query syntax (see below), a BAQ consists of exactly one keyword:value pair which specifies which attribute is queried (keyword) and which value it should have.

Consider now that we want to query all creation dates of the stored annotations. For this, we introduce variables. In this case, the BAQ would read date:?x where ?x would be bound to each annotation creation date value upon execution of the query.

It should be noted that the list attributes (author, agreement, origin) are a special case for variables: Consider the three cases author:?x, author:(?x) and author:(?x,?y). In the first case, it comes as natural that the whole list (of authors) is matched and returned. If the lists are ordered, then it can be argued that the first variable in the list returns the first element and so on; thus, for ordered lists, in the second case the query would return the first element of the list for ?x and in the third case it would additionally provide the second element for ?y. If the lists are unordered, the second and third cases are not trivial.

Note also that more than one variable in a BAQ is possible for RDF triple metadata since we allow variables in all positions - subject, predicate and object (f.e. triple: ?x ?y ?z).

BAQs are used as basic building blocks for the more complicated atomic and compound query syntax but also show how, instead of specifying the value of each attribute in an annotation query, only few (or even just one) attributes can be used for queries if it is clear what the resulting short form is supposed to mean when evaluated. See the syntactic sugar section for more details about this.

3.4.3 WHERE clause

Let us consider the above example of a query for all annotation creation dates: date:?x. What if want to have annotations that were created f.e. after a specific date, say after the first of January 2009?

For this and other arithmetics or string operations, we introduce WHERE clauses that are comparable to those of S(PAR)QL.

Although not yet fully worked out here, example clauses could be
$WHERE\ ?x >' 01 - 01 - 2009'$ (see above) or
$WHERE\ ?n <= 3$ (arithmetics) or even
$WHERE\ ?x =' /[Bb]ob/'$ (string operations/regex).

It should be clear that the variable used in a WHERE clause following a query must be in the query itself.

Extended WHERE clauses could be used for the extended (compound) annotation query, see later section for the compound query.

3.4.4 Atomic annotation query (AAQ)

An atomic annotation query (AAQ) is a complete query for annotations in the sense that in an AAQ, more than one attribute of an annotation can be queried and thus, every attribute value can be specified. An AAQ is therefore just one BAQ or a list of BAQs that are all combined and the resulting query checked against the annotations stored in the system as usual.

Consider the running example with the programming language knowledge of the members of the team. It is feasible to imagine that one would want to know what programming languages Bob knows: the resulting AAQ would read (stag:'knows'(?x),ci:'kiwiusr:Bob'). With the appropriate WHERE clause as described above, one could even have all programming languages for all users with an AAQ similar to
(stag:'knows'(?x),ci:'?y') $WHERE\ ?y =' /kiwiusr : .*/'$.

Following is the syntax for AAQs (and thus BAQs) including the WHERE clauses where *Constraint* is an expression as hinted at above.

Note that the way it is designed allows the user to specify one or more attributes more than once and it also allows nesting. The basic syntax for the attributes (keyword/value) has not changed compared to the basic annotation tuple syntax but the additions mentioned in the annotation section are applied.

```
AAQ ::= BAQ ["WHERE" Constraint] | "(" BAQList ["WHERE" Constraint] ")"
  BAQList ::= BAQ "," AAQ {","  AAQ}

BAQ ::= CommonAttrsExp | SystemAttrsExp

CommonAttrsExp ::= MetadataExp | CIExp | KnowledgeSpaceExp
SystemAttrsExp ::= AuthorsExp | DateExp | AgreementExp  | OriginExp

MetadataExp ::= AtomicTagExp | StructuredTagExp | RDFTripleWithKeywordExp

  AtomicTagExp ::= "tag:" [Polarity] IDV
    Polarity ::= "+" | "-"
  StructuredTagExp ::= "stag:" NameV "(" IDListV ")" | "stag:" Variable
  RDFTripleWithKeywordExp ::= "triple:" RDFTripleExp
    RDFTripleExp ::= Subject Predicate Object
      Subject ::= Variable | Resource
      Predicate ::= Variable | Resource
      Object ::= Variable | Resource | Literal

CIExp ::= "ci:" IDV

KnowledgeSpaceExp ::= "ks:" IDV

AuthorsExp ::= "author:" "(" IDListV ")"

DateExp ::= "date:" Literal

AgreementsExp ::= "agreements:" "(" [UserAgreementList] ")"
  UserAgreementList ::= ID ":" Literal {"," ID ":" Literal}

OriginExp ::= "origin:" "(" [IDList] ")"


Variable ::= "?" Name
ID ::= URI
IDV ::= Variable | URI
```

```
IDList ::= ID {"," ID}
IDListV ::=  IDV {"," IDV}
NameV ::= Variable | Name
Resource ::= URI
```

Examples

An example with all possible attributes queried is the following:

```
(tag:?x,ci:'kiwici:mswin',author:(?y),date:'2009-01-01',
agreements:('kiwiusr:bob':'-1'),origin:('kiwiusr:anton'),ks:?z)
```

This query looks for all annotations with some atomic tag that is assigned to the content item 'kiwici:mswin' (which might be a page about a prominent series of operating systems). The annotation can have any author and it must have been created on the first of January 2009. It is further demanded that the KiWi user Bob completely disagrees with that annotation. The annotation must have been derived with information based on Anton. The knowledge space that the annotation belongs to can be any of the existing ones.

While the example uses all of the attributes there are, one can see that it is not always a good idea to specify all attributes in such a query.

Here are some examples for very short AAQs:

```
tag:+'approved'
```

This AAQ queries any annotation in the system that contains as metadata the positive atomic tag 'approved'. As with all BAQs, all other attributes are not cared about. The query returns annotations for content items of positively approved projects or funds.

```
(tag:?x, ks:'kiwi:ProjectSpace')
```

This query, when applied, finds all annotations with an atomic tag in the knowledge space 'kiwi:ProjectSpace'. The result consists of all annotations attached to any content item that is a project page in the wiki.

```
(tag:-'approved', author:(?x) WHERE ?x='/kiwiusr:Bo/')
```

Similar to an above example, here annotations are queried having as metadata the negative atomic tag 'approved' and having as author someone whose value and thus name is further defined in the WHERE clause. In this case the value is matched with a regular expression stating that the name (or URI) should start with 'kiwiusr:Bo', strongly suggesting that f.e. Bob is a possible wanted author. Thus, this query could look for all annotations on content items of projects or funds that Bob (or somebody else whose name starts with 'Bo') does not approve of.

```
(tag: +'review', date:?d WHERE ?d<'2009-01-01')
```

Again, similar to an above example, annotations with a positive atomic tag 'review' as metadata are searched that have some creation date further specified in the WHERE clause. In this case, all such annotations who were created before the first day of 2009 are queried. This could be useful f.e. for finding any projects or even software code that were to be reviewed before 2009. Again, as with the examples above, not all attributes are really necessary in this case since it does not need to matter f.e. what knowledge space the annotation actually belongs to.

3.4.5 Atomic RDF query (ARQ)

In addition to annotations, users can also query RDF triples that are stored in the system. The attributes described above do not make sense for normal RDF triples but the RDF triples will also be stored in knowledge spaces, hence the query features the knowledge space attribute. If it is not specified, all knowledge spaces are queried for the triple.

```
ARQ ::= "(" RDFTripleExp ["," KnowledgeSpaceExp] ")"
```

3.4.6 Compound KWRL query

A compound (KWRL) query combines AAQs and ARQs with the following constructs:

Non-monotonic negation A negated query with non-monotonic negation succeeds if the positive query does not return any result. We use the keyword 'not' to designate non-monotonic negation in KWRL. Consider for example *not (stag:'knows'('Java'),ci:'kiwiusr:Bob')* which would succeed if there is no such annotation that says that Bob knows Java. Similarly, *not (stag:'knows'(?x),ci:'kiwiusr:Bob')* succeeds if there is no annotation specifying that Bob knows any programming language (or even anything).

conjunction of queries A conjunction of queries succeeds if every single query in the conjunction succeeds, i.e. returns one or more results. A conjunction is designated by the keyword 'and'. An example for this is *(stag:'knows'('C++'), ci:?x) and (stag:'knows'('Java'), ci:?x)* which returns in the variable ?x all content items (of users) that are annotated such that the respective user knows both C++ and Java.

disjunction of queries A disjunction of queries succeeds if at least one query in the disjunction succeeds. A disjunction is mainly designated by the keyword 'or'. An example for this is (stag:'knows'('C++'), ci:?x) or (stag:'knows'('Java'), ci:?x) which returns in the variable ?x all content items (of users) that are annotated such that the respective user knows C++, Java or both of them.

The syntax allows arbitrary nesting and combination of the above constructs.

```
CompoundQuery ::= AAQ | ARQ | "not" CompoundQuery |
                  Conjunction | Disjunction
Conjunction ::= "(" CompoundQuery { "and" CompoundQuery }")"
Disjunction ::= "(" CompoundQuery { "or" CompoundQuery }")"
```

Examples

```
not ('kiwiusr:bob' 'rdf:type' 'kiwi:manager',ks:'kiwi:UserSpace')
```

This example for non-monotonic negation is very simple. The query checks whether the specified RDF triple can NOT be found in the kiwi:UserSpace knowledge space. It would thus succeed if there is no such RDF information that declares that Bob is a manager.

```
((tag:+'review', ci:?x) and (?x 'rdf:type' 'kiwi:project'))
```

This query succeeds for those content items that are of type kiwi:project and that are annotated with a positive atomic tag review. Thus, this could be query for those projects (described by their content item) that anyone thinks must be reviewed. With this last example, it is feasible to not only think about (annotation) queries but also (annotation) construction. What if, for example, all projects that should be reviewed should also get a new annotation specifying exactly who is in charge for that review of the project?

For this consider something like this: For every match of ((tag:+'review', ci:?x) and (?x 'rdf:type' 'kiwi:project')) construct an annotation (stag:'review'('ResponsiblePerson'), ci:?x).

This leads us to the following statements: We are now talking about actual rules and a compound query is complete in the sense that it can represent a whole rule body.

Therefore, we now need to look at (annotation) construction to specify the complete rule syntax afterwards.

3.5 Rule syntax - rule head

3.5.1 Annotation construction

In the rule environment as seen above, it is important that variables in the (annotation) construction part have to be checked for correct use.

One criterion is range restriction: every variable universally quantified at rule level which is used in the construction part of a rule must also be used in the positive part of a query in that same rule. At the same time, however, we need (in this chapter implicit) existential variables for annotations especially when we look at the RDF triple translation (see later section) of rules.

Another point to consider is the context of the variables since for example variables bound to structured tags can not be used for a (negative) atomic tag construction (we disallow negative structured tags).

Note that this can not be represented in the construction syntax and has to be dealt with semantically.Similar to the query part of KWRL, we define basic annotation construction (BAC), atomic annotation construction (AAC) and the construction of RDF triples (ARC).

3.5.2 Basic annotation construction (BAC)

Syntactically, the BAC is the same as the BAQ, the specification of an annotation attribute with a keyword/value pair, with the following exception which is also true for AACs:

Only the common attributes can be specified by the user as the system attributes are automatically set by the system. The values of the system attributes of an annotation construction are set as follows:

- Author of a derived annotation is the author of the rule by which the annotation was derived (i.e. the authors list contains only one author, the author of the rule).
- Date of a derived annotation is the date when the derivation took place.

- User- and content-item-origins are computed by the system.
- Marker is 'user' if the annotation has been created by a user, 'derived' if it has been constructed by a rule head and 'system' if it was generated by the system.

3.5.3 Atomic annotation construction (AAC)

Again, similar to the BAQ/AAQ situation, an AAC is the specification of one or more attributes of an annotation. As described above, however, only the common attributes can be given by the user.

Thus, the AAC syntax is reduced to the following:

```
AAC ::= BAC | BACList
BACList ::= "(" BAC "," BAC {"," BAC} ")"
BAC ::= CommonAttrsExp
```

Examples

Keep in mind that in all AACs, the system attributes are set at the time of construction as described above.

```
(tag:+'approved', ci:?x)
```

Here, an annotation with a positive atomic tag approved is constructed for a content item specified in the variable ?x. In the corresponding rule body, ?x might be bound to the URI of a content item about a specific project.

```
(tag:+'approved', ci:'kiwici:TheMilleniumProject',
agreements:('kiwiusr:bob':'1'), ks:'kiwi:ProjectSpace')
```

In this example, the above notion is extended. This annotation construction features a positive atomic tag approved for the CI of a TheMilleniumProject. The annotation has a pre-set full agreement of the user Bob and the annotation is stored in the already mentioned ProjectSpace.

```
(triple: 'kiwi:TheMilleniumProject' 'kiwi:hasBudget' 'kiwi:big',
ci:'kiwici:TheMilleniumProject', agreements:('kiwiusr:bob':'1'),
ks:'kiwi:ProjectSpace')
```

RDF triples can also be used for metadata just as in the query part. Here, an annotation is constructed just like the above one with the exception that an RDF triple is given as metadata. This triple specifies that the project has a big budget.

Note that the last two annotations are easily fathomable with a variable in the CI attribute just like in the first example and could be easily used in a rule stating something like 'If a project (or s.e.) has a big budget, Bob approves of it.' which could be written like this:

```
(triple: ?x 'kiwi:hasBudget' 'kiwi:big', ci:?x, ks:'kiwi:ProjectSpace')
-> (tag:+'approved', ci:?x, agreements:('kiwiusr:bob':'1'),
ks:'kiwi:ProjectSpace')
```

For the rule syntax, see the later section.

3.5.4 An atomic RDF triple construction (ARC)

We have already seen that 'raw' RDF triples can be queried in the system. Good news is, RDF triples can be constructed, too. The RDF triple construction syntax is the same as the RDF triple query syntax. If no knowledge space is given, a default knowledge space for RDF triples is used.

```
ARC ::= ARQ
```

When having a rule with matching variables in its ARQs (in the rule body) and its ARCs (in the rule head), it should be checked whether the variables are set correctly, especially if an object is queried in a variable ?x which is then used in a predicate position in an ARC, it should either be invalid or it must be clear that a resource (i.e. URI) is bound instead of a literal value which can not be used in the predicate position.

3.5.5 Compound KWRL construction

A compound construction in KWRL includes at least one of an RDF triple construction or an atomic annotation construction in a conjunctive clause. During evalutation of the clause, each annotation/RDF triple given should be constructed one after the other. Non-monotonic negation and disjunctions, as possible in compound queries, are obviously disallowed since non-monotonic negation does not make any sense here and disjunctions would lead to non-determinism when used in rules and non-determinism is not something we want for our wiki environment. Rule heads consist of compund constructions and nothing else.

```
CompoundConstruction ::= ConstrElement { "and" ConstrElement }
ConstrElement ::= AAC | ARC
```

Examples

```
(tag:+'approved', ci:'kiwici:TheMilleniumProject',
agreements:('kiwiusr:bob':'1'), ks:'kiwi:ProjectSpace') and
(triple: 'kiwi:TheMilleniumProject' 'kiwi:hasBudget' 'kiwi:big',
ci:'kiwici:TheMilleniumProject', agreements:('kiwiusr:bob':'1'),
ks:'kiwi:ProjectSpace') and
('kiwi:TheMilleniumProject' 'rdf:type' 'kiwi:SoftwareProject',
ks:'kiwi:ProjectSpace')
```

This example should look familiar as two of the examples for AACs are taken and combined with the additional information, in form of a normal RDF triple, that TheMilleniumProject is also a software project.

```
(tag:+'review', ci:'kiwici:TheMilleniumProject') and
(triple: 'kiwi:TheMilleniumProject' 'rdf:type' 'kiwi:Project',
ci:'kiwici:TheMilleniumProject')
```

This compound construction states that the project is indeed a project and that it can/should/has be(en) reviewed.

3.6 KWRL Rules

A KWRL rule is composed by a rule body (LHS, left hand side) which consists of a compound query and a rule head (RHS, right hand side). Its basic structure is $LHS- > RHS$.

Two main types of rules are distinguished: construction rules and constraint rules.

A construction rule has a compound construction as its RHS with the keyword *construct* before it. It states that if the LHS holds, the RHS should be constructed that is all AACs and ARCs in the compound construction have to be created if they do not already exist.

A constraint rule has another compound query as its RHS with the keyword *exists* before it, stating that if the LHS holds, the RHS compound query must also hold.

If no keyword is given, a rule is assumed to be a construction rule per default.

An inconsistency constraint rule, however, has a special RHS, as it should define how an inconsistency (f.e. clashing opinions) can be derived. We define this inconsistency as a kind of marker that does not interfere with the inferencing process.

```
Rule ::= ConstructionRule | ConstraintRule
ConstructionRule ::= CompoundQuery "->" ["CONSTRUCT"] CompoundConstruction
ConstraintRule ::= CompoundQuery "->" "EXISTS" CompoundQuery
IConstraintRule ::= CompoundQuery "->" "inconsistency(" KVList ")"
```

where K(eyword)V(alue)List could contain f.e. a number of variable assignments and rule ids. Note that all following examples deal with construction rules rather than constraint rules.

Examples

```
(triple: ?x 'kiwi:hasBudget' 'kiwi:big', ci:?x, ks:'kiwi:ProjectSpace')
-> CONSTRUCT (tag:+'approved', ci:?x, agreements:('kiwiusr:bob':'1'),
ks:'kiwi:ProjectSpace')
```

This example we have seen before. As a very simple rule, its compound query and construction consists of only one atomic element. So here, IF an annotation can be found that has as metadata the specified RDF triple where ?x is bound to the subject position and ?x is also the value of the CI attribute and the annotation relies in the kiwi:ProjectSpace THEN construct an annotation with the positive atomic tag approved for the CI that was bound in ?x in the rule body and specify the agreements and knowledge space for it as written.

```
((tag:+'approved', ci:?x, author:('kiwiusr:manager')) and
(?x 'rdf:type' 'kiwi:Project')) -> (stag:'review'('bob'),ci:?x)
```

This example rule states that if an annotation with the positive atomic tag approved can be found for some CI ?x written by a or the manager and the ?x is of rdf:type kiwi:Project then another annotation should be constructed for this CI ?x with the given structured tag as its metadata. This rule could be expressed in words like this: For every project approved by the manager, Bob should review that project.

3.7 KWRL Programs

A KWRL program is a set of rules that can be evaluated by a reasoner for KWRL.[3]

We have seen how to query annotations (and RDF) and how to construct annotations (and RDF) and how to combine those two to form rules. Rules should be processed by a rule engine or reasoner. Using a certain inference technique, the rules should be evaluated, querying the LHS part of each rule and constructiong the RHS part of it if the query succeeds. There is one thing to consider, though.

All annotations are stored in knowledge spaces[4], given by their respective attribute. If rules like we have defined above are evaluated, there should be a way to state default source and destination knowledge spaces, i.e. knowledge spaces where the annotations from the LHS of the rules are searched and knowledge spaces where the constructed annotations from the RHS of the rules are then saved. These values can then be overwritten locally in the rules themselves with the knowledge space attribute.

Thus, syntactically, a KWRL program consists of a list of rules combined with two predicates that specify default source and destination knowledge spaces.

```
Program ::= KSImportList <WS> OutputKS <WS> RuleList
KSImportList ::= "import" ID {<WS> "import" ID}
OutputKS ::= "output" ID
RuleList ::= Rule {<WS> Rule}
```

In this syntax, $< WS >$ represents a whitespace character.

The specification of knowledge space using the 'output' keyword creates a default which is used when no knowledge space is specified in the rule construction. If there is no knowledge space specified in a part of a rule body then all imported knowledge spaces are queried.[5]

Example

```
import 'kiwi:ProjectSpace'
output 'kiwi:ProjectSpace'

(tag:+'review', ci:?x, author:('kiwiusr:manager')) and
(tag:-'bug', ci:?x, author:('kiwiusr:programmer'))
-> (tag:+'approved', ci:?x) and (?x 'rdf:type' 'kiwi:urgent')
```

In this example, both the default source (although more than one can be specified) and destination knowledge spaces are the project knowledge space which we have seen many times before.

The one rule in this program does not have any knowledge spaces specified anywhere, thus, the default project knowledge space is used for every annotation.

[3] See outlook section in the conclusion

[4] Consider for example annotations belonging to content items of projects, users etc. The annotations should reside in the same knowledge spaces

[5] Note that now the syntax allows to specify only one knowledge space in simple queries. This could be extended or worked around by repeating the rules.

This means that the query looks for two annotations for a content item ?x located in the project knowledge space: one with a positive atomic tag review that was written by the management, the other with a negative atomic tag bug which was written by a programmer.

If, during evaluation of the rule of the program, the query succeeds, both an annotation with a positive atomic tag approved for that same content item and an RDF triple specifying that the (project from the) CI is deemed urgent are created in the project knowledge space.

3.8 Syntactic sugar

We have seen that for construction rules, both rule bodies and rule heads consist of compound queries or constructions which in turn (aside from plain RDF triple queries/constructions) consist of AAQs or AACs that feature at least one keyword value pair for a (syntactically allowed) keyword value pair of an annotation attribute. Therefore, it is allowed that attribute key value pairs must not be specified by the user.

Recall the annotation attributes and their type:

Attribute name	Attribute type
metadata	common attribute
content item	common attribute
authors	system attribute
date of creation	system attribute
marker	system attribute
agreements	system attribute
tracking information	system attribute
knowledge space	common attribute

If there is one or more AAQs in a rule body that do not contain one or more of these attributes, the rule body must be evaluated as specified in the table below.

Attribute name	Effect if not specified in rule body
metadata	insert shared system-defined rule-wide variable in its place (?MD) for all metadata
content item	insert shared system-defined rule-wide variable in its place (?CI)
authors	insert system-defined unique unshared existential variable
date of creation	insert system-defined unique unshared existential variable
marker	- (cannot be queried)
agreements	insert system-defined unique unshared existential variable
tracking information	insert system-defined unique unshared existential variable
knowledge space	in program: use imported knowledge spaces (see above), otherwise shared variable (?KS)

In a rule head, the effects of missing attributes is slightly different:

ATTRIBUTE NAME	EFFECT IF NOT SPECIFIED IN RULE HEAD
metadata	if not specified somewhere in rule body: use shared variable (see rule body), otherwise disallow
content item	if not specified somewhere in rule body: use shared variable (see rule body), otherwise disallow
authors	- (system-defined, see AAC)
date of creation	- (system-defined, see AAC)
marker	- (system-defined)
agreements	- (system-defined, see AAC)
tracking information	- (system-defined, see AAC)
knowledge space	in program: use output knowledge space (see above), otherwise see other common attributes

As can be seen, the effects of a missing attribute is mainly dependent on its type:

Any one of the system attributes, when not specified in any AAQ of the rule body, is not cared about, i.e. since the user does not want to specify anything for it, a unique (generated) existential variable is used that will match with any value possible (this includes the empty value/list). Consider for example annotations from any author or any point of time. In rule heads, system attributes can not be specified since the system itself sets their values.

The interesting part is how to deal with non-specified common attributes.

If a common attribute is not specified, we extend the syntax in the following way, considering an iterative inference process which evaluates one rule after the other:

The value of a non-specified common attribute in any AAQ defaults to its specific shared rule-wide system-defined variable (?MD for metadata instead of the whole keyword/value pair, recognizable as the only attribute in an AAQ without a keyword; ?CI for content item; ?KS for knowledge space).

This is directly related to the rule head: For any AAC that has a common attribute not specified, its shared variable is automatically used if it has been already used at least once in the rule body. The possible combinations of (non-)specifications of a common attribute (CA) and the results are best shown in another table (AAQ means 'at least once not specified in an AAQ', AAC 'at least once not specified in an AAC'):

AAQ?	AAC?	RESULT
no	no	all values of CA are specified in the rule thus nothing to do
no	yes	disallowed (what values should be used for CA in rule head?)
yes	no	shared variable in rule body
yes	yes	all values of non-specified CA set to shared variable thus rule head (construction) takes values from rule body

Since default source and destination knowledge spaces must be set in a KWRL program, the knowledge space attribute defaults to those values and the shared variable is ignored when used in a KWRL program.

Note that the shared variables for the common attributes can be used in rule construction by the rule programmer. See the respective examples from the following list:

Example rule list (numbered for convenience)

```
1) tag:+'bug' -> tag:+'review'
```

Here, as in other examples, all system attributes are not specified, i.e. not cared about. This means that any value for them will do in the rule body - in this case in the one AAQ - and the values will be constructed as described above in the rule head - in this case for the one AAC. For brevity reasons, this concept for system attributes, if applicable, will not be explained for all following examples.

Note also that the handling of the knowledge spaces should be clear - either the values are given by the program or the values are treated just like the values for the other two common attributes.

The metadata attribute is specified for both sides of the rule, the specification of the other two common attributes is missing. The results (see above) of this capture the intuitive meaning of this rule perfectly: For every annotation found that contains the tag 'bug', we want an annotation with a tag 'review' - both the content item and the knowledge space should be the same (why declare the new annotation for any other content item or in any other knowledge space?).

The full version of this rule would read something like this:

```
(tag:+'bug', ci:?CI, author:?AU423, date:?DA735,
agreement:?AG981, origin:?OR119, ks:?KS) ->
(tag:+'review', ci:?CI, author:[constructed],
date:[constructed], agreement:[empty],
origin:[constructed], ks:?KS)
```

where constructed means constructed by the system as described in the AAC section.

Therefore if f.e. an annotation

```
(tag:+'bug', ci:'kiwici:TheMilleniumProject', ...,
ks:'kiwi:ProjectSpace')
```

is found,

```
(tag:+'review', ci:'kiwici:TheMilleniumProject', ...,
ks:'kiwi:ProjectSpace')
```

is the correspondent constructed annotation.

```
2) tag:+'bug' -> ci:'kiwici:TheMilleniumProject'
```

This rule is disallowed since it is an instance of the second case of the CA table above: In the rule head (here: the AAC), a CA (in this case: metadata) is not specified that is specified completely in the rule body (here: the AAQ).

The lethal question would be: What should be used as metadata for the AAC? Since there is no shared variable from the rule body that could be used, only one specific possibility remains: the atomic tag specified in the AAQ. The resulting rule

```
(tag:+'bug') -> (tag:+'bug', ci:'kiwici:TheMilleniumProject')
```

would not make much more sense and even worse, when evaluated it could throw the program into an infinite loop.

Note that the full version of this rule, if it were legal, would read something like this:

```
(tag:+'bug', ci:?CI, author:?AU238, ..., ks:?KS) ->
(?MD, ci:'kiwici:TheMilleniumProject', ..., ks:?KS)
```

Notice especially the ?MD variable for all metadata possiblities.

```
3) (tag:+'bug' or tag:+'error') ->
(stag:+'check'(?CI), ci:'kiwici:ProgrammerPage')
```

Contrary to the last 'rule', this rule is completely legal. Note two interesting points here:

1. The usage of the shared variable ?CI in the argument position of the structured tag.

2. If it were not for that shared variable, this rule, depending on the used inferencing method, could produce the AAC (much) more than once when evaluated. See the semantics section for this problem.

The rule features an annotation with a specified content item in the rule head and none specified in the rule body. According to the CA table, it is the third case and thus, the content items in the rule body are represented by the (already used) shared variable. The slightly extended rule reads like this:

```
((tag:+'bug', ci:?CI) or (tag:+'error', ci:?CI)) ->
(stag:+'check'(?CI), ci:'kiwici:ProgrammerPage')
```

This means that for every content item, which could also be a piece of code, that is atomically tagged with either 'bug' or 'error' an annotation with a structured tag telling the programmers to check that content item is assigned to their page.

```
4) (tag:+'foo',ci:?x) and (tag:+'bar',ci:?y) -> (?x 'kiwi:relatedTo' ?y)
```

This rule is obviously allowed since the construction part of the rule merely features an RDF triple construction with variables that are bound in the rule body during querying.

It should come as no surprise that the query part of the rule states that for it to succeed, one annotation must exist that has as metadata a positive atomic tag 'foo' on any content item ?x and another with a positive atomic tag 'bar' on another content item ?y. The construction part specifies that if such a combination exists, ?x is related to ?y.

The long variation of this rule would read something like this:

```
(tag:+'foo', ci:?x, author:?AU582, ..., ks:?KS)
and (tag:+'bar', ci:?y, author:?AU372, ..., ks:?KS)
-> (?x 'kiwi:relatedTo' ?y)
```

Note that the query in this form also succeeds if there is one content item annotated with both atomic tags, in which case ?x would be bound to the same (URI) as ?y. To ensure that the content items are not equal, an additional WHERE clause could be used, f.e.

```
(tag:+'foo',ci:?x) and (tag:+'bar',ci:?y) WHERE ?x <> ?y
```

3.9 Translation to RDF triples

3.9.1 KWRL programs

In the KiWi system RDF data is used and so the annotations must be in RDF, too. Recall a few definitions:

KWRL rule A rule as described in the syntax above. Additionally, every rule is assigned a unique rule id (i.e. URI) with which it can be further described in the system.

KWRL program A program, as we have seen, consists of one or more rules in the annotation tuple format. It is also possible to specify default input and output knowledge spaces as described in the respective section.

Knowledge space Each knowledge space can hold RDF triples in the KiWi system. There can be numerous knowledge spaces for different subjects. To express that RDF triples (f.e. translated annoation tuples) belong to knowledge spaces, named graphs are used. [6]

Rule processor As part of the KIWI system, it reads and parses the rules as instructed.

Consider some KWRL program program1 with KWRL rules ruleruri1, ruleuri2 and ruleuri3. Then the RDF triple representation in the system looks like this (note that all syntax and system description is heavily simplified):

```
ruleuri1 rdf:type kiwi:rule
ruleuri2 rdf:type kiwi:rule
ruleuri3 rdf:type kiwi:rule
program1 rdf:type kiwi:program
program1 kiwi:hasRule ruleuri1
program1 kiwi:hasRule ruleuri2
program1 kiwi:hasRule ruleuri3
```

The rules themselves are represented like this:

```
ruleuri1 kiwi:rule 'compound query 1 -> compound construction 1'
ruleuri2 kiwi:rule 'compound query 2 -> compound construction 2'
ruleuri3 kiwi:rule 'compound query 3 -> compound construction 3'
ruleuri4 kiwi:rule 'compound query 4 -> EXISTS compound query 5'
```

[6] see section about named graphs and knowledge spaces

with the literals being the actual string representations of the rules with the whole syntax described above, f.e. $(tag : -'bug') - > (tag : +'approve')$.

When actually needed, the rules are processed by the rule processor which translates everything to RDF triples internally to match the query and bind the variables and evaluates them according to the semantics described below.

At the end of the process, new annotations might be constructed in RDF format.

3.9.2 Annotation tuples in RDF format

Considering what an annotation actually is, an assignment of metadata to a content item with certain other attributes, the first RDF triple of a translation of an annotation tuple in the form of (..., ci:'ex:foo',...) is ('ex:foo' 'kiwi:hasAnnotation' annotationID) with possibly another triple (annotationID 'rdf:type' 'kiwi:annotation) where annotationID could be the URI of the translated annotation.

Following this pattern, the attributes of the annotation might just be written (simplified) as f.e.

```
(annotationID 'kiwi:hasMetadata' URI)
(annotationID 'kiwi:hasAuthor' 'kiwiusr:Bob')
(annotationID 'kiwi:hasDate' '2009-12-12')
(annotationID 'kiwi:hasMarker' 'derived')
(annotationID 'kiwi:hasAgreement' '(kiwiusr:Alice:1)')
(annotationID 'kiwi:hasOrigin' '(kiwiusr:Chuck, kiwici:TheProject)')
(annotationID 'kiwi:hasKnowledgeSpace' 'kiwi:ProjectSpace')
```

The value of the list attributes can be specified in two ways:

1. extra RDF list vocabulary: (aID kiwi:hasAuthor listURI1), (listURI1 rdf:type kiwi:list), (listURI1 kiwi:member kiwiusr:bob)

2. multiple RDF triples as such: (aID kiwi:hasAuthor kiwiusr:bob), (aID kiwi:hasAuthor kiwiusr:alice)

3.9.3 Translation of annotation attributes with values

Following are more detailed examples and sketches of how the different annotation attributes are specified in RDF. Note that the 'single value' attributes (date, marker and knowledgespace) can be translated as easily as seen above.

hasMetadata Assuming that each of the 3 types of metadata are assigned URIs (see above), we get the following possibilites:

1. Atomic tag:
   (x hasMetadata atomictagURI1) (atomictagURI1 hasPolarity pos) (atomictagURI1 hasLabel 'fish')

2. Structured tag:
   (x hasMetadata stagURI1) (stagURI1 hasLabel 'fish') (stagURI1 hasArgument 'foo') (stagURI1 hasArgument 'bar)

3. RDF triple:
   (x hasMetadata tripleURI1) (tripleURI1 rdf:subject s)
   (tripleURI1 rdf:predicate p) (tripleURI1 rdf:object o)

hasAuthor (list attribute)
1. (x hasAuthor listURIx)
   (listURIx rdf:type kiwi:list)
   (listURIx kiwi:member kiwiusr:bob) (listURIx kiwi:member kiwiusr:alice)
2. (x hasAuthor kiwiusr:bob) (x hasAuthor kiwiusr:alice) ...

hasDate (x hasDate '2009-12-12')

hasMarker (x hasMarker 'derived')

hasAgreement (list attribute) Either a similar format to that of hasAuthor is used or extra vocabulary is defined like this:
(x hasAgreement agreement1) (x hasAgreement agreement2)
(agreement1 kiwi:hasUser kiwiusr:bob) (agreement1 kiwi:hasValue '1')
(agreement2 kiwi:hasUser kiwiusr:alice) (agreement1 kiwi:hasValue '0')

hasOrigin (list attribute)
1. (x hasOrigin listURIx) (listURIx rdf:type kiwi:list) (listURIx kiwi:member kiwiusr:bob)
   (listURIx kiwi:member kiwici:TheProject)
2. (x hasOrigin kiwiusr:bob) (x hasOrigin kiwici:TheProject) ...

hasKnowledgeSpace (x hasKnowledgeSpace 'kiwi:ProjectSpace')

3.9.4 Query and construction of annotation tuples

The query or construction of single attributes is presented similar to above, it should be clear that, for several specified attributes, all triples are created, for the creation of the system attributes see the construction syntax. Consider a content item CI and an annotation with ID aID so that (CI hasAnnotation aID) and the following specified attributes in an annotation query or construction:

hasMetadata *(?x, ...)* translates to (aID hasMetadata ?x).
(tag:?x, ...) translates to (aID hasMetadata ?tagURI)
(?tagURI rdf:type kiwi:atomictag) (?tagURI hasLabel ?x).
(stag:foo(?x), ...) translates to (aID hasMetadata ?stagURI)
(?stagURI rdf:type kiwi:stag)
(?stagURI hasLabel 'foo') (?stagURI hasArgument ?x).
(triple: ?s ?p ?o, ...) translates to (aID hasMetadata ?tripleURI)
(?tripleURI rdf:subject ?s) (?tripleURI rdf:predicate ?p)
(?tripleURI rdf:object ?o).

hasAuthor *(..., author:?x, ...)* translates to (aID hasAuthor ?x).
(..., author:(?x, ?y), ...) translates to
(aID hasAuthor ?x) (aID hasAuthor ?y).

hasDate *(..., date:?x, ...)* translates to (aID hasDate ?x).

hasMarker Not specifiable in query or construction, generated by the system (see above).

hasAgreement *(..., agreement:(kiwiusr:alice:?x), ...)* translates to
(aID hasAgreement agreement1)
(agreement1 hasUser kiwiusr:alice) (agreement1 hasValue ?x).

hasOrigin *(..., origin:?x, ...)* translates to (aID hasOrigin ?x).
(..., origin:(?x, ?y), ...) translates to
(aID hasOrigin ?x) (aID hasOrigin ?y).

hasKnowledgeSpace *(..., ks:?x)* translates to (aID hasKnowledgeSpace ?x).

3.9.5 Non-monotonic negation

Again, consider any content item CI with annotation IDs aIDx so that (CI hasAnnotation aIDx).

Regarding non-monotonic negation for the RDF triples of an annotation, one or more RDF triples of the above type are involved in a query with weak negation (shared variables for common attributes are left out, think (aID hasKnowledgeSpace ?KS)):

not (tag:+bug) is interpreted like this: there is no aIDx for CI so that the following tuples exist in the system:
(aIDx hasMetadata tagURI1), (tagURI1 rdf:type kiwi:atomictag),
(tagURI1 hasLabel 'bug'), (tagURI1 hasPolarity 'pos')
i.e. there is no content item with the positive tag 'bug'.

not (tag:-bug, ci:'kiwici:theproject') is interpreted like this: there is no annotationID aIDx so that the following tuples exist in the system:
(aIDx hasMetadata tagURI2), (tagURI2 rdf:type kiwi:atomictag),
(tagURI2 hasLabel 'bug'), (tagURI2 hasPolarity 'neg'),
(aIDx hasCI 'kiwici:theproject) i.e. the CI kiwici:theproject does not have an annotation with the negative atomic tag 'bug'.

not (tag:?x) is interpreted like this: there is no aIDx so that:
(aIDx hasMetadata tagURI3), (tagURI3 rdf:type kiwi:atomictag).
if ?x has been bound before: (tagURI3 hasLabel valueof(?x)) (see above), otherwise content items that do not have tags as annotations.

not (?x) is interpreted like this: there is no metadata and thus no annotation for the content item

3.10 Semantics

The semantics for KWRL are kept brief and informal. The semantics and the implementation will be issues for further research.

3.10.1 Rules

One of the desired core features of the language is the usage of declarative rules, i.e. rules with a declarative meaning, instead of production or business rules. This means that the order of evaluation of the rules in a program should make no difference as to the outcome of said program (see program semantics).

Therefore, instead of f.e. Java code, both the LHS and the RHS of each rule can be seen as mere statements (declarations).

LHS/rule body Every LHS of a KWRL rule is a compound query of one or more AAQs or ARQs combined with WHERE clauses and operators *and* and *or* whose behavior follows FOL.

Thus, with X and Y being AAQs, *(X and Y)* succeeds if both X and Y succeed. Similarly, *(X or Y)* succeeds if X succeeds, Y succeeds or both.

The not operator for non-monotonic negation is not an operator of first order logic - though it can be modelled - and it is different from classical negation.

As already described above, *(not X)* succeeds if X can not be proven, i.e. an annotation with the specified attributes can not be found in the system. This is a huge difference to classical negation which would have (not X) succeed if X is proven false as opposed to X can not be proven true.

RHS/rule head Every RHS of a KWRL construction rule is a conjunction of AACs and ARCs. The used operator for conjunction is *and* just like in the LHS part of a rule but it is, by the very nature of a construction as opposed to a query, not quite the same. It IS the same in that for a conjunction of a construction to 'succeed', all atomic constructions must 'succeed'.

A single construction can not really succeed however, since the specified annotation/RDF triple usually does not exist yet and has to be constructed - built by the system - first. If a specified annotation/RDF triple already exists, nothing is done. Thus, given some rule in the $LHS- > RHS$ format, if the LHS succeeds, the RHS must 'succeed', too, such that every annotation or RDF triple specified by the atomic constructions must either already exist or be created by the system.

As hinted at in the syntax section, a disjunction (or) of atomic constructions in the RHS of a rule would lead to non-determinism: which atomic constructions should be evaluated, the first one? All of them? As we will see, we assume fixpoint semantics for the evaluation of KWRL programs based on the fact that we do not allow disjunctions in the rule head.

In the same vein, non-monotonic negation is just as problematic. How could a construction be negated in the non-monotonic sense of negation described above? Consider some rule $LHS- > NOT(tag :?x, ...)$. If LHS succeeds here, what happens with the specified AAC? If it already existed, would it have to be removed so that the negation would 'succeed'? So, in english, non-monotonic negation in the RHS would translate to something like 'If the LHS succeeds, make sure that the following annotation(s)/RDF triple(s) do NOT exist in the system'

This issue is unproblematic for constraint rules, as their RHSs are compound queries themselves, featuring non-monotonic negation and all operators as described above. The semantics are the same as for normal queries, except for constraint rules, nothing is constructed but rather further checked for existence.

3.10.2 KWRL Programs

When executing KWRL programs, we assume fixpoint semantics, that is, all rules are applied to the data over and over depending on the inference method until no new data can be derived.[7] This is possible because we have declarative rules as described above that allow us to do so.

[7]See the (semi-)naive forward chaining inference methods in the inference chapter for examples.

Kapitel 4

Conclusion

4.1 Preliminaries

In the first chapter we discussed the basics of RDF/S inferencing and the theory behind it. We covered such topics as the RDF model theory, negation and inferencing methods.

We had a look at existing RDF related languages and reasoners in the second chapter, describing them and evaluating their use for the project. We saw that even though some of the reasoners provide many features that we would want to have, none of them can be used completely especially considering our background of the social wiki environment where several non-standard features would have to be implemented.

This is why, in the third chapter, we defined KWRL - a rule language for annotations based on RDF data.

4.2 KWRL

4.2.1 Definition

At first we defined the notion of *annotation*s - assignments of metadata to content items with extra parameters (*attributes*) such as author of the annotation and creation date.

A high-level representation of an annotation is an *annotation tuple* which is a tuple of keyword/value pairs specifying the values of the attributes of a single annotation.

Combined with variables, we built basic queries for annotations (AAQ) that in turn, combined with logic operators and non-monotonic negation and RDF triple queries, make up compound queries which are used as rule bodies in KWRL. For the rule heads we similarly defined compound constructions (again with constructions of RDF triples) and disallowed problematic operators such as non-monotonic negation and disjunctions.

We said that a *KWRL program* is a set of thusly created rules with the specification of default input and output knowledge spaces.

Everything mentioned until now is part of the high-level language but ultimately the annotations are stored as RDF data. That is why we defined a translation of annotations to RDF data and explained how the KiWi system evaluates KWRL programs by translating the high-level rules to the RDF translation to then, if the rule is applicable, construct new annotations in RDF.

The syntactic sugar part showed what attributes do not have to be specified in the high-level language, what not completely specified annotations translate to in the high-level language and in RDF.

Finally, we sketched the semantics for KWRL rules and programs.

4.2.2 Outlook: Implementation

With a defined rule language and a suggestion of how it can be applied to use cases, what remains is an actual implementation of a reasoner for KWRL.

As we have seen in the KWRL semantics sections, KWRL rules are declarative and thus we can assume fixpoint semantics for KWRL programs, i.e. a reasoner for KWRL would apply the rules over and over in turns until no new data has been constructed in the last turn.

For this task, the forward chaining algorithms described in the introduction chapter of the thesis can be used: the semi-naive forward chaining algorithm could be optimized for RDF and annotations and the Rete algorithm could make use of the almost definite appearance of multiple common parts of rule bodies (f.e. (CI hasAnnotation aID)).

Literaturverzeichnis

[1] Clark, J., DeRose, S., et al.: XML path language (XPath) version 1.0. W3C recommendation **16** (1999) 1999

[2] Schaffert, S., Bry, F., Dolog, P., Eder, J., Grunwald, S., Herwig, J., Holy, J., Nielsen, P., Smrz, P.: The kiwi vision: Collaborative knowledge management, powered by the semantic web. Technical report, Technical report, August 2008. FP7 ICT STREP KiWi project Deliverable D8

[3] Klyne, G., Carroll, J., McBride, B.: Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation **10** (2004)

[4] Manola, F., Miller, E., McBride, B.: RDF primer. W3C recommendation **10** (2004)

[5] McGuinness, D., Van Harmelen, F., et al.: OWL web ontology language overview. W3C recommendation **10** (2004) 2004–03

[6] Brickley, D., Guha, R., McBride, B.: RDF vocabulary description language 1.0: RDF schema. W3C recommendation **10** (2004)

[7] Hayes, P.: Rdf model theory. Technical report, Technical report, W3C Working Draft (2001)

[8] Bry, F., Eisinger, N., Eiter, T., Furche, T., Gottlob, G., Ley, C., Linse, B., Pichler, R., Wei, F.: Foundations of rule-based query answering. Lecture Notes in Computer Science **4636** (2007) 1

[9] Schmedding, F., Sawas, N., Lausen, G.: Adapting the Rete-Algorithm to Evaluate F-Logic Rules. Lecture Notes in Computer Science **4824** (2007) 166

[10] Doorenbos, R.: Production matching for large learning systems. PhD thesis, Citeseer (1995)

[11] Ramakrishnan, R.: Magic templates: A spellbinding approach to logic programs. JLP **11**(3&4) (1991) 189–216

[12] Kowalski, R., Kuehner, D.: Linear resolution with selection function. Artificial Intelligence **2**(3/4) (1971) 227–260

[13] Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Proceedings of the Third International Conference on Logic Programming. Volume 225., Springer (1986) 84–98

[14] Francois, B.: Query evaluation in recursive databases: bottom-up and top-down reconciled. Data & Knowledge Engineering **5** (1990) 289–312

[15] Broekstra, J., Kampman, A.: Inferencing and Truth Maintenance in RDF Schema: exploring a naive practical approach. In: Workshop on Practical and Scalable Semantic Systems (PSSS), Citeseer (2003)

[16] Furche, T., Linse, B., Bry, F., Plexousakis, D., Gottlob, G.: Rdf querying: Language constructs and evaluation methods compared. Reasoning Web **4126** (2006) 1–52

[17] Ogbuji, C.: Versa: Path-Based RDF Query Language. Retrieved January **8** (2005) 2007

[18] Souzis, A.: RxPath: a mapping of RDF to the XPath Data Model. In: Extreme Markup Languages. (2006)

[19] Seaborne, A.: Rdql-a query language for rdf. W3C Member submission **9** (2004) 29–1

[20] Broekstra, J., Kampman, A.: SeRQL: An RDF query and transformation language. In: Submitted to the International Semantic Web Conference, ISWC. Volume 2004. (2004)

[21] PrudHommeaux, E., Seaborne, A., et al.: SPARQL query language for RDF. W3C working draft **4** (2006)

[22] Berners-Lee, T., Connolly, D.: Primer: Getting into RDF & Semantic Web using N3. World Wide Web Consortium (2000)

[23] Sintek, M., Decker, S.: Triple-a query, inference, and transformation language for the semantic web. Lecture Notes in Computer Science (2002) 364–378

[24] Corby, O., Faron-Zucker, C.: Corese: A corporate semantic web engine. In: Workshop on Real World RDF and Semantic Web Applications, WWW Conference, Hawaii. (2002)

[25] Hawke, S.: Rule Interchange Format Working Group Charter. W3C Semantic Web Activity (2005)

[26] Boley, H., Tabet, S., Wagner, G.: Design rationale of RuleML: A markup language for semantic web rules. In: International Semantic Web Working Symposium (SWWS), Citeseer (2001) 381–402

[27] Bry, F., Furche, T., Ley, C., Linse, B., Marnette, B.: RDFLog: Its like Datalog for RDF. In: Proceedings of 22nd Workshop on (Constraint) Logic Programming, Dresden (30th September–1st October 2008), Citeseer (2008)

[28] ???: Graphpath language. `http://www.langdale.com.au/GraphPath/`

[29] Naudts, G., Gevaert, A.: An inference engine for RDF

[30] Batory, D.: The LEAPS algorithms. Department of Computer Sciences, University of Texas at Austin, Technical Report 94–28

[31] Miranker, D.: TREAT: a new and efficient match algorithm for AI production systems. PhD thesis, Columbia Univ., Columbia, NY (1987)

[32] Berners-Lee, T.: cwm - a general purpose data processor for the semantic web. `http://www.w3.org/2000/10/swap/doc/cwm.html`

[33] Kiryakov, A., Ognyanov, D., Manov, D.: Owlim-a pragmatic semantic repository for owl. Lecture Notes in Computer Science **3807** (2005) 182

[34] Jang, M., Sohn, J.: Bossam: An extended rule engine for owl inferencing. Lecture Notes in Computer Science **3323** (2004) 128–138

[35] Reynolds, D.: Jena 2 inference support. Online manual (2004)

[36] Wu, Z., Eadon, G., Das, S., Chong, E., Kolovski, V., Annamalai, M., Srinivasan, J.: Implementing an inference engine for RDFS/OWL constructs and user-defined rules in oracle. In: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, IEEE Computer Society Washington, DC, USA (2008) 1239–1248

[37] Boley, H., Mei, J., Sintek, M., Gerd Wagner, G.: RDF/RuleML Interoperability. In: W3C Workshop on Rule Languages for Interoperability Position Paper. (2005) 27–28

[38] Schaffert, S., Eder, J., Grünwald, S., Kurz, T., Radulescu, M., Sint, R., Stroka, S.: KiWi–a platform for semantic social software. In: Proceedings of the 4th Workshop on Semantic Wikis, European Semantic Web Conference. (2009)

[39] Halpin, H.: Provenance: The Missing Component of the Semantic Web. In: Proceedings of the First Workshop on Trust and Privacy on the Social and Semantic Web (SPOT2009). (2009)

[40] Harth, A., Polleres, A., Decker, S.: Towards a social provenance model for the web

[41] Hartig, O.: Provenance Information in the Web of Data. In: Proc. of the Linked Data on the Web Workshop at WWW. (2009)

[42] Carroll, J.J., Bizer, C., Hayes, P., Stickler, P.: Named graphs, provenance and trust. In: WWW '05: Proceedings of the 14th international conference on World Wide Web, New York, NY, USA, ACM (2005) 613–622

[43] Carroll, J., Bizer, C., Hayes, P., Stickler, P.: Named graphs. Web Semantics: Science, Services and Agents on the World Wide Web **3**(4) (December 2005) 247–267

[44] Schenk, S., Staab, S.: Networked rdf graphs. Technical report (2007)

[45] Pediaditis, P., Flouris, G., Fundulaki, I., Christophides, V.: On explicit provenance management in rdf/s graphs. In: TAPP'09: First workshop on on Theory and practice of provenance, Berkeley, CA, USA, USENIX Association (2009) 1–10

[46] Ding, L., Finin, T., Peng, Y., da Silva, P.P., Mcguinness, D.L.: Tracking rdf graph provenance using rdf molecules. (2005)

[47] Glavic, B., Dittrich, K.: Data provenance: A categorization of existing approaches. In: BTW07 - 12. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web. (2007)

[48] Watkins, E., Nicole, D.: Named graphs as a mechanism for reasoning about provenance. (2006) 943–948

[49] Prud'hommeaux, E., Seaborne, A.: Sparql query language for rdf

[50] Sintek, M., van Elst, L., Scerri, S., Handschuh, S.: Distributed knowledge representation on the social semantic desktop: Named graphs, views and roles in nrl. (2007) 594–608

[51] Jeita: Exif specifications. `http://www.exif.org/specifications.html`

[52] Bry, F., Weiand, K.: Kwql, querying for social semantic software. In: Proceedings of Reasoning Web, Fifth International Summer School 2009, Bressaone, Italy (30th August–4th September 2009). (2009) (Poster).

[53] Bry, F., Weiand, K.: Flavours of kwql, a keyword query language for a semantic wiki. In: Proceedings of OFSEM 2010: 36th International Conference on Current Trends in Theory and Practice of Computer Science, pindlerv Mln, Czech Republic (23rd–29th January 2010). (2010)