Tequila - a query language for the Semantic Web

Jakub Galgonek

Department of software engineering, Faculty of Mathematics and Physics,
Charles University in Prague,
Malostranské nám. 25, 118 00, Prague 1, Czech Republic
galgonek@ksi.mff.cuni.cz

Abstract. In order to realize the idea of the Semantic Web, many tools and technologies need to be developed, including a query language. One of the most important technologies that creates a base for the Semantic Web is the RDF (Resource Description Framework). There are many query languages for the RDF, but almost all of them are very weak and they can not be used for a general purpose. In this paper, we introduce the Tequila query language, developed for the recently introduced Trisolda infrastructure for the Semantic Web. In particular, we describe the syntax and semantics of the Tequila language and we show usage of this language and its power on several examples. The examples show how to convert a RDF Collection into a RDF Bag or how to evaluate an aggregation function (e.g., maximum of values) without the need of a built-in support. Unlike almost all other query languages, Tequila is powerful enough to express both of the tasks.

Keywords: Semantic Web, RDF, query language, Trisolda

1 Introduction

The Semantic Web is an idea of storing data together with their meaning. In order to its realisation, many tools and technologies need to be developed. It includes a language for storing data, vocabularies for describing an ontology and others. In order to have the ability to query on specific data, it also includes a query language as one of the important parts.

The Resource Description Framework (RDF) [3] is widely used as a language for storing data and it creates a base of the Semantic Web. The simple RDFS vocabulary or the more complicate OWL vocabulary are used for describing an ontology. There are also many query languages for RDF [6][8], such as the SPARQL language [4] or the SeRQL [2] language. However, almost all of them are very weak and they are not well applicable for a general purpose.

1.1 Troubles of the current query languages

Troubles of the current query languages for RDF can be divided into practical troubles and philosophical troubles. Most of them are related to each other.

K. Richta, J. Pokorný, V. Snášel (Eds.): Dateso 2009, pp. 105–118, ISBN 978-80-01-04323-3.

The practical troubles include the poor ability to resolve a general task. The languages are either too specialised or they have just weak constructs. For example, many of the query languages are not able to select a RDF Collection from a data source, although the structure of a RDF Collection is directly defined in the RDF specification. Generally, the current query languages have troubles with a selection of a recursively defined structure.

Yes, some languages introduce powerful constructs such as regular paths (the ARQ language [1]) or deductive rules (the TRIPLE languages [9]). Although these constructs give very good power for selecting data, they are still too weak for creating data. The ability to create data is useful not only for storing new data to a data source but also for composing queries, where a result of one query is used as an input source for other query.

The weak ability to create data also includes the bad support for blank nodes. For example, the SPARQL language cannot refer to the same blank node from different CONSTRUCT patterns, so it is impossible to connect more sub-solutions to one blank node. On the other hand, the SeRQL language introduces global identification of blank nodes, although it roughly breaks the RDF blank node semantics.

The philosophical troubles include that the current languages are not closed. So a solution of a query cannot be used as an input for other query. In addition, they do not use RDF to represent solutions of their queries. It is not a good property of the current RDF query languages.

1.2 A new query language

Due to these troubles, we have decided to design a new query language [7]. Several requirements have been taken on the proposed query language. It must be applicable for a general purpose. It must be closed, so it must use the RDF data model for solutions of its query. The language must also be strong as for selecting data so as for creating data.

Patterns have been used as a base of the language. They are widely used in other languages and they are very intuitive for a user. It is possible to say that everything in the new language is a pattern. Not only selecting data but whole evaluation is controlled by the patterns. However, many languages use patterns and they are weak, so there is the need to do the patterns stronger. The new language uses a simple way to do this. It makes it possible to name a pattern and to use it later by its name. So it is possible to make a recursive pattern. For example, a selection of a RDF Collection is simple. A pattern selects one element of the RDF Collection and then it uses itself on the rest of the RDF Collection recursively.

The troubles with blank nodes are resolved by introduction of local identification for a blank node, which make it possible to refer to a blank node. However, it still keeps the fact that a concrete name of a blank node is not significant and that it is not global.

The new language has been implemented for the semantic web infrastructure Trisolda [5] and it has got the name Tequila, which comes from Trisolda Query

Language. The next section describes the language in detail. Section 4 presents some examples of usage of the language.

2 The Tequila language

The main feature of the Tequila language is its ability to name a pattern (Section 2.3), which makes it possible to express a recursive pattern. Other powerful features include query composition (Section 2.5), usage of multiple data sources (Section 2.6), a good support for blank nodes (Section 2.10), and the ability to construct new triplets (Section 2.4).

2.1 Base of the language

The Tequila language is based on the SPARQL language, thus it is a pattern-based query language. It fully adopts the syntax and semantics of URIs (including qnames), literals, blank nodes, variables and comments. It also adopts the syntax and semantics of definitions of qname prefixes, which can be used in the prologue of a query.

The syntax of Tequila patterns is also similar to the syntax of SPARQL patterns, but its semantics is totally different. The main difference is that a solution of a Tequila query (or generally a pattern) is not a solution mapping, but a RDF graph is. Variable bindings, which are called a solution mapping in the SPARQL language, have effect only during evaluation of a pattern.

A pattern can have more than one solution. A solution can evoke some variable bindings. If a variable is bound to some value, then the next use of this variable is equivalent to use this value, so the binding influences solutions of other patterns. Due to this fact, it is reasonable speak about a solution (or solutions) of a pattern with respect to the actual variable bindings. A binding of a variable is active, until the solution that evokes this binding is refused.

In the next text, the word "solution" always means "a solution with respect to the actual variable bindings", but sometimes it is used the full form due to emphasis.

2.2 Base patterns

This subsection introduces base patterns, which are more or less adopted from the SPARQL language. Tequila language specific patterns are introduced in the next sections.

Query patterns. A Tequila query, which is fully called a query pattern, consists of the keyword **get** followed by a compound subpattern. A solution of a query pattern is the union of all solutions of its compound subpattern. In the case of a main query, a solution of the query may be represented as a sequence of solutions of the subpattern.

Compound patterns. A compound pattern consists of a sequence of subpatterns enclosed in braces. It is possible use any type of a pattern as its subpattern. The semantics of the evaluation of a compound pattern is a little bit complicated, but it is very important for the Tequila language. However, we hope that this semantics will be familiar for people who know the semantics of the Prolog language.

For the evaluation of a compound pattern, its subpatterns are subsequently evaluated. If some subpattern (with respect to the actual variable bindings) has no other solution, the evaluation goes back on the previous subpattern, it refuses the solution of this subpattern and it searches for the next solution of this subpattern. If it also has no other solution, the evaluation goes back again. If a solution is found, the evaluation continues in search for solutions of the next subpatterns. The evaluation of the next subpatterns starts from scratch, so solutions that have been found previously have no effect on the next solutions.

If a solution of the last subpattern is found, then a solution of the compound pattern is found and it is equal to the union of solutions of the subpatterns. In order to search for the next solution of the compound pattern (i.e., after a refusal of the previous solution of the compound pattern), the solution of the last subpattern is refused and it is searched for the next one. If the first subpattern has no other solution, then also the compound pattern has no other solution.

Triplet patterns. A triplet pattern is a basic construct for selecting data from a data source. The base form of a triplet pattern is similar to a triplet pattern from the SPARQL language. A solution of a triplet pattern is one triplet from a data source that matches the pattern with respect to the actual variable bindings. The next solutions return other triplets that match the pattern from the data source. When a solution (i.e. a triplet) is found, the unbounded variables of a triplet pattern are bound to values according to the values from the found triplet. Note that a blank node term do not have a variable character as in the SPARQL language, it is just identification of a blank node. See section 2.10.

Filter patterns. A FILTER pattern is the only direct way how to test values of variables. A FILTER pattern consists of the keyword **filter** followed by an expression and enclosed by the point. If a value of a FILTER expression is true (with respect to the actual variable bindings), then the FILTER pattern has just one solution, namely an empty RDF graph. If its value is false, then the FILTER pattern has no solution.

The semantics of a FILTER pattern might looks strange. However, in combination with the semantics of a compound pattern, it has a very good point. If a value of the expression is false, then the FILTER pattern has no solution and thus the evaluation of the compound pattern goes back, so that a bad solution is filtered. If a value of the expression is true, then a solution of the FILTER pattern is just an empty graph and thus the evaluation of the compound pattern goes through.

```
1 get
2 {
3     ?author ex:name "Milan Rufus".
4     optional ?author ?ex:born ?place.
5     get { ?book ex:author ?author. }
6 }
```

Listing 1.1. A simple example for a library system

Union patterns. A UNION pattern is useful for describing variants. It consists from two subpatterns connected by the keyword **union**. A solution of a UNION pattern is initially searched in the first subpattern. After what the first subpattern has no other solution, then the solution of the UNION pattern is searched in the second subpattern.

Optional patterns. An OPTIONAL pattern consists of the keyword **optional** followed by a subpattern. If the subpattern has any solution, then the evaluation of an OPTIONAL pattern is equivalent to the evaluation of its subpattern. If the subpattern has no solution, then a solution of the OPTIONAL pattern is an empty RDF graph. Hence, an OPTIONAL pattern has always at least one solution.

Example. Listing 1.1 shows an example of a simple query, which finds information about Milan Rufus. It finds a triplet with the object "Milan Rufus" and it binds his URI to the variable `?author`. Then it selects his birth place, if this information is included in a data source. Finally, it selects all his books.

2.3 Data selection

As mentioned above, the ability to select data is improved by the introduction of the named patterns.

Named patterns. A named pattern is identified by a URI. The URI is used only for identification and it need not meet any special conditions. A definition of a named pattern must precede a main query. It consists of a named pattern URI followed by a list of formal parameters (i.e. list of variables) enclosed in parentheses. A compound pattern that represents the body of the named pattern follows.

A use of a named pattern (i.e. the named pattern itself) consists of the keyword **use** followed by the URI of the named pattern and a list of actual parameters enclosed in parentheses. The evaluation of a named pattern is equivalent to the evaluation of the named pattern body that has been specified in the definition of the named pattern. However, there are some differences. Variables that are used in the named pattern body are local for the evaluation of the named pattern. Therefore, before the evaluation of the named pattern starts, a copy of its body is made and the variables of its formal parameters are unified

with the corresponding actual parameters. So if an actual parameter is a value, then a corresponding formal parameter variable is bound to this value. If an actual parameter is a variable, then a corresponding formal parameter variable is considered to be the same as the actual parameter variable.

A named pattern can also use a variable instead of a URI. If this variable is bound to a URI value, then evaluation of the named pattern is equivalent to the case in which it is directly used the URI value. If the variable is not bound to a URI value, then the named pattern has no solution.

If the URI that is used in a named pattern does not appear in any definition of a named pattern, then the named pattern has no solution.

Imports of named patterns. Some definitions of named patterns can be useful for many different queries. So it is useful to have a mechanism that allows reuse of them.

In the Tequila language, it is possible to write down definitions of named patterns to a file. This file then can be import to a query. An IMPORT directive consists of the keyword **import** followed by a file name enclosed by quotation marks. IMPORT directives follow PREFIX directives. PREFIX or IMPORT directives can be also used in an imported file, then an effect of the PREFIX directive is local for the file.

Example of a named pattern. Listing 1.2 shows the definition of the simple recursive named pattern `rdf:list`, which selects a RDF Collection. It is just an example of an import file, so it does not contain a main query.

The formal parameter `?N` is a resource that represents a RDF Collection. If it is different from `rdf:nil`, then the first case of the UNION pattern (line 5) matches the first element of the RDF Collection. In detail, the first triplet pattern (line 6) matches the `rdf:first` property triplet and the second triplet pattern (line 7) matches the `rdf:rest` property triplet of the element. Then the named pattern use itself (line 8) on the rest of the RDF Collection recursively. If the resource that represents the RDF Collection is `rdf:nil`, then the second case of the union matches it (line 11).

It is good to say that if somebody creates a cyclic RDF Collection, then this named pattern never ends its evaluation. It is possible to adapt this example to be cyclic safe, but then this example becomes more complicated and less effective.

2.4 Data construction

Now it is time to show how it is possible to construct a new triplet in the Tequila language. It is possible in just one way by using of a CONSTRUCT pattern.

Construct patterns. A CONSTRUCT pattern consists of the keyword **construct** followed by a triplet of variables, URIs, literals or blank nodes enclosed

```
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

rdf:list(?N)
{
    {
        ?N rdf:first ?F.
        ?N rdf:rest  ?R.
        use rdf:list(?R)
    }
    union
    {
        filter ?N = rdf:nil.
    }
}
```

Listing 1.2. The RDF Collection pattern

by the point. A CONSTRUCT pattern has just one solution, which is equal to its triplet with respect to the actual variable bindings. If some variable of the CONSTRUCT pattern is not bound to a value, then the variable is bound to a new (unique) blank node.

More CONSTRUCT patterns can be enclosed by braces following the keyword **construct**. The individual CONSTRUCT patterns then have not the keyword **construct**.

2.5 Query composition

The ability to create a query composition is very important for a query language. The Tequila language has a more general construct.

From pattern. A FROM pattern consists of two subpatterns connected by the keyword **from**. The second subpattern is used as a source pattern for the first subpattern. Initially, a solution of the source pattern is found and it is used as a data source for the first subpattern. Solutions of the FROM pattern are then searched in the first subpattern. If the first subpattern has no other solution, then the next solution of the source pattern is searched for and it is used as data source for the first pattern again. The evaluation of the first pattern starts from scratch. If the source pattern has no other solution, then the FROM pattern also has no other solution.

2.6 Multiple sources

Other useful property of a query language is the ability to combine multiple data sources. Also this property is supported by a pattern.

Source pattern. A SOURCE pattern consists of the keyword **source** followed by a URI of a data source. Usually, it is used in combination with a FROM pattern.

A SOURCE pattern has just one solution, which includes all triplets of the data source of the pattern. It is possible to use a variable instead of a URI of a data source. If the variable is bound to a URI, semantics is same. If the variable is not bound to a URI, solution is an empty RDF graph.

2.7 Other patterns

Now we introduce other patterns, which are also very important for the ability to use the language for a general purpose.

Match patterns. It is sometimes useful to bind variables to values according to the content of a data source. It is especially useful when we want to perform some conversion of a data source but we do not want to include the original content of the data source to a solution. Although some pattern can make this binding, it also returns some triplets in its solution. To avoid this, a MATCH pattern has been introduced.

A MATCH pattern consists of the keyword **match** followed by a subpattern. The evaluation of a MATCH pattern is equivalent to the evaluation of its subpattern, but with one difference. If the subpattern has a solution, then a solution of the MATCH pattern is an empty RDF graph. If the subpattern has no other solution, then the MATCH pattern also has no other solution. Thus only variable bindings are performed and the content of the solution is ignored.

Else patterns. An ELSE pattern is a way how to express a graph condition. An ELSE pattern consists of two subpattern connected by the keyword **else**. If the first subpattern has any solution, then the evaluation of the ELSE pattern is equivalent to the evaluation of the first subpattern. If the first subpattern has no solution, then the evaluation of the ELSE pattern is equivalent to the evaluation of the second subpattern.

Any patterns. For a recursive walk through a graph, it is useful to have the ability to select just one solution from all possible solutions. It is a goal of an ANY pattern. An ANY pattern consists of the keyword **any** followed by a subpattern. If its subpattern has a solution, then a solution of the the ANY pattern is an arbitrary solution of solutions of the subpattern and the pattern has no other solution. If its subpattern has no solution, then the ANY pattern has also no solution.

2.8 Syntactic sugar

There are some other constructs for more comfort. They just form a syntactic sugar and they do not increase a power of the language.

Where patterns. A WHERE pattern is a syntactic sugar for people familiar with the SQL-like syntax. A WHERE pattern consists of two subpatterns connected by the keyword **where**. The pattern `subpattern1 where subpattern2` is equivalent to the following construct:

```
match {subpattern2} {subpattern1}
```

Uniplet pattern. Sometimes, it is useful to create (or to select) only a single value, not a whole triplet. On the other hand, it is not good to change the RDF data model. So the Tequila language introduces the syntactic sugar for a uniplet.

The uniplet `node`, which can be used instead of a triplet in a triplet pattern or a CONSTRUCT pattern, is equivalent to the triplet `tql:shadowSubject tql:shadowPredicate node`[1].

Get-where-from query. If either a main query (i.e. query pattern which is not used as a subpattern) or a expression query has a following structure[2]:

```
get {get {getpattern} where {wherepattern} from {frompattern}}
```

then the outer keyword **get** with its braces can be omitted.

2.9 Pattern operator priority

By now, we only say that a pattern consists of some subpattern in the definitions and we do not say, which types of the subpatterns are possible to use. A restriction on types of subpatterns is needed to have an unambiguous grammar for the language.

It is possible to look at the keywords that are used for specifying patterns as on pattern operators and on the braces of compound patterns as on parentheses. The restriction on types of subpatterns is then done by an operator priority. The operators have following descendent priorities:

1. `any`, `match`, `optional`, `get`
2. `where`
3. `from`
4. `else`
5. `union`

2.10 Blank nodes.

According to the RDF specification, blank nodes have no names. However, for many reasons, it is necessary to introduce some kind of identification of blank nodes. One of the reasons is the need to identify blank nodes in a data source and to distinguish them from each other. Other reason is the necessity to have the ability to refer to a concrete blank node in a triplet CONSTRUCT pattern.

The syntax of the Tequila language as well as of the SPARQL language include a blank node term[3], but the semantics are totally different. In contrast to the SPARQL language, where the semantics of a blank node term is more similar to the semantics of a variable, a blank node term is just a local name of a blank node in the Tequila language. If a blank node term had the semantics

[1] The `tql:` prefix is bound to the URI http://ulita.ms.mff.cuni.cz/tequila/term#.
[2] The WHERE part or the FROM part can be missing.
[3] A qname with the prefix `_:`

similar to a variable, then a use of the blank node term would not be equivalent to a use of a variable bound to a blank node. It is other reason for the used semantics.

Although a blank node term means an identification of a blank node in the Tequila language, the semantics of the Tequila language keep the fact that blank nodes from two different data sources can never be equal. Also it keeps the fact that a concrete form of a blank node name that is used in a non-query data source is hidden for a user. These are the main ideas of blank nodes.

How does it do that? A name of a blank node has two components. The first component determines a data source name; the second component determines a local name of the blank node in the data source. Two blank nodes are equal, if their source names and their local names are equal. Therefore, two blank nodes from two different sources can never be equal, because their source names are different.

A blank node term written directly in a pattern determines the blank node that has the source name `[query]` and the local name that is equal to the local name of the qname of the blank node term. Therefore, a blank node term can never match a blank node from a non-query data source, so that the fact that a concrete form of a name of a non-query blank node is hidden for a user is kept. Finally, note that a blank node created by a CONSTRUCT pattern during binding unbound variables has the source name `[construct]`.

2.11 Expressions

A base of the syntax and semantics of the Tequila expressions is adopted from the SPARQL language. In addition, it is extended by various concatenations and a query expression. The LIKE operator from the SeRQL language is also adopted.

A Tequila expression is used in a FILTER pattern in the same way as in the SPARQL language. In addition, it is also possible to use it in a triplet pattern, a CONSTRUCT pattern or a named pattern instead of a RDF term. In this case, an expression must be enclosed by parentheses.

If a pattern contains an expression, then (before the evaluation of the pattern begin) the expression is evaluated first and its result value is used during the evaluation of the pattern instead of the expression as a RDF term. If the evaluation of the expression returns the error value, then the pattern has no solution.

It is important to point out that an element `?V` used in a pattern is a variable, but an element `(?V)` is an expression. If the variable is bound to a value, the effect is same. But if the variable is not bound to a value, then the evaluation of the expression returns the error value and the pattern has no solution.

Literal concatenations. A literal can be concatenated with other literal or a URI. The simple string concatenation is used; a datatype and a language tag are ignored, so a result of the literal concatenation is always a simple literal.

URI concatenations. A URI can be concatenated with other URI or a literal. The simple string concatenation is used again. The URI concatenation is important, for example, for creating a RDF Container, where it is necessary to create URIs that have the form `rdf:_n`.

Blank node concatenations. A result of a concatenation of two blank nodes is a blank node, which source name is obtained by concatenation of their source names and its local name is obtained by concatenation of their local names. An exception is a concatenation of two query blank nodes. In this case, the source name of a result is `[query]` again, because local names of query blank nodes are not considered as hidden for a user.

If the second operand is not a blank node, then the source name of the second operand is rated as `[query]` and the local name of the second operand is rated as a result of the `STR` function applied on the second operand. Therefore, it is not possible to get any other blank node from knowledge of some non-query blank node, so the fact that a name of a blank node is hidden for a user is still kept.

The blank node concatenation is a flexible way, how to create a new blank node and be able to refer to them. For example, if there is the need to create a blank node for each work group, then we simply create one by a concatenation of a blank node base with a work group URI. If there is need to have a blank node for each person, we concatenate other blank node base with a person URI. If a person is added to different work groups, then this procedure creates always the same blank node for this person. In addition, if there is the need to have a blank node for each combination of a person and a work group, a solution is also simple. We concatenate some blank node base with a work group URI and a person URI.

Determining a type of the + operation. The numeric addition and all types of the concatenations use the same symbol +. Determining a type of the operation depends on the type of its first operand.

Query expressions. It is possible to use a query pattern as a unary expression. A query expression is evaluated as a normal query pattern. If its solution has just one triplet, then the object of the triplet is a result value of the expression. If the solution has more than one triplet, then one triplet is selected from the solution randomly and its object is used. If the solution has no triplet, then a result value of the expression is `"false"^^xsd:boolean`.

3 Possible improvements

Apparently, the Tequila language is not finished yet and there is still space for improvements. Specially, it is in the following areas:

```
1  prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  prefix ex:   <http://www.example.org/term#>
3
4  ex:convert(?bag, ?list)
5  {
6      {
7          match any
8          {
9              ?bag ?pred ?item.
10             filter ?pred like ("" + rdf: + "_[0-9]+").
11         }
12
13         use ex:convert(?bag, ?sublist) from get
14         {
15             ?bag ?Y ?Z.
16             filter ?Y != ?pred && ?Z != ?item.
17         }
18
19         construct ?list rdf:first ?item.
20         construct ?list rdf:rest  ?sublist.
21     }
22     else
23     {
24         match ?list. from construct {rdf:nil.}
25     }
26 }
```

Listing 1.3. Converting a RDF Collection into a RDF Bag

1. The expressions should be more powerful. A good thing is a general support for different datatypes, not only for built-in datatypes.
2. A support of a RDF dataset will bring the ability to have multiple sets of triplets in one pattern solution. This may be useful for some type of problems.
3. A support for a RDF representation of a pattern will bring the ability to store the pattern in a data source. Therefore, it will be possible to store data together with the patterns, which can manipulate with these data.

4 Examples

This section presents two interesting examples. The first one shows how to convert a RDF Collection into a RDF Bag. The second one shows how to evaluate the aggregation function `max` without the need to have a built-in support for it.

4.1 Conversion of a RDF Collection into a RDF Bag.

This example is shown in Listing 1.3. The conversion is performed by the named pattern `ex:convert`, which has two formal parameters. The parameter `?bag` represents an input bag and the parameter `?list` represents an output collection. If the input bag is not empty, then the first case of the ELSE pattern (line 6) is a success. The ANY pattern (line 7) selects one item from the bag. Due to the MATCH pattern (line 7), only variable bindings are considered. Then the `ex:convert` pattern uses itself on the rest of the bag (line 13). The rest of the bag is generated by the query pattern (line 13). Finally, a new collection is made (line 19) from the selected item (variable `?item`) and from the created sub-collection (variable `?sublist`). The variable `?list` (line 19) is not bound

```
1  prefix tql: <http://ulita.ms.mff.cuni.cz/tequila/term#>
2
3  tql:max()
4  {
5      {
6          match any ?subj ?pred ?obj.
7
8          match ?max. from use tql:max() from get
9          {
10             ?X ?Y ?Z.
11             filter !(?X = ?subj && ?Y = ?pred && ?Z = ?obj).
12         }
13
14         {
15             filter ?obj > ?max || ?max = "none".
16             construct ?obj.
17         }
18         else
19         {
20             construct ?max.
21         }
22     }
23     else
24     {
25         construct "none".
26     }
27 }
```

Listing 1.4. Evaluation of the aggregation function max.

to a value, so the CONSTRUCT pattern bind it to a new blank node, which will represent the new collection.

If the bag is empty, then the second case of the ELSE pattern (line 23) is a success and the parameter `?list` is bound to the empty collection `rdf:nil` (line 24) by the MATCH pattern (line 24).

4.2 Evaluation of the aggregation function max.

This example, which is shown in Listing 1.4, is partially similar to the previous one. The aggregation function `max` is implemented by the named pattern `tql:max`. The maximum value is calculated from the objects of the triplets of an input data source.

If the data source is not empty, then the first case of the ELSE pattern (line 5) is a success. The ANY pattern and the MATCH pattern (line 6) select just one triplet and they bind the object of the triple to the variable `?object`. Then the `tql:max` pattern uses itself (line 8) on the rest of the input to get the maximum value of the rest. The rest of the input is generated by the query pattern (line 8). The maximum value of the rest is bound to the variable `?max` by the MATCH pattern (line 6). Then the variable `?max` and the variable `?object` are compared by the ELSE pattern (lines 18 and 15) and the biggest value is constructed (line 16 or 20). If the variable `?max` is bound to the value `"none"`, then a value bound to the variable `?object` is constructed.

If the data source is empty, then the second case of the ELSE pattern is a success (line 23) and the special value `"none"` is constructed.

5 Conclusion

In this paper, we introduce the new pattern-based query language Tequila. We demonstrate its power on several examples, which are difficult (or impossible) to express in other languages. Although the Tequila language is good suitable to solve these examples, it does not use any single-purpose constructs to express them. The language is designed to involve only general-purpose constructs.

The main construct of the language is a named pattern, which can be used to express a recursive query. This construct makes the Tequila language different from other query languages and makes the language so strong. On the other hand, it is possible to specify a query, for which the evaluation will never finish. It is one of disadvantages of the Tequila language, but it is the tax for the power.

Acknowledgements

This research was supported in part by Czech Science Foundation Project 201/09/068

References

1. ARQ - Property Paths, as accessible in February 2009.
   URL http://jena.sourceforge.net/ARQ/property_paths.html.
2. User Guide for Sesame,Chapter 6. The SeRQL query language (revision 1.2).
   URL http://www.openrdf.org/doc/sesame/users/ch06.html.
3. RDF Primer, W3C Recommendation, February 2004.
   URL http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.
4. SPARQL Query Language for RDF, W3C Recommendation, January 2008.
   URL http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.
5. Jiří Dokulil, Jakub Yaghob, and Filip Zavoral. Trisolda: The environment for semantic data processing. In *International Journal On Advances in Software 2008*, volume 1. IARIA, 2009.
6. Tim Furche, Benedikt Linse, François Bry, Dimitris Plexousakis, and Georg Gottlob. Rdf querying: Language constructs and evaluation methods compared. In *Reasoning Web, Second International Summer School 2006*, volume 4126 of *LNCS*. 2006.
   URL http://www.pms.ifi.lmu.de/publikationen/#PMS-FB-2006-33.
7. Jakub Galgonek. Query languages for the Semantic web. Master thesis, Charles University in Prague, Czech republic, 2008.
   URL http://siret.ms.mff.cuni.cz/galgonek/thesis/thesis.pdf.
8. Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of rdf query languages. In *Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, 2004.*, November.
   URL http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/rdfquery.pdf.
9. Michael Sintek and Stefan Decker. Triple - a query, inference, and transformation language for the semantic web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 364–378, London, UK, 2002. Springer-Verlag.
   URL http://portal.acm.org/citation.cfm?id=646996.711416.