LDScript: A Linked Data Script Language

Olivier Corby[1](✉), Catherine Faron-Zucker[2], and Fabien Gandon[1]

[1] Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis, France
{olivier.corby,fabien.gandon}@inria.fr
[2] Université Côte d'Azur, I3S, Inria, CNRS, Sophia Antipolis, France
faron@i3s.unice.fr

Abstract. In addition to the existing standards dedicated to representation or querying, Semantic Web programmers could really benefit from a dedicated programming language enabling them to directly define functions on RDF terms, RDF graphs or SPARQL results. This is especially the case, for instance, when defining SPARQL extension functions. The ability to capitalize complex SPARQL filter expressions into extension functions or to define and reuse dedicated aggregates are real cases where a dedicated language can support modularity and maintenance of the code. Other families of use cases include the definition of *functional* properties associated to RDF resources or the definition of procedural attachments as functions assigned to RDFS or OWL classes with the selection of the function to be applied to a resource depending on the type of the resource. To address these needs we define *LDScript*, a Linked Data script language on top of the SPARQL filter expression language. We provide the formal grammar of the syntax and the Natural Semantics inference rules of the semantics of the language. We also provide a benchmark and perform an evaluation using real test bases from W3C with different implementations and approaches comparing, in particular, script interpretation and Java compilation.

Keywords: SPARQL · Linked Data · Programming · Semantic Web

1 Introduction

RDF is the standard framework recommended by the W3C to represent and exchange Linked Data on the Web. It is associated with RDF Schema and OWL for ontology-based modelling on the semantic Web and with SPARQL for data and ontology querying. The development of the Web of data opens up a wide range of use cases where, in addition to the existing standards, Semantic Web programmers would benefit from having a dedicated programming language enabling them to define functions on RDF terms or RDF graphs. This is the case, for instance, when defining SPARQL extension functions implemented for a special purpose and domain or application dependent. This would also be needed to capitalize a complex SPARQL filter expression or the definition of special purpose extension aggregates to be reused across queries or sub-queries.

C. d'Amato et al. (Eds.): ISWC 2017, Part I, LNCS 10587, pp. 208–224, 2017.
DOI: 10.1007/978-3-319-68288-4_13

Another kind of use cases is the definition of *functional* properties associated to RDF resources, the results of which are computed on demand. For instance, the value of the `surface` property of a rectangular object could be computed as the product of the values of its width and length properties; the age of a person could also be computed from her date of birth. These use cases can also be extended to the definition of procedural attachments as functions assigned to RDFS or OWL classes. The selection of the function to be applied to a resource can then be made dependent on the types (classes) of the resource.

The requirements we propose for a programming language enabling such definitions of functions are:

– The objects of the language are RDF terms (URIs, blank nodes and literals), RDF triples and graphs as well as SPARQL query solutions. This is required to facilitate the access and manipulation of Linked Data in their native model without adding the burden of parsing, encapsulating, mapping, and serializing in other programming languages.
– The statements of the language include SPARQL filter expressions and SPARQL queries (the SELECT and CONSTRUCT query forms). This is required to be able to directly leverage in the program these operations that are tailored to Linked Data access and processing. Again, encapsulating and mapping these operations to other programming paradigms can make them clumsy and inefficient.
– The language provides function definition and function call. This is the core motivation of our proposal to add all the needed primitives to program and execute functions inside RDF frameworks.
– Functions can be exchanged and shared among Semantic Web plaforms and are interoperable.

In this paper, we address the research question: *Can we define a standard-based programming language that meets the above described requirements?* To answer this question, we define *LDScript*, a Linked Data script language on top of the SPARQL filter expression language, taking advantage of the fact that the SPARQL filter expression language potentially enables users to express the definitions of functions. We effectively define a *programming language* on top of SPARQL filter expression language. Syntactically, it consists in a couple of additional statements: a FUNCTION statement enabling users to define functions and a LET statement enabling them to define local variables. We present the syntax and semantics of LDScript as well as an implementation and we illustrate the simplicity as well as the expressive power of this extension with real cases of LDScript functions. We also provide a benchmark and we report the results of an evaluation of LDScript using real test bases from W3C with different implementations and approaches comparing, in particular, script interpretation and Java compilation.

This article is based on an initial research report [4] and presents our scientific contributions with the following plan. In Sect. 2 we present state-of-the-art approaches to define extension functions. In Sect. 3 we introduce LDScript with an overview of this language. In Sect. 4 we formally define the syntax and semantics of LDScript. In Sect. 5 we study several use cases and we show how they can easily be addressed by using LDScript. In Sect. 6 we evaluate the efficiency of different implementations of LDScript on various test cases. In Sect. 7 we conclude and draw some perspectives of our work.

2 Related Work

Although no contribution so far has directly addressed the research question we target in this paper, a number of previous works are somehow related to the issue. SPIN is a W3C member submission which proposes a SPARQL-based rule and constraint language and, additionally, enables one to represent both SPARQL queries (SPIN templates) and SPARQL extension functions (SPIN functions) [6]. SPIN is represented in RDF. In SPIN, a SPARQL extension function is identified by a resource of type `sp:Function` (with `sp` the prefix denoting the SPIN namespace) which is linked by property `sp:body` to its definition as a SPARQL query of the form SELECT or ASK.

Jena provides a `java` URI scheme for naming and accessing SPARQL extension functions implemented in Java. This enables one to dynamically load the bytecode implementing the function. By convention, the location of the Java class must be found in the Java classpath and the local name of the function must be the name of the Java class implementing it[1]. Here is an example of a SPARQL query using an extension function `f:myTest` implemented in Java. It filters the RDF triples for which `f:myTest` returns true when called with the triple's subject and object as parameters:

```
PREFIX f: <java:app.myFunctions.>
SELECT ?x WHERE { ?x ?p ?y FILTER f:myTest(?x, ?y) }
```

When compared to the above stated requirements for a Semantic Web programming language, Jena relies only on Java for extension functions and the code of the function has to deal with mappings from the Linked Data models and syntaxes to the object oriented models and syntax of Java.

G. Williams [9] proposes to implement SPARQL extension functions in JavaScript as an agreed-upon programming language, and to share implementations among query engines by using an embedded JavaScript interpreter. Functions are identified by URLs and their source code may be retrieved at run time by dereferencing their URL. It relies on an RDF schema enabling one to describe a SPARQL extension function and retrieve its source code at run time by dereferencing its URL. Here is an example of RDF statements describing a SPARQL extension function to compute a geographical distance in kilometers. The location of its JavaScript source code is the value of property `ex:source` (with `ex` the namespace prefix of the extension function schema) and the function name in the source code that should be called to execute the extension function is the value of property `ex:function`.

[1] https://jena.apache.org/documentation/query/writing_functions.html.

```
<http://example.com/functions/distance> a ex:Function;
  dc:description"Geographic distance in km";
  ex:source <http://example.com/distance.js>;
  ex:function "gdistance" .
```

When compared to our proposed requirements, the code of the function has to deal with mappings from the Linked Data models and syntaxes to the object oriented model and syntax of JavaScript.

M. Atzori [1] proposes to implement SPARQL extension functions based on both a generic extension function `wfn:call` and the SPARQL SERVICE clause. Function `wfn:call` is similar to the Lisp *funcall* function and takes the extension function to be evaluated as its first argument. Any occurrence of the `wfn:call` function is replaced by a SERVICE call to delegate the evaluation of the extension function to the SPARQL endpoint implementing the function. The SPARQL endpoint's IRI is computed from the extension function's IRI, based on a Function-to-Endpoint IRI pattern. When compared to the requirements for a Semantic Web programming language, this proposal does not provide a language but rather an RPC-like hook to call functions that have to be hosted and executed in a separate server process.

When compared to these four state-of-the-art proposals, the key idea of our proposal described in the following is to extend the SPARQL language, and more precisely its filter language, in order to enable the definition of extension functions in the SPARQL filter language *itself* and using its native syntax. With LDScript we propose a self-contained and Linked Data oriented programming language for extension functions.

Relatedly, SPARQL-Generate [7] also extends SPARQL with a few constructs with the specific target of enabling the generation of RDF from heterogeneous sources. It is a template language whereas our proposal is a programming language. In fact we could emulate SPARQL-Generate with LDScript and STTL SPARQL Template Transformation Language [3].

PL/SQL[2] is a programming language that is tightly integrated with the SQL query language for relational databases. LDScript is designed with similar goals but for triple stores.

Finally, it must be noted that Prolog may be worth considered for entailments purpose which is not the topic of this work.

3 Overview of LDScript

Our goal is to define functions, the objects of which are Linked Data entities (URI, RDF literals, RDF triples, etc.) with the main objective of defining SPARQL extension functions and possibly extend SPARQL itself. One possibility is to rely on an existing programming language, e.g. Java, and use the specific API of the SPARQL implementation of the RDF entities. However, this approach has several weaknesses. First, it is not interoperable because other

[2] http://www.oracle.com/technetwork/database/features/plsql/.

SPARQL implementations of RDF entities do not use the same API. Hence, the functions cannot be reused across different SPARQL implementations. Second, one does not benefit of SPARQL native function library dedicated to RDF terms (functions `isURI`, `isBlank`, `isLiteral`, `datatype`, `strdt`, `strlang`, `langMatch`, `uri`, `bnode`, etc.) Third, one must switch back and forth from SPARQL to (e.g.) Java environments with their compiler, project management environment, etc., and link the compiled functions to the SPARQL interpreter.

Another possibility would be to design a specific programming language the object of which would be RDF entities with a library implementing SPARQL functions. The weakness of this approach would be that users would have to learn yet another programming language.

We propose LDScript, a third way that reconciles the above two approaches: a programming language whose objects are RDF entities, SPARQL compatible and embedding the complete SPARQL function library.

LDScript primarily relies on the SPARQL filter expression language. A SPARQL filter is either (a disjunction or conjunction of) a relational expression or a call to a built-in or externally defined boolean function. Among the built-in SPARQL functions stands the IF ternary function which evaluates the first argument and returns the value of the second argument if the first argument results in an effective value of true, or else the value of the third argument. A SPARQL filter restricts the solutions of a graph pattern matching to those satisfying the constraint it expresses: the filtered solutions result in the boolean value *true* when substituted into the filter expression.

We propose to define functions by taking advantage of the fact that the SPARQL filter language enables users to define expressions. Handbooks of programming languages explain that typical programming languages include as commands: variables declaration, assignment, call, return, sequential blocks, iterative commands and *if* statements. For this reason, in this section we will introduce the corresponding statements in LDScript. LDScript primitives are also directly descending from the historical definitions of *function* and *function definition* as introduced by John McCarthy [8].

Here are the namespaces and prefixes used in the definitions:

```
prefix xt: <http://ns.inria.fr/sparql-extension/>
prefix us: <http://ns.inria.fr/sparql-extension/user/>
prefix rq: <http://ns.inria.fr/sparql-function/>
prefix dt: <http://ns.inria.fr/sparql-datatype/>
prefix ex: <htp://example.org/>
```

3.1 LDScript Function Definition

In LDScript, a *function definition* starts with the FUNCTION keyword. The first argument of the declaration is the name (a URI) of the function being defined followed by its argument list. The variables in the argument list play the usual role of function arguments. The second argument is the body of the function being defined. It is an LDScript expression or a sequence of expressions. For example, the *factorial* function us:fac is defined as follows, by using the SPARQL IF built-in function and embedding a recursive call:

```
FUNCTION us:fac(?n) {
  IF (?n = 0, 1, ?n * us:fac(?n - 1)) }
```

Here is another example of function definition. A call to the us:status function returns the status (married or single) of the resource given as parameter. Its definition uses the SPARQL built-in IF function and EXISTS operator.

```
FUNCTION us:status(?x) {
IF (EXISTS { ?x ex:hasSpouse ?y } || EXISTS { ?y ex:hasSpouse ?x },
    ex:Married, ex:Single) }
```

A call to a defined function returns the result of the evaluation of its body, with its arguments bound by the function call. In the body, the arguments are *local* variables in the sense that the variable bindings are local to the body of the function and exist only during the execution of the function. For instance, according to its above definition, a call to function us:fac will return the value returned by a call to the IF SPARQL built-in function form, with a given value for variable ?n.

The language for defining the body of a function is LDScript, i.e. the SPARQL filter expression language extended with statements presented in this document. Hence, to define extension functions, LDScript programmers can make use of the expressivity of the whole SPARQL filter expression language. In particular, this includes built-in SPARQL functions, among which the IF function form enabling to consider alternatives, and the EXISTS operator to test the existence of graph patterns. This also includes extension functions defined in LDScript or externally defined in another language (as SPARQL allows it). LDScript provides overloading in the sense that it enables users to define several functions with the same name and a different number of arguments.

For example, the SPARQL query below comprises the definition of function us:fac in a FUNCTION clause and the WHERE clause embeds a call to this function to search the resources whose income is greater or equal to $10! = 3{,}628{,}800$.

```
SELECT ?x ?i
WHERE { ?x ex:income ?i FILTER (?i >= us:fac(10)) }
FUNCTION us:fac(?n) { IF (?n = 0, 1, ?n * us:fac(?n - 1)) }
```

3.2 Local Variable Declaration

LDScript function definitions can embed *local variable declarations*. These are expressed in a LET statement. Its first argument declares a local variable and its value; its second argument is an expression which is evaluated with the transient binding of the local variable declared. After completion of the expression evaluation, the binding vanishes. The result of a LET statement is the result of its second argument. For instance, the example LET statement below returns the pretty-printing of the current date, e.g. "29/01/2017".

```
LET (?n = now()) { concat(day(?n), "/", month(?n), "/", year(?n)) }
```

A LET statement can also execute a SPARQL query in its first argument and bind the *select* variables with the first query solution. For instance, the following function uses a LET statement to return the first retrieved type of a resource.

```
FUNCTION us:type(?s){ LET (SELECT ?t WHERE {?s a ?t}){ ?t }}
```

An alternative syntax enables users to explicit the (subset of) variables to be bound. The binding of the variables of the LET statement is done by name (not by position).

```
FUNCTION us:type(?s){ LET (((?t)) = SELECT ?t WHERE {?s a ?t}){?t}}
```

3.3 Loop Statements

In order to iterate a statement on the elements of a list of values, LDScript is provided with the FOR loop statement. As an example, the following function iteratively calls the `xt:display` function on the prime numbers among a given list of natural numbers.

```
FOR (?n IN xt:list(1, 2, 3, 4, 5)) {
  IF (us:prime(?n)) { xt:display(?n) } }
```

The FOR statement can iterate on the results of a SPARQL SELECT or CONSTRUCT query. In the case of a CONSTRUCT query, it iterates on the triples of the graph. For instance, the following function iteratively calls the `xt:display` function on RDF triples of the form `?x a foaf:Person`.

```
FOR ((?s, ?p, ?o) IN CONSTRUCT WHERE { ?x a foaf:Person }) {
  xt:display(?s, ?p, ?o) }
```

Note that there are a lot of possibilities to integrate queries as expressions. We have chosen a generic principle that matches LET/FOR statements and SELECT/CONSTRUCT queries with the same design pattern. In addition this choice has the advantage of being generalizable to other objects such as lists, triples and graphs:

```
FOR ((?fst, ?snd) IN ?listOfPairs)
LET ((?s, ?p, ?o) = ?triple)
```

The model and the implementation take such objects as solution mappings and graphs into account. They are implemented using a pointer datatype, that is a kind of blank node pointing to an object. As a blank node it can pass through all statements. As a pointer, it is exploited by pattern matching expressions like in the above LET and FOR statements.

3.4 Function Evaluation

LDScript is provided with the FUNCALL function to call a function whose name is dynamically computed. For instance, the following example retrieves the name of the appropriate `us:surface` method and applies it to argument `?x`.

```
FUNCALL(us:method(us:surface, ?x), ?x)
```

LDScript is provided with the APPLY function to iteratively call a binary function on a list of arguments. For example, the following function call enables to compute the sum of the elements of a list of numbers with the binary `rq:plus` function.

```
APPLY(rq:plus, xt:list(1, 2, 3, 4, 5))
```

3.5 List Datatype

LDScript is provided with a `dt:list` datatype to manage lists of values. A `dt:list` datatype value is a list whose elements are RDF terms: URIs, literals, blank nodes or sublists of type `dt:list`. The elements of a list need not be of the same kind, neither of the same datatype. The `dt:list` datatype comes with a set of predefined functions among which `xt:size` returns the size of the list, `xt:get` returns the n^{th} element, `xt:sort` sorts the list according to the ORDER BY rules of SPARQL, `xt:iota` returns the list of n first integers, `xt:cons` adds an element to the head of the list, etc.

The MAPLIST function enables one to apply a function to the elements of a list and return the list of the results. For instance, the call to function MAPLIST shown below returns the list of the results of the calls to function `us:fac` on the first ten integers.

```
MAPLIST(us:fac, xt:iota(10))
```

There are several variants of the MAPLIST function: MAP applies a function and returns true, MAPSELECT returns the list of elements such that the boolean function returns true.

4 LDScript Formal Definition

The previous section gave an overview of LDScript. In this section we formally define the syntax and semantics of this language.

4.1 LDScript Syntax

LDScript grammar is based on SPARQL[3]. The definition of `BuiltInCall` is extended with LET, FOR, MAP, FUNCALL and APPLY statements.

[3] http://www.w3.org/TR/sparql11-query/#grammar.

```
Function ::= 'FUNCTION' iri ('()' | VarList) Body
Body ::= '{' '}' | '{' Expression (';' Expression)* '}'
VarList ::= '(' Var (',' Var)* ')'
BuiltInCall ::= SPARQL_BuiltInCall | IfThenElse
| 'LET' '(' (Decl (',' Decl) *       | SelectQuery) ')' Body
| 'FOR' '(' (VarOrList 'IN' ExpQuery | SelectQuery) ')' Body
| Map   '(' iri ',' Expression ')'
| 'funcall' '(' Expression (',' Expression)* ')'
| 'apply' '(' iri ',' Expression ')'
IfThenElse ::= 'IF' '(' BuiltInCall ')' Body
  ( 'ELSE' ( Body | IfThenElse ) ) ?
Decl ::= VarOrList2 '=' ExpQuery
VarOrList ::= Var | VarList
VarOrList2 ::= VarOrList | '(' VarList ')'
ExpQuery ::= Expression | SelectQuery | ConstructQuery
Map ::= 'MAP' | 'MAPLIST' | 'MAPSELECT'
```

4.2 LDScript Semantics

As usually done for programming languages, we formally defined the semantics of the core of LDScript by a set of Natural Semantics inference rules [5]. These rules enable us to define the semantics of the evaluation of the expressions of the language in an environment with variable bindings. The bottom of the rule is the conclusion and the top is the condition. The $\vdash$ symbol states that the expression on the right side is evaluated in the environment given on the left side. The $\rightarrow$ symbol represents the evaluation of the expression on the left side into the value on the right side. An environment is a couple (μ, ρ) where μ is the basic graph pattern (BGP) solution mapping and ρ represents local variable bindings. In addition, the environment must contain a reference to the SPARQL dataset, which is not detailed hereafter.

Rule 1 states that local variables are evaluated within ρ which is managed as a stack, latest variable binding first; rule 2 states that global variables are evaluated within μ which is a BGP solution. Rules 3 and 4 specify the evaluation of function calls. The $\Rightarrow$ symbol represents a function definition lookup for the function name on the left side. The solution mapping environment is empty during function body evaluation: there are no global variables. Each function call creates a fresh environment with function parameters (if any) as local variables. Rule 5 specifies the evaluation of the LET clause which declares a local variable to be added to environment ρ. Hence, a declared local variable may hide a function parameter or a BGP variable. BGP variables are accessible in a LET statement (e.g. in a filter), but recall that, inside a function body, the μ environment is empty. Rules 6 & 7 specify the evaluation of a LET clause with a SPARQL query of the SELECT form which binds the variables of the SELECT clause and evaluates the expression. Rule 8 specifies the evaluation of the FOR statement by evaluating the first expression that returns a list of values and then binds the variable successively with each element of this list and evaluates the second expression with each local binding. Since this statement does not compute a result in itself, it always returns *true*. Rules 9, 10, 11, 12, and 13 specify MAP, FUNCALL and APPY statements. Rule 14 specifies the evaluation of an LDScript expression. The semantics is that of standard SPARQL expression evaluation, except that the overall environment comprises an environment for local variables in addition to the standard environment for BGP variables.

$$\frac{}{\mu, \rho[x = v] \ \vdash \ x \ \rightarrow v} \tag{1}$$

$$\frac{x \notin \rho}{\mu[x = v], \rho \ \vdash \ x \ \rightarrow v} \tag{2}$$

$$\frac{f() \Rightarrow f() = body \ \wedge \ \emptyset, \emptyset \ \vdash \ body \ \rightarrow res}{\mu, \rho \ \vdash \ f() \ \rightarrow res} \tag{3}$$

$$\frac{\begin{array}{l} f(e_1, ... e_n) \Rightarrow f(x_1, ... x_n) = body \\ \forall i \in \{1..n\} \ \ \mu, \rho \ \vdash \ e_i \ \rightarrow v_i \\ \emptyset, [x_1 := v_1; ... x_n := v_n] \ \vdash \ body \ \rightarrow res \end{array}}{\mu, \rho \ \vdash \ f(e_1, ... e_n) \ \rightarrow res} \tag{4}$$

$$\frac{\mu, \rho \ \vdash \ e_1 \ \rightarrow v_1 \ \wedge \ \mu, \rho\,[x := v_1] \ \vdash \ e_2 \ \rightarrow res}{\mu, \rho \ \vdash \ let(x = e_1, \ e_2) \ \rightarrow res} \tag{5}$$

$$\frac{\begin{array}{l} \mu, \rho \ \vdash \ query \ \rightarrow \{\mu_1\} \cup \Omega \\ \mu, \rho.\mu_1 \ \vdash \ exp \ \rightarrow v \end{array}}{\mu, \rho \ \vdash \ let(query, exp) \ \rightarrow v} \tag{6}$$

$$\frac{\begin{array}{l} \mu, \rho \ \vdash \ query \ \rightarrow \emptyset \\ \mu, \rho \ \vdash \ exp \ \rightarrow v \end{array}}{\mu, \rho \ \vdash \ let(query, exp) \ \rightarrow v} \tag{7}$$

$$\frac{\begin{array}{l} \mu, \rho \ \vdash \ e \ \rightarrow (v_1, ... v_n) \\ \forall i \in \{1..n\} \ \ \mu, \rho\,[x := v_i] \ \vdash \ b \ \rightarrow r_i \end{array}}{\mu, \rho \ \vdash \ for(x = e, \ b) \ \rightarrow true} \tag{8}$$

$$\frac{\begin{array}{l} \mu, \rho \ \vdash \ e \ \rightarrow (v_1, .. v_n) \\ \forall i \in \{1..n\} \ \ \mu, \rho \ \vdash \ f(v_i) \ \rightarrow r_i \end{array}}{\mu, \rho \ \vdash \ map(f, e) \ \rightarrow true} \tag{9}$$

$$\frac{\mu, \rho \ \vdash \ e \ \rightarrow f \ \wedge \ \mu, \rho \ \vdash \ f(e_1, .. e_n) \ \rightarrow v}{\mu, \rho \ \vdash \ funcall(e, e_1, .. e_n) \ \rightarrow v} \tag{10}$$

$$\frac{\begin{array}{l} \mu, \rho \ \vdash \ e \ \rightarrow (v_1, .. v_n) \\ \mu, \rho \ \vdash \ apply(f, (v_1, .. v_n)) \ \rightarrow v \end{array}}{\mu, \rho \ \vdash \ apply(f, e) \ \rightarrow v} \tag{11}$$

$$\frac{\mu, \rho \vdash f() \rightarrow v}{\mu, \rho \vdash apply(f, ()) \rightarrow v} \tag{12}$$

$$\frac{\begin{array}{l}\mu, \rho \vdash apply(f, (v_2, ..v_n)) \rightarrow r \\ \mu, \rho \vdash f(v_1, r) \rightarrow v\end{array}}{\mu, \rho \vdash apply(f, (v_1, ..v_n)) \rightarrow v} \tag{13}$$

$$\frac{sparql(\mu, \rho \vdash exp \rightarrow v)}{\mu, \rho \vdash exp \rightarrow v} \tag{14}$$

The above described Natural Semantics inference rules defining the semantics of the evaluation of LDScript expressions should be completed with the Natural Semantics inference rules defining the semantics of the evaluation of SPARQL queries within LDScript. These should be written according to Sect. 18.6 of the SPARQL recommendation[4].

5 Examples of Use Cases

In this section, we present the definition of several LDScript extension functions showing the expressive power and usability of the language. Some additional examples can be found at: http://ns.inria.fr/sparql-extension.

5.1 Extended Aggregates

LDScript enables programmers to simply define extended aggregates with a simple extension of the SPARQL interpreter. We introduce the `aggregate` function which is as an additional generic aggregate. This function takes as arguments an expression (e.g. `?v` in the example below) and aggregates the results of the expression into a `dt:list`. Then we can call a custom aggregation function with this list as argument. The example below defines `sort_concat`, a variant of the `group_concat` aggregate which sorts the elements before concatenation occurs. The `rq` prefix and namespace are used to assign a URI to each SPARQL standard function, hence `rq:concat` function is SPARQL `concat` function.

```
SELECT (aggregate(?v) AS ?list) (us:sort_concat(?list) AS ?res)
WHERE { ?x rdf:value/rdf:rest*/rdf:first ?v }
FUNCTION us:sort_concat(?list){ apply(rq:concat, xt:sort(?list)) }
```

[4] https://www.w3.org/TR/sparql11-query/#sparqlAlgebraEval.

5.2 Procedural Attachment

LDScript enables programmers to perform procedural attachment to RDF resources. The idea is to annotate the URI of a function to declare that it is a method associated to a class. In the example RDF annotation below, two functions are described, `us:surfaceRectangle` and `us:surfaceCircle` which compute surfaces; it states that they implement the method `us:surface` for `us:Rectangle` and `us:Circle` respectively.

```
us:surfaceRectangle a xt:Method ; xt:name us:surface ;
  xt:input (us:Rectangle) ; xt:output xsd:double .
us:surfaceCircle a xt:Method ;  xt:name us:surface ;
  xt:input (us:Circle) ; xt:output xsd:double .
```

Then, the method `us:surface` can be called on a resource as follows, without mentioning its type:

```
SELECT * (funcall(xt:method(us:surface, ?x), ?x) as ?m)
WHERE { ?x a us:Figure }
```

The `xt:method` function is defined below. It retrieves the function `?fun` implementing the method `?m` by finding the type `?t` of the resource (line 3) and then finding a method attached to the type, or a superclass of the type (line 4). In the latter case, this implements method inheritance following the `rdfs:subClassOf` relation.

```
01  FUNCTION xt:method(?m, ?x){
02    LET (SELECT * WHERE {
03      ?x rdf:type/rdfs:subClassOf* ?t .
04      ?fun a xt:Method ; xt:name ?m ; xt:input(?t)})
05    { ?fun } }
```

Finally, the methods `us:surfaceRectangle` and `us:surfaceCircle` are defined as follows:

```
FUNCTION us:surfaceRectangle(?x){
  LET (SELECT * WHERE {?x us:width ?w ; us:length ?l})
  { ?w * ?l } }
FUNCTION us:surfaceCircle(?x){
  LET (SELECT * WHERE {?x us:radius ?r})
  { 3.14159 * power(?r, 2) } }
```

Below are some RDF descriptions of figures for which we can now compute the surface using the above defined procedural attachment.

```
us:Circle    rdfs:subClassOf us:Figure .
us:Rectangle rdfs:subClassOf us:Figure .
us:cc a us:Circle ; us:radius 1.5 .
us:rr a us:Rectangle ; us:width 2 ; us:length 3 .
```

5.3 Calendar

We wrote LDScript functions to compute the weekday of a date literal of type `xsd:date`[5] and functions to generate a calendar given a year[6]. We designed a dynamic Web page generated from DBpedia events where events of a given year are placed into the calendar[7]. The performance of LDScript is such that the Web page is computed and displayed in real time.

5.4 SHACL

As part of a SHACL validator, we wrote an interpreter for W3C SHACL Property Path language[8]. This real use case shows that LDScript enables users to write such programs in a few lines: the function below recursively rewrites a property path shape expression `?pp` as an LDScript list.

```
FUNCTION sh:path(?shape, ?pp){
  LET (SELECT ?shape ?pp ?q ?path WHERE {
        GRAPH ?shape {
            # rdf:rest is for a sequence
            values ?q {
                sh:inversePath sh:alternativePath
                sh:zeroOrMorePath sh:oneOrMorePath
                sh:zeroOrOnePath rdf:rest }
            ?pp ?q ?path } } ) {
      IF (! bound(?q)){
        IF (isURI(?pp)){ ?pp } ELSE { error() }}
      ELSE IF (?q = rdf:rest) {
        xt:list(sh:sequence, sh:list(?shape, ?pp)) }
      ELSE { xt:list(?q, sh:path(?shape, ?path)) } } }
```

6 Implementation and Evaluation

We implemented LDScript using the SPARQL interpreter of the Corese Semantic Web Factory [2]. The FUNCTION, LET and other statements are implemented by the SPARQL parser, compiler and interpreter. Should an error occur, function evaluation resumes in error mode, according to the same model as SPARQL evaluation error: in a FILTER, the filter fails; in a SELECT or a BIND clause, *"the variable remains unbound for that solution but the query evaluation continues"*[9].

[5] http://ns.inria.fr/sparql-extension/calendar.
[6] http://corese.inria.fr/srv/template?transform=st:calendar.
[7] http://corese.inria.fr/srv/template?profile=st:calendar3.
[8] http://ns.inria.fr/sparql-extension/datashape.
[9] http://www.w3.org/TR/2013/REC-sparql11-query-20130321/#assignment.

6.1 Generic Evaluation and Validation

LDScript has been validated on the functions described in Sect. 5 and extensively used in several STTL transformations on a server available online[10] [3]. We measured the performance of our implementation of LDScript on the execution of (1) a recursive Fibonacci function to test an exponential number of calls, (2) on the Bubble sort algorithms to evaluate loops and (3) on statistics functions for the calculation aspect. We compared with Java and JavaScript implementations which, of course, benefit from many years of optimizations. The goal of this first evaluation was just to show that a direct implementation on top of a SPARQL engine without dedicated optimizations is already usable. The results are shown using a logarithmic scale in Fig. 1. The extension function `fib` implements the Fibonacci sequence. The computation of fib(35) = 9227465 on a HP EliteBook laptop takes 3650 ms in LDScript, 68.8 ms in JavaScript and 24.9 ms in Java.

```
FUNCTION us:fib(?n) {
  IF (n <= 2, 1, us:fib(?n - 2) + us:fib(?n - 1)) }
```

Bubble sort on an array of 1000 items takes 580.1 ms in LDScript, 6 ms in JavaScript and 9.5 ms in Java. Three statistic functions together (average, median and standard deviation) on an array containing 100000 integer values take 157.1 ms in LDScript, 80.5 ms in JavaScript and 6.9 ms in Java.

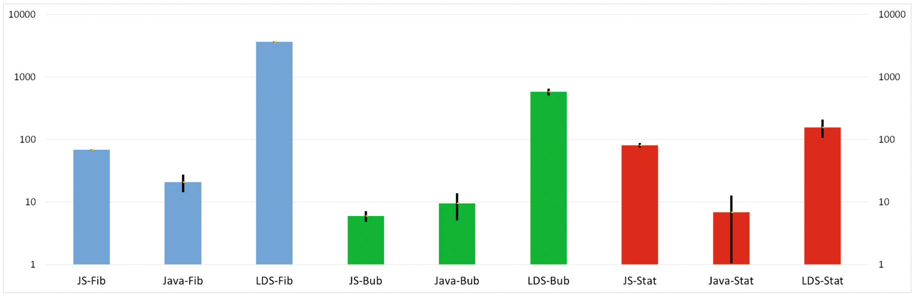

Fig. 1. Comparison of mean times and their mean absolute differences for JavaScript (JS), Java and LDScript (LDS) computing recursive Fibonacci (-Fib in Blue), Bubble sort (-Bub in green) and statistics (-Stat in red) using a logarithmic scale. (Color figure online)

The above described first implementation of LDScript is a proof of concept where we focused on the design and the semantics. Although focusing on performance is future work, we already took the first step of providing and comparing alternative implementations, as described in the following.

[10] http://corese.inria.fr.

6.2 Java Compiling and Specific Evaluation

In addition to LDScript interpreter, we have written a compiler from LDScript to Java. This enables us, among other things, to evaluate the performance of the LDScript interpreter compared to the equivalent Java code. We have compared performances on three LDScript programs with different programming characteristics (e.g. recursive calls) as well as a full implementation of the SHACL standard as a real use case. We therefore have four programs for this last evaluation: the recursive Fibonacci function, a calendar function that computes the weekday of a given date, a parser of Roman to Arabic numbers and reverse, and a SHACL validator. As a result, the execution time of the compiled Java code is higher than pure Java because the compiled code operates on RDF terms with XSD datatypes whereas a native Java code would operate on Java datatypes. Our experiments show that the LDScript interpreter can be as fast as the Java code produced by the LDScript compiler and sometimes slightly faster. One exception is the generated Java code for the Fibonacci function which is much more efficient, e.g. by a factor of 20 for fib(35). This is due to the optimized implementation of function call and recursion in Java. For the three other use cases, the performances of the LDScript interpreter and the Java code are equivalent. For the SHACL validator, we have tested it on the 97 SHACL test cases from W3C. The Java code runs in 5.2 s while the LDScript interpreter runs in 4.93 s (average time of ten runs after warmup on an HP laptop). The Java source code is 3220 lines and 37055 bytes, the LDScript source code is 2009 lines and 17819 bytes. The complete codes, the evaluation report and the table of performance are documented and available online[11].

7 Conclusion and Future Work

Dedicated programming language enabling Semantic Web programmers to define functions on RDF terms, triples and graphs or SPARQL query results can facilitate the development and improve the reuse and maintenance of the code produced for Linked Data. To address these needs we detailed in this article a lightweight extension of SPARQL filter expression language to enable the *definition* of extension functions and we defined a Linked Data script language on top of the SPARQL filter expression language. Compared to the state-of-the-art we directly extend the SPARQL language in order to enable the definition of extension functions in the SPARQL language *itself* and using its native syntax, building on a well-known and widely accepted component of the Web of data. The key point of our proposal is that a programming language can easily be integrated in SPARQL to define extension functions.

LDScript has all the features of Turing-complete programming languages: constants, lists, variables, expressions, boolean connectors, if-then-else, iteration, local variable definition, function definition, function call, recursion. LDScript is a programming language where values are RDF terms, hence its use to define

[11] http://ns.inria.fr/sparql-extension/example/table.html.

extension functions avoids to cast datatype values from RDF to the target language (e.g. Java) and back. It enables us to associate function definitions directly to a SPARQL query, with no need to compile nor link code. All standard SPARQL functions are natively available in LDScript and can be used directly in a LDScript function definition. LDScript extends the SPARQL filter expression language with several classical programming statements, among which FUNCTION, LET, FOR, FUNCALL and APPLY. The SELECT and CONSTRUCT SPARQL query forms are also statements of LDScript and can be used as well in the definition of functions. In addition, the language provides recursion, hence enabling recursive SPARQL queries: a function can execute a SPARQL query that can call the function. An LDScript interpreter is implemented in the Corese Semantic Web Factory as well as a compiler of LDScript into Java and the performances of the LDScript interpreter have been demonstrated on generic and concrete test cases. We successfully implemented several use cases with LDScript, among which a SHACL validator.

As future work, we will work on type checking function definition. LDScript indeed needs a type analysis because several new constructs have implicit type constraints: for example, the expression argument in `Map` must be list-valued; the first argument of `funcall` must be URI-valued. We also intend to extend higher order functions application to predefined functions. In the current version, higher order functions operate only on LDScript user-defined functions. On another note, we will work on the improvement of the performance of our implementation, and plan to provide a second implementation of LDScript on another triple store to validate the language. Finally, we plan to investigate the notion of "Linked Functions" and go further in the definition of a function programming language for SPARQL. The principle would consist in (1) dereferencing a function URI to get the function definition, provided security rules on function namespaces, and (2) annotating function URIs in the spirit of Linked Data.

References

1. Atzori, M.: Toward the Web of functions: interoperable higher-order functions in SPARQL. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8797, pp. 406–421. Springer, Cham (2014). doi:10.1007/978-3-319-11915-1_26
2. Corby, O., Faron-Zucker, C.: The KGRAM abstract machine for knowledge graph querying. In: IEEE/WIC/ACM International Conference on Web Intelligence, Toronto, Canada (2010)
3. Corby, O., Faron-Zucker, C., Gandon, F.: A generic RDF transformation software and its application to an online translation service for common languages of linked data. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 150–165. Springer, Cham (2015). doi:10.1007/978-3-319-25010-6_9
4. Corby, O., Faron-Zucker, C., Gandon, F.: LDScript: a Linked Data Script Language. Research report RR-8982, INRIA (2016)
5. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Vidal-Naquet, G., Wirsing, M. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987). doi:10.1007/BFb0039592

6. Knublauch, H.: SPIN - SPARQL Syntax. Member Submission, W3C (2011). http://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/
7. Lefrançois, M., Zimmermann, A., Bakerally, N.: A SPARQL extension for generating RDF from heterogeneous formats. In: Blomqvist, E., Maynard, D., Gangemi, A., Hoekstra, R., Hitzler, P., Hartig, O. (eds.) ESWC 2017. LNCS, vol. 10249, pp. 35–50. Springer, Cham (2017). doi:10.1007/978-3-319-58068-5_3
8. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960)
9. Williams, G.: Extensible SPARQL functions with embedded javascript. In: ESWC Workshop on Scripting for the Semantic Web, SFSW 2007, Innsbruck, Austria. CEUR Workshop Proceedings, vol. 248 (2007)