Toward the Web of Functions: Interoperable Higher-Order Functions in SPARQL

Maurizio Atzori

Math/CS Department
University of Cagliari
Via Ospedale 72
09124 Cagliari (CA), Italy
atzori@unica.it

Abstract. In this work we address the problem of using any third-party custom SPARQL function by only knowing its URI, allowing the computation to be executed on the remote endpoint that defines and implements such function. We present a standard-compliant solution that does not require changes to the current syntax or semantics of the language, based on the use of a `call` function. In contrast to the plain "Extensible Value Testing" described in the W3C Recommendations for the SPARQL Query Language, our approach is *interoperable*, that is, not dependent on the specific implementation of the endpoint being used for the query, relying instead on the implementation of the endpoint that declares and makes the function available, therefore reducing interoperability issues to one single case for which we provide an open source implementation. Further, the proposed solution for using custom functions within SPARQL queries is quite expressive, allowing for true higher-order functions, where functions can be assigned to variables and used as both inputs and outputs, enabling a generation of Web APIs for SPARQL that we call *Web of Functions*. The paper also shows different approaches on how our proposal can be applied to existing endpoints, including a SPARQL-to-SPARQL compiler that makes the use of `call` unnecessary, by exploiting non-normative sections in the Federated Query W3C Recommendations that are currently implemented on some popular SPARQL engines. We finally evaluate the effectiveness of our proposal reporting our experiments on two popular engines.

1 Introduction

During the years, the Web enlarged its initial scope of place where to publish static interlinked hypertexts. Standards such as HTTP enabled an incredible diversity of applications. Recently, Linked Data has shown the huge potential of having the Web as an unbounded, decentralized and free crowdsourced data store where everyone can access and contribute. Nowadays, structured data can be shared by a simple URI, in a similar way to HTML pages, where URIs can link each others by using the RDF model, forming a huge graph. Each data publisher provides a part of this graph, and through endpoints those subgraphs

can be effectively queried by means of a powerful standard (by W3C) query language, the SPARQL. When it was announced, Sir Tim Berners-Lee declared that "SPARQL will make a huge difference" making the Web machine-readable. More recently, as also detailed later in the paper, a number of researchers worked on extending the relations between SPARQL and the Web of Data, allowing for instance dynamic exploration of the linked data by dereferencing the URIs appearing in the query, and therefore not relegating SPARQL to be a language for local data only.

In our work, we further embrace this research line, but instead of dealing with the data, we focus on the computational power and expressivity of functions that can be used within a query. While extending the language with user-defined custom functions (sometimes called extension functions) represented by URIs is a native feature of the language, the mechanism only works on the single endpoint featuring that specific function. Instead, in this paper we envision a Web where also functions can be openly published, making them available to all the users of any other endpoint. We address the problem of practically realizing such *Web of Functions* with a Remote Procedure Call (RPC) approach, that is, users can call a function by only knowing its corresponding URI, as it is the case for the Web of Data, while the implementation and the computational resources are made available by the function publisher, as it happens with RPC and usual Web APIs. In other words, we present a first step toward a new generation of Web APIs available within any endpoint, to be used in SPARQL queries, and strictly coupled with the Web of Linked Data. Our contribution enables the Web of Functions with a surprisingly strong backward compatibility, since *(a)* it does not require any change to current SPARQL specifications, *(b)* it does not require a special implementation of the endpoint that declares and publish a custom function and *(c)* only requires the definition of a `call` function on the querying endpoint. Regarding the last point, we provide an open source implementation for the required function, along with two other approaches making no requirements on the querying endpoint, based on W3C SPARQL Federated Queries (FED). The approaches are proved to work on Apache Jena and on any other endpoint that implements the non-normative parts described in the FED Recommendations.

Organization. In Section 2, we review the related work. Section 3 introduces the problem and the desiderata through simple examples. Section 4 describes our solution based on a function to implement remote calls. In Section 5 we show how to deploy the remote call functionality on existing endpoints, including Section 5.3 where we show a pure SPARQL approach that under some hypothesis can realize the Web of Functions with backward compatibility, through a SPARQL-to-SPARQL compiler. In Section 6, we discuss notable aspects and limitations of the approach. In Section 7 we show our outcomes experimenting our prototype implementation on two popular engines and finally, in Section 8, we draw the conclusions and identify possible future work in this research area.

2 Related Work

In [1], Gregory Williams defines an approach to make the extension functions interoperable. The paper suggests to act on the engine implementation of the endpoint in order to allow each SPARQL engine to run code, specifically javascript code, downloaded from third-party servers at query time. This allows any kind of functions to be run on the SPARQL endpoint containing the data. Although effective, that approach requires modifications of the way SPARQL functions are published, with mandatory dereferenceability of functions, the development of a code interpreter on each endpoint, other than security and other performance issues already discussed in the original paper. Work in [1] is definitely an interesting research proposal and present a non-backward-compatible way of implementing the Web of Functions, with high costs in terms of SPARQL engine codebase modifications. In contrast, our approach already works off-the-shelf in existing SPARQL engines, as it is totally compliant with current SPARQL standards, without requiring any changes in the existing SPARQL endpoint codebase, relying on RPCs made possible by the Federated Queries part of the SPARQL 1.1 standard.

Another interesting and related line of research is the one of query execution through link traversal, in Olaf Hartig et al. [2–5], including SQUIN[1], a query interface for the Web of Linked Data based on these studies. In the same area of research is DIAMOND [6, 7]. In our opinion these papers show the growing trend of joining tools like endpoints and the SPARQL query language thought for local usage, with the larger Web of Linked Data. We believe our work can be considered in this research streamline, although focusing on functions instead of data.

In our paper we also deal with the expressivity of SPARQL. Existing work that focuses on its computational power and expressivity can be found in [8]. In [9] we propose a simple function that enhances the computational power of SPARQL by introducing recursive SPARQL functions.

In [10] a recent proposal to combine REST scalability with Linked Data expressivity is presented. The work is focused on the use of JSON-LD. While powerful and related, their approach seems not to consider SPARQL endpoints as a querying source. Our work is instead focused on envisioning a Web of Functions where functions are callable from any endpoint, while minimizing (or avoiding at all) technology changes to existing engine installations by leveraging the federated queries mechanism [11].

3 Problem Definition

In this Section we detail the problems in current custom functions that we are addressing in this paper. In our opinion, there are two main issues that limit the free use of "public" (that is, shareable, third-party) custom functions inside SPARQL queries: interoperability and expressivity.

[1] `http://squin.org/`

3.1 Limited Interoperability

The first problem is that user-defined functions are not interoperable. In fact, current official SPARQL extension function mechanism is based on the exploitation of unspecified behavior in case of function errors, a sort of trick on the semantics of the `PrimaryExpression` SPARQL 1.1 grammar rule, where expected error values are instead used as an extension point to implement custom functions in specific engines, including Virtuoso, Jena, Sesame and others. Specific implementations of SPARQL endpoints can therefore compute a function instead of throwing an error, allowing the execution of custom user-defined functions missing in the SPARQL specification. The specification correctly states that "SPARQL queries using extension functions are likely to have limited interoperability", as the custom function will only run on the endpoint implementing that specific function. The custom function seems to be confined to the endpoint that implements it.

In order to better show the problem, let us stick to the following running example and have some fun stealing names from cryptography. Suppose we are Alice, the owner of the domain `alice-server.org`, with semantic data exposed as linked open data and through an endpoint at `http://alice-server.org/sparql`. We want to query our data using a complex function that is not part of standard SPARQL. That function has been already implemented in Bob's endpoint, `http://bob-server.org/sparql`, and he called it `http://bob-server.org/fn/complexFunction`.

The desiderata for Alice would be to run on her own endpoint a query similar to the following:

```
PREFIX bob: <http://bob-server.org/fn/>
SELECT *
WHERE {
    # within Alice data, find useful values for ?arg1, ?arg2 and ?arg3
    ...
    # now use Bob's function
    FILTER(bob:complexFunction(?arg1, ?arg2, ?arg3) )
}
```

Note that this is perfectly compliant with the syntax of SPARQL, but nonetheless it is not going to work on Alice's endpoint because it does not recognize the semantics of `bob:complexFunction`. On the other end, if she runs the query on Bob's endpoint, the function will be working but Alice data would be inaccessible. A simple trick could be to use federated query on Bob's endpoint to get Alice data and then use `bob:complexFunction`. Unfortunately, this is not working in the general case where we also want to use functions from other function providers, such as `carol:otherFunction`. Therefore, custom functions such as `bob:complexFunction` and `carol:otherFunction` are totally acceptable from a syntactic point of view, but unfortunately they are not interoperable, i.e., difficult or impossible to be used by other endpoints such as Alice's. Another important thing is that, in our view, functions should be utilizable by just sharing the URI, not also the endpoint address. This is a principle in the Web of

Linked Data, where anyone can access and refer to data by just referring to its URI. Our desiderata requires the same principle, applied to functions. While the syntax of SPARQL already allows functions to be represented as URIs, they are not usable unless the endpoint that defines their semantics is also known and, even worse, currently the same URI may have different semantics on different endpoints.

3.2 Limited Expressivity

As far as we have seen, the syntax of SPARQL is ready for open functions. In fact the query shown above, that Alice would like to run, validates. Nevertheless, in terms of expressivity, the SPARQL language lacks of basic syntax to handle higher-order functions (HOF) effectively. First let us review the previous query example in terms of language expressivity. Function names are URIs, and therefore can be assigned to variables. Functions can be called passing variables, therefore the syntax allows functions to accept functions as input, and also to return functions. Further, although our examples always use functions as FILTER expressions for the sake of readability, it is well-known that SPARQL does not relegate functions to this usage, for instance we can use

```
BIND( bob:complexFunction(?arg1, ?arg2, ?arg3) AS ?result )
```

assigning the result of a function call to a variable. This may apparently and surprisingly put SPARQL in the class of languages that natively support higher-order functions. Unfortunately expressivity is strongly limited as we are going to show in the following example.

```
SELECT *
WHERE {
    # within Alice data, find useful values for ?arg1, ?arg2 and ?arg3
    ...
    # now Alice also binds ?f to a URI that represents a function
    # for instance, suppose ?f is bound to alice:fn
    BIND(alice:fn AS ?f)
    ...

    # now Alice wants to call the ?f function with arguments
    FILTER( ?f(?arg1, ?arg2, ?arg3) )
}
```

As before, in the last line we want to call a function with 3 arguments, but here the function is assigned to a variable, whose value is known only at runtime. This is not valid syntax in SPARQL 1.1 recommendation. This is not only a syntactic sugar restriction, it strongly limits the expressivity of function usage in SPARQL, since e.g., we cannot call a function which is the result of another function call. Here another related example of unfeasible query:

```
PREFIX alice: <http://alice-server.org/fn/>
SELECT *
WHERE {
    # within Alice data, find useful values for ?arg1, ?arg2 and ?arg3
```

```
    ...
    FILTER( alice:memoize(alice:fn) (?arg1, ?arg2, ?arg3) )
}
```

The function `alice:memoize`, that is supposed to memoize the input function `alice:fn` by caching the computed result, returns an (unnamed) function that should be called with the three arguments. Notice that in this example all the functions are local on purpose (i.e., not referring to Bob's endpoint) to better show that this is not a problem of interoperability.

In the following Section, we propose our simple solution to handle both limitations while sticking on the current syntax of SPARQL, i.e., without proposing unlikely future language extensions and changes.

4 A Function to Rule Them All

We found a simple and elegant solution to the problem of expressivity just described, that also strongly reduces the problem of interoperability to one single case. Let us show how the proposed approach can be applied in practice, and later we will detail how the remaining interoperability issue can be solved.

Given a query such as:

```
{
    ?arg a :UsefulThing # some interesting stuff
    FILTER( fn:thirdPartyFunction(?arg) )
}
```

involving a custom function, we propose to write the last line as:

```
    FILTER( wfn:call(fn:thirdPartyFunction , ?arg)  )
```

where the prefix `wfn` is a loosely short for "Web of Functions", defined as:

```
    wfn : <http://webofcode.org/wfn/>
```

The simple expedient of using `wfn:call` to call a function solves the expressivity problem of Section 3.2. In fact, now we can handle functions in variables:

```
    ?function a :UsefulFunction # match to a function
    FILTER( wfn:call( ?function , ?arg)  )
```

as well as nested calls. Please notice that since the expressivity problem of Section 3.2 originates from a syntax limitation in SPARQL, any solutions will have to deal with the syntax of the query. We believe that our proposal of using a "call" function can be considered, concerning the syntax, an acceptable solution to the problem. In fact, it is similar to already well-known functions (e.g., *apply* or *funcall* in other functional languages) and, more importantly, it does respect the syntax of SPARQL as per current W3C recommendations, therefore a viable solution even for existing endpoints.

If this proposed way of formulating a query involving function calls is used, then the only required custom function to be implemented on an endpoint is

`wfn:call`. In other words, its use solves the expressivity issue and reduces the interoperability issue to one special case, the function `wfn:call` itself.

In the following, we describe what we expect from the execution of the call function, in order to implement it. Then, in Section 5 we devise three different approaches to implement this function, including a pure SPARQL implementation, detailed in Section 5.3, that works under some assumptions.

4.1 Semantics of WFN:CALL

The function `wfn:call`, as we have seen, is a custom function that takes one argument, which is a constant or a variable containing an IRI representing a function, and then zero or more other arguments (either constants or variables). Whenever the function is called, it takes the first argument, the function URI, and call it passing all the remaining arguments. The semantics is therefore the same of *funcall* in the Lisp language.

Remember that given an IRI, there is no explicit semantics of what code should be run, since as we already mentioned, it is left to the specific endpoint implementation. In our approach, calling a function means to execute a SPARQL query over the endpoint that defines it. Here is the tricky part, as there is no way in SPARQL to know the correct endpoint that defines a custom function given a function IRI. We propose three possible answers to this problem:

Explicit Reference List. The endpoint has a user-defined list of *(function URI, endpoint URI)* pairs, therefore when a function should be called, the correct endpoint is found by matching that function URI. This approach requires a static configuration – while restrictive, this has the benefit of reducing the risks of running functions by only allowing known trusted endpoints.

Dereferenceable Functions. Whenever a function should be called, it is dereferenced. The entity should be of type `sd:Function` and also have a property `sd:endpoint` that points to the endpoint URL that implements the dereferenced function. Here we are reusing the official `sd` prefix defined in the SPARQL Service Description recommendation. Dereferecing functions is a feature already mentioned and proposed in literature (e.g., see [1]), but currently not described in the recommendation nor used on a widespread basis. A disadvantage of this approach is that it cannot be implemented using pure SPARQL since dynamic dereferencing (that is, the URI to be dereferenced is contained in a variable) is not allowed in SPARQL. Anyway, we suggest to make function URIs dereferenceable even when this approach is not used to implement `wfn:call`. We also notice that some RDF attributes available through function dereferenceability may also inform on the computational complexity and other function characteristics (number of arguments, returned type, etc.). In this paper we only focus on the problem of running remote custom functions without proposing a specific ontology for functions and algorithms.

Function-to-Endpoint IRI pattern. By sticking on a URI pattern, we may force that given a function URI, there is a deterministic way to find its endpoint. We propose to remove all the ending chars different from "`/`" and then append the string "sparql". That is, if the function IRI is `alice:myFunction` then the computed endpoint URI will be `alice:sparql`. To the best of our knowledge, this is an original proposal that solves the problem. It also allows similar functions (determined by the URI prefix) to be implemented by the same endpoint and viceversa; also, the computation is fast and can be implemented in pure SPARQL as we are going to show in Section 5.3.

As we have seen, all of these 3 strategies to finding the endpoint associated to a function URI have pros and cons. In the following, unless otherwise noted, we assume that the *Function-to-Endpoint IRI pattern* strategy is used.

By using our running example, if Bob wants to share a custom function, he can create an endpoint such as `http://bob-server.org/fn/sparql` where his custom functions, for instance `bob:complexFunction` and `bob:anotherCoolFn`, are implemented. So, now that we know which endpoint should be queried for a given IRI function, the `wfn:call` can query the correct endpoint and execute the custom function. It will proceed by following these protocol steps:

1. compute the endpoint URI by using one of the alternatives mentioned earlier;
2. make a SPARQL query to the remote endpoint using the given parameters as arguments for the remote custom function, binding the results to a variable;
3. get the returning value, and use it as a returning value for the call function; in case of error, the same error must be thrown by the call function, as if the function was run locally.

For nested calls, the inner calls should be computed first, as usual. This also means that every remote call will not contain any reference to the custom function `wfn:call` in the arguments since it should be already resolved.

5 Implementing and Executing the WFN:CALL Function

Once we reduced all the interoperability issues arising from the use of custom functions to one single case, it remains to solve the interoperability issue of the function `wfn:call`. In other words, we need to assure that the `wfn:call` function will be recognized and run by the querying endpoint.

We devise three approaches here to implement the `wfn:call` on an endpoint: (1) through the extensible testing function on the querying endpoint; (2) through a middleware third-party endpoint implementing the function acting as a proxy between the querying endpoint and the remote endpoint that implements the custom function; and (3) through a SPARQL-to-SPARQL compiler (a query rewriting tool).

In the following we detail all of these three solutions, enlightening both positive aspects and disadvantages.

5.1 Native Support on the Querying Endpoint

The simplest way to allow the use of `wfn:call` is to implement it in the SPARQL engine. This can be done by vendors or by the user, as a custom function, and this is exemplified in the upper path of Fig. 1. We implemented that function in Java[2] on an Apache Jena/Fuseki triplestore, as shown later in the Experiments section. Notice that such implementation should be implemented only on the querying endpoint. For instance, the following call:

```
VALUES (?arg1 ?arg2 ?arg3) {("1st" "2nd" "3rd") }
FILTER( wfn:call(bob:complexFunction, ?arg1, ?arg2, ?arg3) )
```

will be caught by the SPARQL processor (also known as ARQ in Apache Jena) and executed by a Java class[3] that extends `FunctionBase`. The Java implementation will use one of the strategies in the previous section to compute the endpoint's URI associated with the `bob:complexFunction` function. Then, it will run on Bob's endpoint a query such as the following[4]:

```
SELECT ?result
WHERE {
    BIND( <http://bob-server.org/fn/complexFunction>("1st,"2nd","3rd")
    AS ?result)
}
```

obtaining the result computed in the remote endpoint. Notice that at the time of running the Java code, arguments are bound and therefore their values can be used in the query.

This solution will work, with the only drawback of requiring a native implementation of `wfn:call` on the endpoint. The next approaches show that under some hypothesis it is not necessary to implement it on the querying endpoint.

5.2 Proxy-based Federated Query

If we cannot rely on a native implementation in our endpoint (e.g., we do not have privileges to register a custom function), we can still use the `wfn:call` implemented on a third-party commodity endpoint by using the SPARQL 1.1 Federated Query mechanism, under a reasonable hypothesis discussed later in this subsection.

[2] Available at `http://atzori.webofcode.org/projects/wfn/`

[3] See `https://jena.apache.org/documentation/javadoc/arq/com/hp/hpl/jena/sparql/function/FunctionBase.html` for details; in OpenLink Virtuoso it will require a C program, and current version forces the use of the prefix `bif` (built-in function)

[4] The actual query is slightly more complex in order to be run on those endpoints (such as Virtuoso) that do not support the use of `BIND` without a previous graph pattern matching; on some other endpoints (e.g., Apache Jena) this is a valid SPARQL query.

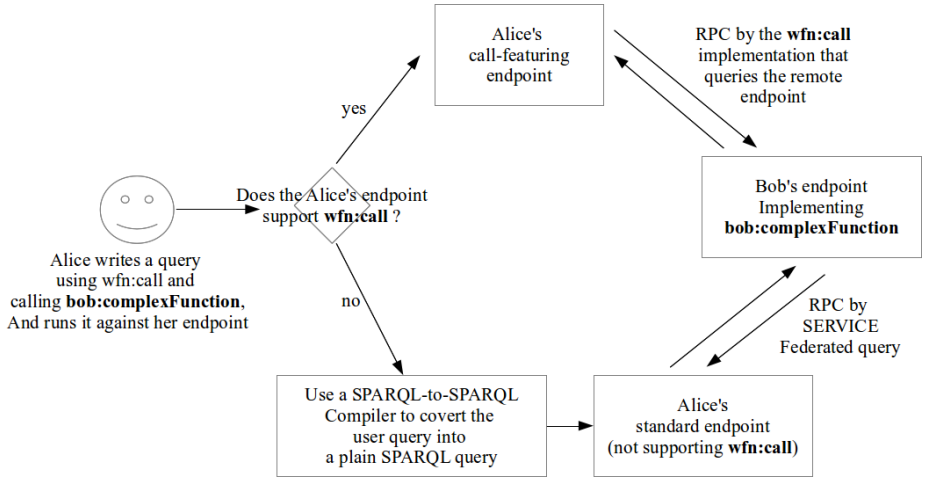

Fig. 1. The workflow of the user query in case *(a)* of an endpoint supporting `wfn:call` and *(b)* of a standard endpoint, not supporting the extension. In the second case, the tool described in our paper can convert the query into a standard one, that can be run on any endpoint.

This approach is based on rewriting each part of the query where the call function appears. For any occurrence of `wfn:call`, starting from inner nested calls and continuing to outer calls, the expression is substituted with a `SERVICE` call to the commodity call-featured server. For instance, let us take Alice's query (`PREFIX`es are removed for the sake of readability):

```
SELECT ?arg1
WHERE {
    # within Alice data, find useful values for ?arg1, ?arg2 and ?arg3
    ...
    # now want to use remote Bob's function
    FILTER( wfn:call(bob:complexFunction, ?arg1, ?arg2, ?arg3) )
}
```

The query will not be changed except for the last FILTER line, that will be replaced with the following lines, exploiting the federated query mechanism:

```
SERVICE wfn:commodityEndpoint {
    BIND ( bob:complexFunction(?arg1, ?arg2, ?arg3) AS ?tmp_result1 )
}
FILTER( ?tmp_result1 )
```

This simple expedient allows the part of the query using the call function to be computed against a commodity endpoint known to feature it.

There is a hidden hypothesis under which this federated query will work. The tricky part is that function arguments (`?arg1`, ...) are bound, but such

bindings are not guaranteed to be passed to the remote server according to normative W3C specifications. Despite this, a non-normative part of the Federated Query specifications (section 2.4 *Interplay of SERVICE and VALUES*[5]) suggests to pass such bound values to the remote server for optimization purposes. In fact, without passing any constraint to the remote server, most of the federated queries would be too long to be computed, making the Federated Query mechanism useless. We verified on two popular SPARQL engines, namely Apache Jena/Fuseki and OpenLink Virtuoso, that on their last versions they actually substitute variables with their values whenever they perform a `SERVICE` query, without even using the `VALUES` keyword. Therefore, assuming that a performance optimization strategy has been implemented (with or without `VALUES`) and used by the engine, variable values (bindings) are actually passed to the remote commodity server, that can use them to finally compute the `wfn:call` function[6].

The exploitation of the performance optimization already implemented in a number of existing endpoints allows therefore the use of higher-order custom functions through a proxy endpoint, without having to implement or register `wfn:call` on the querying endpoint. We consider such exploitation for a completely different problem (implementing remote calls on SPARQL) an original contribution of this paper.

5.3 A Pure SPARQL Approach based on SPARQL-to-SPARQL Query Compiling

In both previous approaches we had to implement `wfn:call` (in the querying endpoint or in the proxy endpoint). Here we are going to show how SPARQL can be turned into a fully HOF-featuring language without relying on the implementation of a call function nor introducing changes to the current SPARQL syntax, We found existing SPARQL specification mechanisms that can be used to implement it as a query rewriting tool. This way we can construct a SPARQL-to-SPARQL compiler able to rewrite a SPARQL query that uses `wfn:call` into another that does not, therefore usable on current endpoints not implementing the call function.

Firstly, we show that through the *Function-to-Endpoint IRI pattern* approach described above we can dynamically compute the endpoint address given a function IRI, by using only SPARQL, which is not possible in general by dereferencing, and very limited by using, instead, a predetermined list of known IRI. Secondly, we generate a pure SPARQL query usable within a SPARQL query from any endpoint by exploiting the federated query mechanism. Thus, we propose a standard-compliant technique to allow easily shareable third-party functions to be used within SPARQL queries from any existing endpoint, assuming the implementation of two non-normative parts in the Federated Query specifications. We stress that, unlike other proposals, this is possible without changing SPARQL

[5] see `http://www.w3.org/TR/sparql11-federated-query/#values`

[6] in case of `VALUES`, the proxy can read the bindings and rewrite the query to be sent to the final endpoint, without using the `VALUES` keyword

specifications nor any codebase of existing engine implementations, since the compiled query is a standard SPARQL 1.1 query where `wfn:call` does not appear, and other custom functions are called remotely against the appropriate endpoint. Of course, the assumption that non-normative parts are implemented does not always hold, and therefore approaches in previous sections are still interesting. Anyway, we verified that on the popular Apache Jena/Fuseki engine the two non-normative parts required for this approach actually hold, and therefore this pure-SPARQL approach may have a practical impact on some existing installations.

The SPARQL-to-SPARQL compiler works as described next. As in the previous proxy-based approach, any occurrence of `wfn:call` will be replaced with a `SERVICE` call. For Alice's query:

```
SELECT ?arg1
WHERE {
    # within Alice data, find useful values for ?arg1, ?arg2 and ?arg3
    ...
    # now want to use remote Bob's function
    FILTER( wfn:call(bob:complexFunction, ?arg1, ?arg2, ?arg3) )
}
```

the compiler will not change the query except for the last `FILTER` line, this time without using any proxy endpoint, relying on the Bob's endpoint directly:

```
SERVICE bob:sparql {
    BIND ( bob:complexFunction(?arg1, ?arg2, ?arg3) AS ?tmp_result1 )
}
FILTER( ?tmp_result1 )
```

Now let us show the case where the function URI is not known at compile time. The query that Alice wants to run may contain a variable `?f` bound to a function:

```
# now Alice wants to call the ?f function with arguments
FILTER( wfn:call(?f, ?arg1, ?arg2, ?arg3) )
```

To solve this, we propose the use of another non-normative Section of the W3C Federated Query Specification (Section 4 titled *SERVICE Variables*[7]), that allows to use `SERVICE` against dynamically computed endpoints. Thus, in this case the compiler will convert the previous `FILTER` line into the following:

```
# dynamically compute the endpoint URI
BIND( URI(REPLACE( STR(?f) ,"/[^/]*$" , "/sparql" ) )
        AS ?tmp_endpoint1)
```

[7] See `http://www.w3.org/TR/sparql11-federated-query/#variableService`. We verified that at least on Apache Jena this non-normative part has been implemented. Note that the query is valid SPARQL, since the syntax for SERVICE VAR is normative, but some vendors may require user configuration or have a different semantics for SERVICE variables.

```
SERVICE ?tmp_endpoint1 {
    BIND ( wfn:call(?f, ?arg1, ?arg2, ?arg3) AS ?tmp_result1 )
}

FILTER( ?tmp_result1 )
```

Here we are featuring a query that has to deal with the expressivity syntactic issue of Section 3.2. Therefore, this time `wfn:call` must appear in the query to maintain syntactic compatibility with SPARQL standard. Anyway, it should be noted that it is performed on the remote server publishing the custom function, that we may assume it is a call-featuring engine. Also notice that the implementation of the `wfn:call` in Bob's engine is straightforward (a simple query rewriting), since at the time of execution the variable `?f` is always instantiated by Alice's endpoint, and it is assumed to point to a Bob's function.

This last example also helps us to show the implications of the different alternatives in the previous section, namely Explicit Reference List, Dereferenceable Functions, and Function-to-Endpoint IRI pattern. Having an Explicit Reference List with a known URI can be used to convert, within the query, a function URI into a SPARQL endpoint URI, after importing the list by using, e.g., the FROM clause. Although working, in this approach it is not clear who should be in charge of maintaining such a centralized list. We believe it is in contrast with the decentralized architecture of the Web of Linked Data. Dereferenceability of function URIs is a generally desirable behaviour, but unfortunately the SPARQL language does not seem to support dereferencing a URI contained in a variable[8], therefore a pure SPARQL compiler would not be possible. Instead, as we have shown, the last approach based on Function-to-Endpoint URI pattern can work in a SPARQL-to-SPARQL tool without drawbacks except for the acceptable constraint on the relation between function's and endpoint's URIs.

6 Discussion on the Proposed Approach

In the following, we discuss some notable aspects arising when using custom functions through the "function-as-a-service" RPC approach we have presented.

Decoupling Functions and Data. Since the basic problem addressed in this paper is about having data and functions in different servers, we should note that decoupling them takes consequences on the possible interactions. For instance, the result of a function that computes the palindrome of a given string only depends on the input string. Such palindrome function has no interaction problem. We notice that almost all the standard SPARQL functions behave like this. On the contrary, a function computing the length of a minimum path between two IRIs

[8] Legacy, non standard solutions may exist on specific engines; for instance, the Virtuoso engine supports the `input:grab-var` pragma to dereference the content of a variable.

also needs the whole graph to get the result, not only the two given IRIs. The same limitation occurs when dealing with blank nodes, which are not sharable across SPARQL endpoints by definition. A consequence is that functions handling RDF collections (e.g., `rdf:List` of *W3C RDF Schema 1.1*) cannot be computed with the call function, since only a reference to the first element is transferred (not the whole collection). Although in a true Linked Open Data this is still feasible, because by dereferencing the IRIs the server implementing such a function can fetch the whole graph, this may lead to reduced performance. Also notice that dereferenceability is not mandatory for this approach: the server implementing the function can run a SPARQL query against the first endpoint to get the other data needed to answer[9].

Performance. Being outsourced, performance on the computation of custom functions depends on other servers. The approach also has an intrinsic performance limitation, requiring the input data to be passed to the external server, and then getting the answer back. Our experiments show that performances are acceptable in many cases, whenever the number of function calls (determined by the number of possible arguments matching locally) is not too large.

The system used to implement the federated query protocol can also contribute to improve the performances. For instance, function calls can be compacted to a single query with the `VALUES` keyword used to specify multiple set of function arguments, reducing delays due to network latency.

True HOF. Our HOF approach allows functions to be used as input and output. In fact, there is no technical constraint that avoids custom functions to be created at runtime. For instance, we implemented a function `compose` that given two other functions returns a dynamically-generated function URI that, whenever used, returns the composition of the two given functions [12].

Security. Our approach based on computing functions on someone else's server, which is declaring the function and providing the implementation, drastically reduces Denial of Services and more serious security threats feasible in other approaches [1], where on the contrary the calling endpoint engine does run unknown code dynamically downloaded from another server. Since we are using only existing standards, most of the security issues known and related to already accepted mechanisms, such as federated queries. Anyway, we remark that enhancing expressivity necessarily also leads to enhanced capabilities for attackers, allowing for instance new kinds of Denial of Services (DoSs) attacks derived by the composition of functions. We suggest an implementation with user-configurable timeouts, that will kill any external function call after a determined period of time. While this paper is not focused on security issues related to querying untrusted federated endpoints, this should mitigate a number of possible attacks.

[9] Although this may introduce interoperability problems in case the data is not shown by the endpoint.

Privacy. Whenever we use a remote custom function, we are providing arguments to a remote (potentially untrusted) endpoint server. Likewise SPARQL Federated Queries, privacy issues must be taken into account before sharing sensitive data with untrusted servers.

7 Experimental Evaluation

In order to verify the effectiveness of our approaches, we implemented `wfn:call` on the Apache Jena/Fuseki endpoint, using the *Extensible Value Testing* mechanism of SPARQL, available at `http://atzori.webofcode.org/projects/wfn/`.

7.1 Time Performance of `wfn:call`

Running a function through a Remote Procedure Call mechanism introduces network delays that must be evaluated. We computed the concatenation of two strings on 3 different settings: (a) locally, i.e., without the use of `wfn:call`, (b) remotely, against a Jena server, and (c) remotely, against a Virtuoso server. Results are shown in Table 1. Differences are negligible considering that times are in milliseconds. We also performed the experiment on a slower network connection, obtaining no sensible differences, requiring approx $5s$ for every setting. Therefore, we conclude that `wfn:call` is effective in terms of performance, executing queries in the same order of magnitude (x3.1) even in the worst-case setting.

Table 1. Time performance in different query settings

Setting	Time (ms)	Overload (times)
local execution of fn:concat	46	x1 (ref)
execution of fn:concat on a remote Jena server	71	x1.5
execution of fn:concat on a remote Virtuoso server	144	x3.1

7.2 Experiments on the Current Feasibility of a SPARQL-to-SPARQL Compiler

According to Lee Feigenbaum, Co-Chair of the W3C SPARQL Working Group, "SPARQL 1.1 defines a mechanism to communicate results from one endpoint to another, but this is not currently widely deployed". We ran a number of queries against the latest versions[10] of Apache Jena/Fuseki and and OpenLink Virtuoso to verify their implementation of the Federated Query protocol and non-normative parts. We found that both engines optimize federated queries sending bindings to the remote endpoint, although they use substitution instead of the suggested `VALUES` clause. For each engine, we found a federated SPARQL query that can take the place of the `wfn:call` function. Further, only Apache Jena implements `SERVICE VAR` where variables can contain references to remote endpoints. More information available at the above web site.

[10] Jena v. 2.11.1, Fuseki v. 1.0.1, Virtuoso Opensource Single Server Edition v. 07.10.3209 – all running on a Linux box

8 Conclusions and Future Work

We have shown how the current specification of the SPARQL language allows the use of higher-order custom functions, which are also interoperable. This is possible through the use of a specific function, or by query rewriting techniques developed in the paper without introducing mandatory language extensions on some engines. We believe this may be the first step toward a novel view of the Web as a place holding code and functions, not only data as the Linked Data is greatly doing. The Semantic Web already shifted URIs from pages to conceptual entities, primarily structured data. We believe that among these concepts there should be computable functions. In other words, Semantic Web and its SPARQL language should include web services as first class resources, not only static Web of Data, enabling what could be called the *Web of Functions*. As a future work we plan to explore this research direction by also including the *property functions* (sometimes called magic properties) and possible integrations with user-friendly SPARQL interfaces such as SWiPE [13].

Acknowledgments. This work was supported in part by the RAS project CRP-17615 *DENIS* and by MIUR PRIN 2010-11 project *Security Horizons*.

References

1. Williams, G.: Extensible SPARQL Functions with Embedded Javascript. In: ESWC07 Workshop on Scripting for the Semantic Web, SFSW07. (2007)
2. Hartig, O., Bizer, C., Freytag, J.C.: Executing SPARQL Queries over the Web of Linked Data. In: International Semantic Web Conference. (2009) 293–309
3. Hartig, O.: An Introduction to SPARQL and Queries over Linked Data. In: Web Engineering - 12th International Conference, ICWE12. (2012) 506–507
4. Hartig, O.: SPARQL for a Web of Linked Data: Semantics and Computability. In: 9th Extended Semantic Web Conference, ESWC12. (2012) 8–23
5. Hartig, O.: SQUIN: a traversal based query execution system for the web of linked data. In: ACM SIGMOD Conference. (2013) 1081–1084
6. Arenas, M., Gutierrez, C., Miranker, D.P., Pérez, J., Sequeda, J.: Querying Semantic Data on the Web? SIGMOD Record **41**(4) (2012) 6–17
7. Miranker, D.P., Depena, R.K., Jung, H., Sequeda, J.F., Reyna, C.: Diamond: A SPARQL Query Engine, for Linked Data Based on the Rete Match. In: Artificial Intelligence meets the Web of Data Workshop, co-located at ECAI 2012. (2012)
8. Angles, R., Gutierrez, C.: The Expressive Power of SPARQL. In: International Semantic Web Conference. (2008) 114–129
9. Atzori, M.: Computing Recursive SPARQL Queries. In: 8th IEEE International Conference on Semantic Computing, ICSC. (2014) 258–259
10. Lanthaler, M.: Creating 3rd generation web APIs with hydra. In: 22nd International World Wide Web Conference, WWW (Companion Volume). (2013) 35–38
11. Aranda, C.B., Arenas, M., Corcho, Ó., Polleres, A.: Federating queries in SPARQL 1.1: Syntax, semantics and evaluation. J. Web Sem. **18**(1) (2013) 1–17
12. Atzori, M.: call: A Nucleus for a Web of Open Functions. (submitted for publication)
13. Atzori, M., Zaniolo, C.: SWiPE: Searching Wikipedia by Example. In: 21st World Wide Web Conference, WWW (Companion Volume). (2012) 309–312