Functional-Logic Programming for Web Knowledge Representation, Sharing and Querying

Matthias Nickles

Insight Centre for Data Analytics and Department of Information Technology
National University of Ireland, Galway
matthias.nickles@deri.org

Abstract. We propose a unified approach to semantically rich knowledge representation, querying and exchange for the Web, based on functional-logic programming. JavaScript- and JSON-based so-called information scripts serve as a unified knowledge representation and query format, with logical reasoning being a constraint solving or narrowing task. This way, our framework provides a highly versatile, easy to use and radically different alternative compared to conventional forms of knowledge representation and exchange for the Web.

Keywords: Knowledge representation, JSON-LD, JSON, JavaScript, Functional-Logic Programming, Relational Programming, Linked Data, Semantic Web.

1 Introduction and Related Work

With this position paper, we propose a functional-logic (FL) approach to expressive knowledge representation, querying and exchange for the Web which is directly based on a popular Web scripting language (JavaScript/JSON). Doing so, we aim to address serious issues with existing Semantic Web (SW) and Linked Data (LD) technologies: while RDF(S) has a foothold in the area of Linked Open Data (LOD), standard semantic technologies (including RDF and OWL) are arguably lacking acceptance in other large communities of potential SW/LD users, in particular those of Web software developers and many non-institutional knowledge providers. Instead, the use of "non-semantic" data formats like JSON, CSV, XML and relational database formats is prevalent in these areas. Furthermore, RDF is severely restricted in terms of logical expressiveness and reasoning capabilities (in particular in connection with SPARQL).

As a response to these issues, we propose so-called *information scripts* as a knowledge format which is at the same time more versatile and likely much easier to use (for non-logicians) than traditional SW or LD technologies. Information scripts are directly encoded using plain JavaScript and/or JSON (JavaScript Object Notation) - languages which are already familiar to many users. First-order features and querying are both based on existentially quantified (query) variables (related to but going beyond RDF's blank nodes), and deduction as well as query processing can be realized in a unified and expressive way as a straightforward constraint solving task. Main advantage over RDF besides the likely better comprehensibility to many people is the ability to formalize "real" logical formulas (including quantified variables, rules and negation).

K. Janowicz et al. (Eds.): EKAW 2014, LNAI 8876, pp. 333–338, 2014.

One flavor of our approach (Sect. 2.2) can be seen on the syntax level as an extension of JSON-LD [1] and allows for the inclusion of links in the form of URIs, but there is no requirement to make any link to RDF(S).

The closest related older approach is the Relational-Functional Markup Language RFML [3]. However, whereas RFML is a new FL language with XML syntax, we propose the direct use of an existing off-the-shelf language (namely JS/JSON) for representation, logical inference and queries. Other related works include approaches to "semantic programming" [5]. But in contrast to these, we do not aim for an integration of existing SW languages into general purpose programming languages. The idea to use query expressions for logic programming can also be found in [4], and the de facto unification of queries and logical rules can already be found in Datalog [2]. The concept of using boolean functions for logic programming is long known in the area of FL programming (e.g., [6]). MiniKanren [7] shares with us the goal to integrate logic programming directly into a host programming language, and can be implemented using general purpose languages including JS. However, our approach is syntactically even more lightweight and focuses on being a knowledge representation framework, compatible with JSON, while miniKanren focuses on being a programming language.

2 Information Scripts

We propose two concrete flavors of the idea outlined so far: 1) the direct use of a defined subset of JavaScript (JS) as a knowledge representation language, and 2) a variant of 1) which uses JSON (or JSON-LD) as underlying serialization format. Both variants (which can be combined) are called *information scripts*. Variant 1) uses syntactically plain JS scripts which can be embedded in Web pages and transferred in exactly the same way as any other parts of Web pages (e.g., microformats or embedded metadata). Only if we want to process queries over information scripts, we need additional functionality, however, this functionality could be provided as JS code also.

JS programs are ideal for this purpose since they are already meant to be transferred between hosts via the Internet, they can even be serialized to JSON if needed (if functions are encoded as source code), and modern browsers have sophisticated security mechanisms in place to ensure that programs do not harm client systems.

2.1 JavaScript as a Logic Language

The basic idea of FL programming is straightforward: logical rules and facts are represented by functions with a boolean result. Predicates become names of functions from their respective domain to type boolean. Logical connectives (including negation) are represented by boolean operators. Parameters of boolean functions correspond to universally quantified logical variables. *Query expressions* are used to obtain instantiations of existentially quantified logical variables (EQVs) (sometimes called *query variables* or just *logical variables*) for which boolean expressions with these variables evaluate to true. EQVs are represented using ordinary JS variables. Various sound evaluation mechanisms for FL programming languages exist, such as various forms of *narrowing* [6] and constraint solving. A way to approach evaluation technically would be by

adding basic constraint solving capabilities in the form of a JS library or some external off-the-shelf constraint solver.

The syntax of information scripts is simple: informally, an *information script* is a non-empty set of JS function definitions of the form `function pred(params) {...}`, where `pred` is a predicate name. Each function needs to be referentially transparent, must not have any side effects, and must return a value of type boolean (variants might relax some of these restrictions, e.g., allow arbitrary result types). The list of parameters (which can be of any type) can be empty. We also need two predefined functions which both invoke the constraint solver (or some other kind of reasoner):

`solveBind(qe[,v][,domains])` accepts a query expression `qe` in the form of an anonymous boolean function which in turn accepts a number of query variables (EQVs). `solveBind` returns an array of JS values which are those instances of the EQVs for which the call of `qe` returns boolean value true. Optionally, the set of EQVs whose instances shall be returned can be restricted to some subset (parameter `v`). Large or infinite domains could be handled by variants of `solveBind` which return their results incrementally in a lazy fashion (e.g., as a data stream). Inference using `solveBind(qe)` is undecidable, but if decidability is really required, information scripts would have to be restricted to some decidable fragment. Optionally, domains for the EQVs can be provided.

`exists(qe[,domains])` is similar to `solveBind(qe)`, but it just returns false or true - the latter if there are any instances of the EQVs for which `qe` is satisfied.

An information script is therefore encoded using a subset of plain JS (we use the ECMAScript 6 standard syntax in the examples below, just to be able to encode anonymous functions syntactically nicely as Lambda abstractions). The syntax might look unusual at first for a logic language, but even though this issue could easily be fixed using a simple syntax preprocessor, a feature of this approach is precisely that the functional nature of rules is not concealed - logical rules are actually also (boolean) functions, and their "function nature" should allow programmers who are not familiar with logic programming or SW technologies to understand the meaning of an information script immediately, once the concept of query variables has been introduced. Example:

```
<script type="information">
function person(x) {
   return x == "ann" || x == "bertrand" || x == "charles" ||
          x == "dottie" || x == "evelyn" || x == "fred" ||
          x == "george" || x == "bill"; }
function parent(x,y) {
   return x=="dottie" && y =="george" || x == "evelyn" &&
          y == "george" || x == "bertrand" && y == "dottie" ||
          x == "ann" && y == "dottie" || x == "anne" && y == "bill"
          || x == "charles" && y == "evelyn"; }
function sameGenCousins(x,y) {
    return (x == y && person(x) ||
           exists(($x1, $y1) => (parent(x, $x1) && parent(y, $y1)
                                 && sameGenCousins($x1, $y1)) )); }
</script>
```

Functions `person(x)` and `parent(x,y)` are rules (in the FL programming sense) although they actually define sets of ground facts such as `person("ann")`. Note that

there is no requirement for the bodies of these functions to use boolean connectives; just as well we could obtain within the bodies of `person(x)` and `parent(x,y)` the person names and parent-relationships from some database or remote server using appropriate JS code. `sameGenCousins` is a recursive rule which states that two persons are same generation cousins whenever they are identical or if they have parents which are in turn same generation cousins. In classical logic, we would write this rule as $\forall x, y : x = y \vee \exists x_1, y_1 : parent(x, x_1) \wedge parent(y, y_1) \wedge sameGenCousins(x_1, y_1) \rightarrow sameGenCousins(x, y)$. Function `exists` calls the constraint solver and returns true *iff* for *some* instances of EQV `$x1` and EQV `$y1` condition `parent(x,$x1) && parent(y,$y1) && sameGenCousins($x1,$y1)` is satisfied. As an example for a query using the knowledge encoded in the script above, we call `solveBind(($x, $y) => sameGenCousins($x, $y))`, resulting in an array of tuples of same generation cousins
`[["ann","ann"], ... , ["dottie","evelyn"], ["ann","charles"]]`.
Observe that our approach so far does not enforce the use of any ontology or schema, or a namespace. There are at least two ways to add a schema or ontology if required: 1) switch from JS to a typed functional language (e.g., TypeScript or PureScript, which both compile to plain JS). This allows to attach type information to function parameters and thus simple schema functionality. 2) Model ontological constraints using the information script itself: e.g., `function person(x) {return user(x);}` asserts that every "user" is also a "person" (*is-a* relation of two concepts).

2.2 Non-ground JSON for Knowledge Representation and Queries

With the approach described above, ground knowledge is represented using JS terms such as `parent(dottie,charles)`. While this is compatible with data representation formats using ground terms as in logic programming, a more compact format using JSON as serialization format might be more handy in the context of the Web. In the following, we therefore propose a variant of the above which employs JSON (or a JSON application such as JSON-LD or Apache Avro) as representation format. We still require `solveBind` (Sect. 2.1), but query expressions, non-ground facts and rules (again in the form of boolean functions) can now optionally be embedded directly into a JSON document, which thus might contain logical variables (in the form of EQVs).

Concretely, we lift JSON to the first-order level by allowing certain JavaScript expressions and logic variables as property (key) values. We call such "higher-order JSON documents" *non-ground JSON* (NG-JSON). NG-JSON documents are valid JSON documents. They can contain expressions and logical variables in string form (e.g., `"age": "$age >= 18"`) and their semantics is operationally defined by a grounding process which maps them to their instances, i.e., a (possibly empty) set of ground JSON files where non-ground expressions have been replaced with their respective results (this does not imply that such an extension must be actually performed). The denotation of a NG-JSON document is not ground if values of logical variables are functions themselves (in JS, functions are first-class citizens), but we ignore this possibility here for lack of space. Note that the extensional form does not need to be finite (infinite groundings, if required, could be handled again by using lazy data streams). The NG-JSON document, which might be significantly more compact than its set of groundings, can

either be directly transmitted to the client or knowledge consumer, or it could be expanded to ground data on server-side (similarly to *stored procedures* in DBMS). As an example for NG-JSON, consider the following document which specifies properties of persons (some omitted for lack of space):

```
[ {  "name": "John Smith",
     "@id": "http://johnsmith.com",
     "age": 23,
     "parents": [
         { "name": "Alice Springs" },
         { "name": "Tom Smith" } ],
     "adultSiblings": "$x.age > 17 && isSibling(this, $x)"
  },
  {  "name": "Mary Hippler",
     "age": 25,
     "parents": [
         { "name": "Alice Springs" },
         { "name": "Tom Smith" } ]
  },
  {  "name": "William Smith",
     "age": 16,
     "parents": [
         { "name": "Alice Springs" },
         { "name": "Tom Smith" } ]
  } ]
```

Like any NG-JSON, this example is a valid JSON file (with optional JSON-LD elements which are handy in a Web context, like `@id` for URIs). Only non-ground property is John's property "adultSiblings", which represents all objects (instances of EQV `$x`) which fulfill constraint `isSibling(this, $x)`. In our example, the only valid instance of `$x` is the person with name Mary Hippler.
`$x.age > 17 && isSibling(this, $x)` is just syntactic sugar for
`($x) => $x.age > 17 && isSibling(this, $x).this` is a JS keyword which here refers to person John (as a JS object). The definition of rule `isSibling` is omitted (`isSibling` is a boolean function which simply checks if two persons are different and have the same parents. It could be provided either as a function-type property or in the way described in Sect. 2.1). An even more compact way to write this example could avoid the repetition of the values of `parents` using a JS expression.

Grounding is a context-sensitive operation - we need to know the domains of EQVs. In the example, the domain of `$x` is the set of all top-level objects in the NG-JSON document, but this is *not* necessarily so. We could allow for, e.g., numerical ranges or databases as domains, and provide a way to specify a context which is shared among different documents (a solution might involve JSON-LD's `@context`).

The same NG-JSON document can be seen both as a *data generator* (which "generates" all valid expansions of non-ground slots) and as a query (which instantiates the EQVs with values from the context, e.g., from some given database or objects in the NG-JSON document itself).

To obtain ground document(s), there are two possibilities: we can expand a non-ground expression to an array obtained from a `solveBind` call (see Sect. 2.1) (the result for the example would look like the NG-JSON document, but with

`"adultSiblings": [{"name":"Mary Hippler", "age": 25, ...}]` instead of the current non-ground key/value pair `"adultSiblings": "$x..."`).

Alternatively, we can create a set of ground JSON documents where each ground document represents one particular logical variable assignment. E.g., if the non-ground value of John's property "age" would be `"$age >= 21 && $age <= 23"`, grounding this way would generate *three* ground JSON documents with three different persons with name John Smith with three different ages 21, 22 and 23. Properties which trigger the former kind of expansion should be appropriately labeled in the NG-JSON document (this is omitted in the example code above).

Technically, the grounding of a NG-JSON document is a two-step process (details omitted for lack of space): firstly, we parse (de-serialize) the document as normal into an array of objects (here: persons), with the exception that each property value which is a string containing a lambda expression is converted into the respective anonymous function (this can be done by providing JSON.parse() with a suitable key (property) handler). Secondly, the array of objects is traversed and for each property p (here: `adultSiblings`) which contains an anonymous function, the anonymous function is passed as an argument to function `solveBind`. The result of `solveBind` is then used to ground property p. Some care is required to pass on the proper execution object context for keyword `this` with each `solveBind` and `isSibling` call.

3 Conclusions

With this paper, we have proposed a new, script-based approach to formal knowledge representation, querying and sharing. While this work certainly leaves room for refinements (e.g., how to express properties of properties?) and does not strive for coverage of all technical details, it is hoped that it provides a contribution towards semantic technologies which are more suitable for many Web-related tasks and Web-related software development than most existing Semantic Web approaches. Future work includes technical refinements and additions (such as formal specifications of query semantics and contexts), and an experimental evaluation.

References

1. `http://www.w3.org/TR/json-ld/`
2. Gallaire, H., Minker, J. (eds.): Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse (1977); Advances in Data Base Theory (1978)
3. Boley, H.: Markup Languages for Functional-Logic Programming. Procs. 9th WFLP 2000 (2000)
4. Polleres, A., Wallner, J.: On the Relation Between SPARQL1.1 and Answer Set Programming. Journal of Applied Non-Classical Logics (JANCL) 23(1-2), 159–212 (2013)
5. Oren, E., Heitmann, B., Decker, S.: ActiveRDF: Embedding Semantic Web data into object-oriented languages. In: Web Semantics: Science, Services and Agents on the World Wide Web (2008)
6. Hanus, M., Lucas, S.: An Evaluation Semantics for Narrowing-Based Functional Logic Languages. Journal of Functional and Logic Programming 2001(2) (2001)
7. Byrd, W.: Relational Programming in miniKanren: Techniques, Applications, and Implementations. PhD dissertation (2009)