Using Blocks to Get More Blocks: Exploring Linked Data through Integration of Queries and Result Sets in Block Programming

Paolo Bottoni
Department of Computer Science
Sapienza, University of Rome
Rome, Italy
Email: bottoni@di.uniroma1.it

Miguel Ceriani
Department of Computer Science
Sapienza, University of Rome
Rome, Italy
Email: ceriani@di.uniroma1.it

Abstract—**While many Linked Data sources are available, there is a lack of effective non-expert user interfaces to build structured queries on them. The block programming paradigm promotes a gradual and modular approach. If—in an integrated user interface—both queries and results are represented as blocks, modularity can be effectively used to support an exploratory way of designing Linked Data queries.**

I. Introduction

The Linked Data [1]—the structured data available online—are increasing both in quantity and diversity [2]. A key advantage of the Linked Data model is its support for serendipitous exploration and reuse of existing data. In practice, though, exploring and querying Linked Data is not trivial and usually requires knowledge of a structured textual query language like SPARQL [3], the standard query language for Linked Data. Existing experimental tools for non-experts (see for example [4], [5]) while being effective for some cases, do not support the user very much in the incremental process of designing queries, because reusing intermediate queries and results for new queries is not easy.

In this paper, we discuss the use of the block programming paradigm to design queries on Linked Data sources. We specifically address the need for exploratory queries, through the integration of queries and their results in a uniform user interface (in the rest of the paper, UI). The discussion will be based on a concrete UI we recently proposed [6] to build queries on Linked Data.

II. Background

The Resource Description Framework (RDF) [7] is the data model proposed by W3C for Linked Data. In this model knowledge is represented via *RDF statements* about *resources*—where a resource can be anything in the "universe of discourse". An RDF statement is represented by an *RDF triple*, composed of *subject* (a resource), *predicate* (specified by a resource as well), and *object* (a resource or a literal, i.e. a value from a basic type). An *RDF graph* is a set of RDF triples. Resources are uniquely identified by an internationalized resource identifier (IRI) [8]. The resources used to specify predicates are called *properties*. *Prefixes* can be used in place of the initial part of an IRI, which represents specific namespaces for vocabularies or sets of resources. For example, the IRI namespace for standard RDF concepts[1] is often abbreviated as `rdf:`, as in `rdf:type`–the property used to associate a resource with its type(s).

III. A User Interface Based on SPARQL

The tool is based on Blockly, an open source library for block programming [9]. The UI (see Fig. 1) mostly sticks to the standard Blockly UI, consisting of a *workspace* for building block programs and a menu on the left—called *toolbox*—to pick the blocks from. Clicking on an item of the toolbox—a *category*—produces the visualization of a particular subset of blocks from which one can be dragged to the workspace. Following common practice, the blocks having the same role in the language also have the same color and belong to the same category in the toolbox. In order to maximize consistency with other block programming environments, the color and organization of blocks has been aligned to the standard Blockly blocks. The following subsections describe the different types of blocks provided, together with the language structure and the motivations behind the main choices.

A. Graph Queries as Blocks

The language mimics the structure of SPARQL, but will be described without explicit references to SPARQL syntax. In Fig. 2 a block representing a *select* query is shown. It corresponds to the natural language question "Give me the two highest Greek mountains." The *where* sub-block (the one to the right of the word "where") is a *graph pattern* —a block or set of blocks used to match parts of the given RDF graph—that "looks for" resources (the variable `mount` in the example) of a certain type (the class `dbo:Mountain`), connected through directed properties (`dbo:elevation`) to other resources or values (the variable `elevation`). Generally speaking, inside graph patterns different types of sub-blocks can be used:

[1] `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

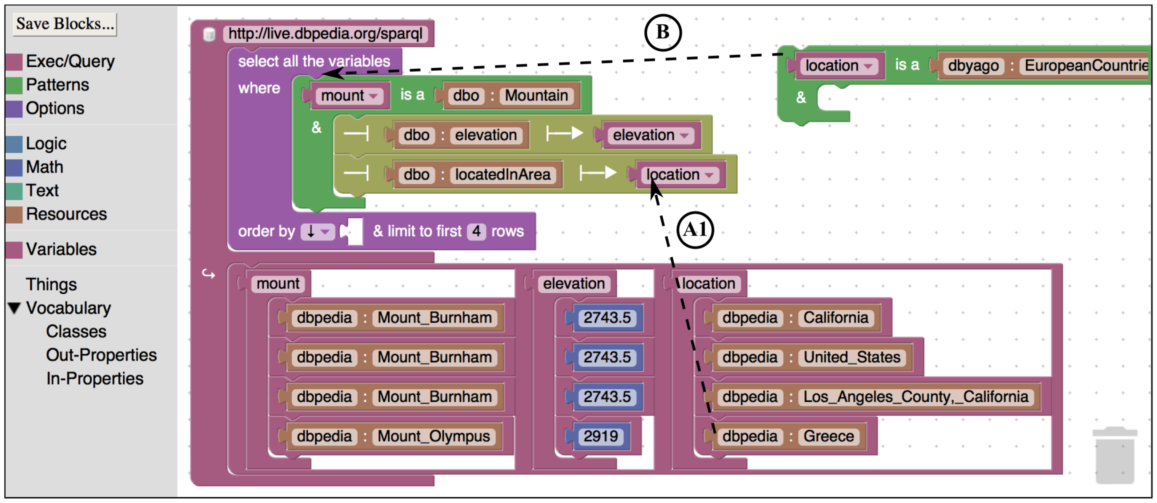

Fig. 1. User Interface after the execution of a query to get four mountains with their locations and heights

resources, *literals*, and *variables*. While resources and literals are constants that must match the same term in the given RDF graph, variables are the placeholders of the patterns that get bound to the terms of the given RDF graph for which the pattern matches. Graph patterns can be combined either by simply joining them together—in that case all must match at same time—, by using the *optional* block—the patterns under the optional may or may not be matched—, or by using the *union* block—at least one of branches of the block must match. The output of a select query is a list of tuples, each corresponding to a valid set of bindings of the used variables.

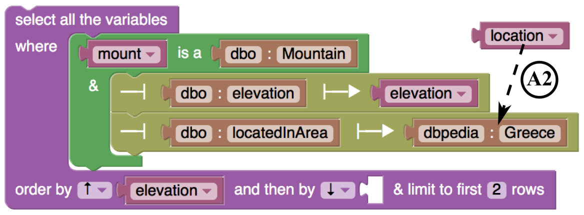

Fig. 2. Query to get the two highest Greek mountains

The blocks are visualized and organized (in the toolbox) according to their role: the query block is provided under the category *Exec/Query*, along the execution block that will be described in III-B; the different blocks used to build graph patterns are grouped under the category *Patterns*; the optional block and the union block are under the category *Options*; variable blocks—available in the category *Variables*—are managed and visualized in the standard way provided by Blockly; literal blocks have a basic type that is associated with one of the standard basic types used in Blockly, *boolean*, *numeric*, and *string* and are grouped under the categories *Logic*, *Math*, and *Text*; resource blocks can be found under the category *Resources*.

B. Execution and Results as Blocks

A specific *execution block* is designed for query execution: as soon as a query is attached to this block the SPARQL query is generated and run on the underlying dataset. Fig. 3 shows an execution block connected to a query and "waiting for the results". When the results are ready (see for example Fig. 1), they appear in tabular format attached to the lower connection of the execution block. The single data items are represented as resource blocks and literal blocks that can be dragged from the result set to create other queries (or even modify the one that generated them).

The existence of two distinct blocks for query and execution allows the user to distinguish the design of the query from its execution if he or she wants so: by building the query separately, the user gets the results only when the query is connected to an execution block. Conversely, by modifying a query already connected to an execution block, the query is executed immediately and the results shown as soon as they are available[2].

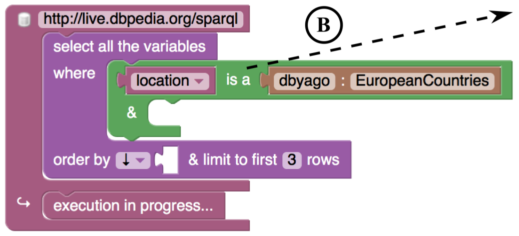

Fig. 3. Execution of a query to get three European countries

IV. Affordances

To argument in favor of the proposed paradigm, we will show some of the affordances—related to the process of designing a query—of such UI.

[2]Query execution is in any case non-blocking, i.e. the UI is reactive and operational even if one or more queries are being executed.

A. Generalization/Specialization

The design of a query may proceed from a generic version, with a minimal number of constraints, designed to start getting some data. Then, by adding constraints the query will gradually become more selective. This process, that we may call *query specialization*, is supported by reducing the free variables (through replacement with constant values or with already used variables), by adding *filter* blocks (see the “provided that” blocks in Fig. 4), by adding more graph patterns to be fulfilled, and by moving graph patterns out from an *optional* or *union* block The usage of blocks for the query output allow the user to get easily the specific blocks corresponding to resources or literals needed to replace variables or to create filter expressions. An example of *query specialization* is the transformation from the query in Fig. 1—that asks for some mountains, their locations and their heights—to the query in Fig. 2—where only Greek mountains are selected–by dragging the resource `dbpedia:Greece` from the results in the place of the variable `location` (see arrow A1 in Fig. 1).

The design of a query may also start from a specific query on known data and then proceed by lifting some constraints to include a greater set of results. We may call this process *query generalization* and it is supported by replacing constant values with variables, by removing filter blocks, by removing graph patterns (or part of), and by moving graph patterns under an *optional* or *union* block. These actions corresponds directly and naturally in our environment to the removal of blocks or the creation of new ones for the variables. An example of generalization can be the reversal of the specialization example, i.e. from the query in Figure 2 to the one in Fig. 1, by dragging on the workspace a new variable block to replace the resource `dbpedia:Greece` (see arrow A2 in Figure 2).

B. Composition/Decomposition

A complex query may be created by composing different queries together, so that, for example, the output of separate components of the query can be checked before composing them. As the proposed UI permits multiple queries to be built and executed in the same workspace, composition is directly executed by joining blocks from different queries[3]. For example, to build a query for European mountains the user may first design a query to get mountains and their locations (Fig. 1) and another query to get the European countries (Fig. 3). The complete query can be composed by dragging the graph pattern of the latter query to add it inside the former one (see arrows B in both figures).

The reverse operation is to decompose a query in smaller ones. From the point of view of the UI the actions are similar to the ones needed for composition: dragging blocks and creating some new ones. Reversing the example of composition gives an example of decomposition.

C. Stepwise Querying

Sometimes a query design comes after some exploratory steps which identify some relevant resources, classes or properties. This approach is supported through permitting values dragged out of the result set of a query to be used by other queries. The old query may then be removed or simply kept aside for further use. For example, the query in Fig. 4 may be used to get some candidate classes to represent the set of mountains, by searching among available classes using their label. As soon as the resource `dbo:Mountain` is identified as the relevant class, the corresponding resource block may be dragged away (see arrow C in Fig. 4) to be used for follow up queries, like for example the one in Fig. 1 (see the “is a” part of the graph pattern).

[3]Blocks may also be duplicated to preserve the original queries.

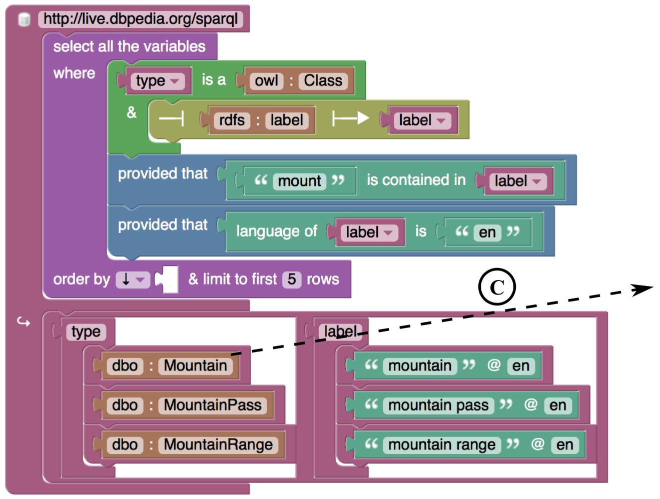

Fig. 4. Execution of a query to get classes with a label containing “mount”

V. Conclusions and Future Work

We propose a new paradigm for querying Linked Data, based on a novel take on block programming: using blocks not only to program but also to show results, which can be incorporated in incremental design of queries. The proposed incremental approach and UI could be possibly extended to a wider set of query languages and data models. We plan to start an extensive user evaluation of usability, both for Linked Data and in a more general setting.

References

[1] T. Berners-Lee, “Linked data,” Design Issues, 2006.
[2] C. Bizer, T. Heath, and T. Berners-Lee, “Linked data-the story so far,” *Int. J. on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, 2009.
[3] S. Harris *et al.*, “SPARQL 1.1 Query Language,” W3C REC 21 March 2013.
[4] S. Ferré, “Sparklis: a SPARQL Endpoint Explorer for Expressive Question Answering,” in *ISWC 2014 Posters & Demonstrations Track*.
[5] A. Russell, P. R. Smart, D. Braines, and N. R. Shadbolt, “Nitelight: A graphical tool for semantic query construction,” 2008.
[6] P. Bottoni and M. Ceriani, “Linked Data Queries as Jigsaw Puzzles: a Visual Interface for SPARQL Based on Blockly Library,” in *Proc. of CHItaly 2015*. ACM, 2015, p. [To Appear].
[7] R. Cyganiak, D. Wood, and M. Lanthaler, “RDF 1.1 Concepts and Abstract Syntax,” W3C REC 25 February 2014.
[8] M. Duerst and M. Suignard, “Internationalized Resource Identifiers (IRIs),” RFC 3987 (Proposed Standard), Internet Engineering Task Force, Jan. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc3987.txt
[9] N. Fraser *et al.*, “Blockly: A visual programming editor,” 2013.