Semantic Faceted Search: Safe and Expressive Navigation in RDF Graphs

Sébastien Ferré, Alice Hermann, Mireille Ducassé

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1964 – Janvier 2011

Semantic Faceted Search: Safe and Expressive Navigation in RDF Graphs

Sébastien Ferré[*] , Alice Hermann[**] , Mireille Ducassé[***]
{*Prenom.Nom*}*@irisa.fr*

Abstract: Faceted search and querying are the two main paradigms to search the Semantic Web. Querying languages, such as SPARQL, offer expressive means for searching knowledge bases, but they are difficult to use. Query assistants help users to write well-formed queries, but they do not prevent empty results. Faceted search supports exploratory search, i.e., guided navigation that returns rich feedbacks to users, and prevents them to make navigation steps that lead to empty results (dead-ends). However, faceted search systems do not offer the same expressiveness as query languages. We introduce *semantic faceted search*, the combination of an expressive query language and faceted search to reconcile the two paradigms. The query language is basically SPARQL, but with a syntax that extends Turtle with disjunction and negation, and that better fits in a faceted search interface: LISQL. We formalize the navigation of faceted search as a navigation graph, where nodes are queries, and navigation links are query transformations. We prove that this navigation graph is *safe* (no dead-end), and *complete* (every query that is not a dead-end can be reached by navigation). That formalization itself is a contribution to faceted search. A prototype, Camelis 2, has been implemented, and a usability evaluation with graduate students demonstrated that semantic faceted search retains the ease-of-use of faceted search, and enables most users to build complex queries with little training.

Key-words: semantic web, faceted search, query language, exploratory search, navigation, expressiveness

---

Recherche à facettes sémantique : une navigation sûre et expressive dans les graphes RDF

Résumé : *La recherche à facettes et l'interrogation sont les principaux paradigmes de recherche d'information dans le Web sémantique. Les langages de requêtes, tels que SPARQL, offre une grande expressivité pour l'interrogation de bases de connaissances, mais ils sont difficiles d'utilisation. Les assistants de requêtes aident les utilisateurs à écrire des requêtes bien formées, mais ils n'empêchent pas les résultats vides. La recherche à facettes permet une recherches exploratoire, c'est-à-dire une navigation guidée offrant aux utilisateurs des résultats riches et leur permettant d'éviter des résultats vides (impasses). Cependant, les systèmes de recherche à facettes n'offrent pas la même expressivité que les langages de requêtes. Nous introduisons la* recherche à facettes sémantique*, la combinaison d'un langage de requêtes expressif et de la recherche à facette afin de réconcilier les deux paradigmes. Le langage de requête est basé sur SPARQL, mais avec une syntaxe qui étend la notation Turtle à la disjonction et à la négation, et qui convient mieux à une interface de recherche à facettes: LISQL. Nous formalisons la navigation de la recherche à facettes par un graphe de navigation, où les noeuds sont des requêtes et les liens de navigation sont des transformations de requêtes. Nous prouvons que ce graphe de navigation est* sûr *(pas d'impasse) et* complet *(toute requête qui n'est pas une impasse peut être atteinte par navigation). Cette formalisation elle-même constitue une contribution à la recherche à facettes. Un prototype, Camelis 2, a été implémenté et une étude d'utilisabilité avec des étudiants de master a permit de démontrer que la recherche à facettes sémantique conserve la facilité d'utilisation de la recherche à facettes et permet à la plupart des utilisateurs de construire des requêtes complexes avec peu de pratique.*

Mots clés : *web sémantique, recherche à facettes, langage de requêtes, recherche exploratoire, navigation, expressivité*

---

[*] Équipe LIS, Université de Rennes 1
[**] Équipe LIS, INSA Rennes
[***] Équipe LIS, INSA Rennes

1 Introduction

With the growing amount of available resources in the Semantic Web (SW), it is a key issue to provide an easy and effective access to them, not only to specialists, but also to casual users. The challenge is not only to allow users to retrieve particular resources (e.g., flights), but to support them in the exploration of a domain knowledge (e.g., which are the destinations? Which are the most frequent flights? With which companies and at which price?). We call the first mode *retrieval search*, and, following Marchionini [Mar06], the second mode *exploratory search*. Exploratory search is often associated to *faceted search* [HEE+02, ST09], but it is also at the core of Logical Information Systems [FR04], and Dynamic Taxonomies [Sac00]. Exploratory search allows users to find information without *a priori* knowledge about either the data or its schema. Faceted search suggests restriction values along various facets, i.e., dimensions of data, and only those that are relevant to the current selection. This has the advantages to remove the need to write queries, and to prevent dead-end queries, i.e., queries with no answer. Therefore, faceted search is *easy* and *safe*: *easy* because users only have to choose among the suggested restriction values, and *safe* because, whatever the choice made by users, the resulting selection is not empty. Logical information systems and dynamic taxonomies have a different presentation, but they work along the same principles. The selections that can be reached by navigation correspond to queries that are generally limited to conjunctions of restriction values, possibly with restricted negation and disjunction. This is far from the expressiveness of query languages for the semantic web, such as SPARQL[1]. SlashFacet [HvOH06] and BrowseRDF [ODD06] are faceted search systems for RDF data that extend the expressiveness of reachable queries, but still to a small fragment of SPARQL. For instance, both of them allow for neither cycles in graph patterns, nor unions of graph patterns (disjunction).

Querying languages for the semantic web, such as SPARQL [AG08], OWL-QL [FHH04], or SPARQL-DL [SP07], are quite expressive but are difficult to use, even for specialists. They do not return enough feedback to offer exploratory search, and nothing prevents users to write a query that has no answer. Indeed, even if users have a perfect knowledge of the syntax and semantics of the query language, they may be ignorant about the data schema, i.e., the *ontology*. If they also master the ontology or if they use a query assistant (e.g., Protégé[2]) or an auto-completion system (e.g., Ginseng [BKK05]), the query will be syntactically correct and semantically consistent w.r.t. the ontology but it can still produce no answer (e.g., it makes sense to ask for a flight from Rennes to Washington, but it happens that there is none).

The contribution of this paper is to extend faceted search to the semantic web, so as to offer an exploratory search that is (1) easy to use, (2) safe, and (3) expressive. Ease-of-use and safeness are retained from existing faceted search systems by keeping their general principles, as well as the visual aspect of their interface. Expressiveness is obtained by representing the current selection by a *query* rather than by a set of items, and by representing navigation links by *query transformations* rather than by set operations (e.g., intersection). We first formally define the *navigation graph* that results from the query language and *safe* query transformations, and we then prove that every query that is not a dead-end is reachable starting from the most general query (*navigation completeness*). In this way, the expressiveness of faceted search is determined by the expressiveness of the query language, rather than by the combinatorics of user interface controls. In this paper, the query language is based on SPARQL graph patterns, but with a syntax that extends Turtle with disjunction and negation, and that better fits in a faceted search interface: LISQL. To satisfy navigation completeness, a number of query transformations have to be introduced, which have no counterpart with set operations and are not all easy to define on SPARQL graph patterns: e.g., switching between the nodes of a graph pattern (*focus change*), forming cycles with variables, and introducing generalized disjunction or negation.

The use of queries for representing selections in faceted search has other benefits than navigation expressiveness. The current query is an intensional description of the current selection that complements its extensional description (listing of items). It informs users in a precise and concise way about their exact position in the navigation graph. It can easily be copied and pasted, stored and retrieved later. Finally, it allows expert users to modify the query by hand at any stage of the navigation process, without loosing the ability to proceed by navigation.

The paper is organized as follows. Section 2 gives preliminaries about the Semantic Web and faceted search. Section 3 is an informal presentation of *semantic faceted search*, based on our prototype implementation Camelis 2. The LISQL syntax is motivated and formally defined in Section 4, and query transformations are defined in Section 5. The theoretical core of semantic faceted search is given in Section 6 with the definition of the navigation graph, and the proof that it is safe and complete. This section also provides some groundings for the user interface that is presented in Section 3, as well as a discussion about complexity issues. Section 7 reports about a user study that demonstrates the usability of our approach. Our approach is also compared in Section 8 to other work in faceted search and query languages for the Semantic Web. Section 9 concludes this paper.

[1] see `http://www.w3.org/TR/rdf-sparql-query/`
[2] See `http://protege.stanford.edu/`

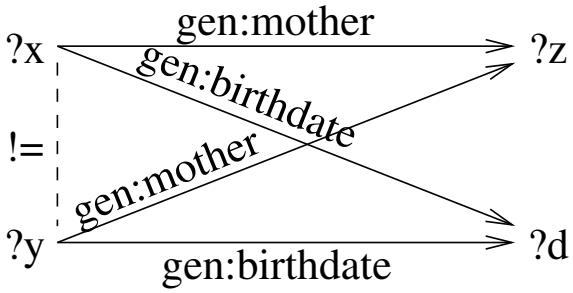

Figure 1: A graphical representation of a graph pattern.

2 Preliminaries

2.1 Semantic Web

The Semantic Web (SW) is founded on several representation languages, such as RDF, RDFS, and OWL, which provide increasing inference capabilities [HKR09]. The two basic units of these languages are *resources* and *triples*. A resource can be either a URI (Uniform Resource Identifier), a literal (e.g., a string, a number, a date), or a *blank node*, i.e., an anonymous resource. A URI is the absolute name of a *resource*, i.e., an entity, and plays the same role as a URL w.r.t. web pages. Like URLs, a URI can be a long and cumbersome string (e.g., `http://www.w3.org/1999/02/22rdfsyntaxns#type`), so that it is often denoted by a qualified name (e.g., `rdf:type`). We assume pairwise disjoint infinite sets of URIs (U), blank nodes (B), and literals (L). The set of *resources* is then defined as $R = U \cup B \cup L$.

A triple (s, p, o) is made of 3 resources, and can be read as a simple sentence, where s is the subject, p is the verb (called the predicate), and o is the object. For instance, the triple `(ex:Bob,rdf:type,ex:man)` says that "Bob has type man", or simply "Bob is a man". Here, the resource `ex:man` is used as a class, and `rdf:type` is used as a property, i.e., a binary relation. The triple `(ex:Bob,ex:friend,ex:Alice)` says that "Bob has friend Alice", where `ex:friend` is another property. The triple `(ex:man,rdfs:subClassOf,ex:person)` says that "man is subsumed by person", or simply "every man is a person". The set of all triples of a knowledge base forms a RDF graph.

Definition 1 (RDF graph) *An* RDF graph *is defined as a set of* triples $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$*, where* s *is the* subject*,* p *is the* predicate*, and* o *is the* object*.*

A *vocabulary* is a set of resources having a meaning defined by convention. RDF(S) is a vocabulary used to represent the membership to a class (`rdf:type`), subsumption between classes (`rdfs:subClassOf`), subsumption between properties (`rdfs:subPropertyOf`), the domain (`rdfs:domain`) and range (`rdfs:range`) of properties, the meta-class of classes (`rdfs:Class`), the meta-class of properties (`rdf:Property`), etc. OWL introduces additional vocabulary to represent complex classes and properties: e.g., restrictions on properties, intersection of classes, inverse properties. The variant OWL-DL is the counterpart of Description Logics (DL) [BCM+03], where resources are individuals, classes are concepts, and properties are roles. A RDF graph that uses the OWL vocabulary to define classes and properties is generally called an *ontology*. Each vocabulary comes with a semantics, and the richer the vocabulary is, the more expressive and the more complex inference is.

Vocabulary for genealogy. For illustration purposes, we consider RDF graphs about genealogical data. To this purpose, we introduce a custom vocabulary for genealogy. The URIs of this domain are associated to a namespace (`gen:`). This prefix is omitted if there is no ambiguity. Resources can be *persons*, *events*, *places* or literals such as names or dates. Persons belong either to the class of *men* or to the class of *women*, may have a *firstname*, a *lastname*, a *sex*, a *father*, a *mother*, a *spouse*, a *birth*, and a *death*. A birth or a death is an event that may have a *date* and a *place*. Places can be described as *parts* of larger places. The classes of *men* and *women* are declared as subclasses of the class of *persons*. The properties *father* and *mother* are declared as subproperties of the property *parent*.

Query languages provide on semantic web knowledge bases the same service as SQL on relational databases. They generally assume that implicit triples have been inferred and added to the base. The most well-known query language, SPARQL, reuses the `SELECT FROM WHERE` shape of SQL queries, using graph patterns in the WHERE clause. For instance,

twin siblings can be retrieved by the following query:

```
SELECT DISTINCT ?x ?y FROM <mygen.rdf>
WHERE { ?x gen:mother ?z. ?x gen:birthdate ?d.
        ?y gen:mother ?z. ?y gen:birthdate ?d
        FILTER ?x != ?y }
```

Figure 1 shows a graphical representation of the graph pattern of this query, where arrows represent triples oriented from the subject to the object. The query reads "two persons `?x` and `?y` are twins if they share a same mother and a same birthdate, and are different". The FILTER condition is necessary because nothing prevents two variables to bind to a same resource. In the following, we use the compositional syntax and semantics of graph patterns, as introduced by Angles and Gutierrez [AG08]. It is proved equivalent to the standard semantics, and its compositionality makes it easier to manipulate. The atomic graph patterns are triple patterns, and the composing binary operations (with their respective relation algebra operation) are AND (join), UNION (set union), MINUS (set difference), and OPT (left join). The operations AND and UNION are associative and commutative. Another operation, FILTER, combines a graph pattern and a filter constraint, which is a Boolean combination of atomic filter constraint. The only atomic filter constraint we need in this paper is equality, either between 2 variables (`?x = ?y`), or between a variable and a resource (`?x = r`). As an example, the graph pattern of the above query can be rewritten as `((?x,gen:mother,?z) AND (?x,gen:birthdate,?d) AND (?y,gen:mother,?z) AND (?y,gen:birthdate,?d)) FILTER` $\neg$`(?x = ?y)`.

Additionally, we introduce the *empty* graph pattern, denoted 1, which acts as a neutral element for operation AND; and $R(?x)$ as a shorthand for the triple pattern `(`$?x$`,rdf:type,rdfs:Resource)`, forcing the variable $?x$ to bind to a resource of the dataset.

2.2 Faceted Search

Faceted search [HEE+02, ST09] covers a family of user interfaces for browsing a collection of items. It is becoming a *de facto* standard in e-commerce websites, and its scope of application is wide (see Chap. 9 in [ST09]). It is suitable for *retrieval search*, i.e., the quick retrieval of an item already known to the user. It is also suitable for *exploratory search* [Mar06], i.e., the discovery of the objects that best suits the needs of the user, who has no prior knowledge of the item collection. An example of the later is when users want to buy a new camera. They do not know which models exist and what their features are, but they have constraints and preferences such as low cost, high resolution, or brand. Faceted search systems guide users through the item collection, and give them the feeling to have considered all the possibilities. At each navigation step, users only have to make a choice among a set of alternatives that are suggested by the system.

The data model underlying faceted search is simple. Each item is described along a set of *facets*, or dimensions. Each facet has a range of values. Therefore, each item is described by a set of pairs (facet,value), which we call *features*. A facet is not necessarily defined on all items. At any navigation step, the focus is defined as a set of items. The initial focus is generally the whole item collection. From the current focus, a set of *restriction values* are computed and displayed to the user. A restriction value is a feature that matches at least one item of the current focus. Each restriction value is generally accompanied by the number of items it matches. Restriction values are organized by facets, and for the sake of conciseness, most facets are initially collapsed, and expanded on demand. On the one hand, restriction values provide a summary of the current focus. On the other hand, each restriction value is a selector for a subset of the focus. The summary plays a crucial role in exploratory search because for each facet it shows only and all of the relevant values for the current focus. This allows for the informed choice of a restriction value: e.g., the lowest price or the highest resolution that is available given previous selected restriction values. The selection of a restriction value entails a change of the focus, which becomes a subset of the previous focus, and a new set of restriction values to reflect the new focus. The list of all selected restriction values is generally displayed, and any of them can be removed by users, leading to a larger focus. This is useful to relax a constraint, for example in order to get more choices. The list of all selected restriction values can be seen as a *query*, which in general is limited to a conjunction of restriction values, while restricted forms of negation and disjunction are sometimes available.

Dynamic Taxonomies (DT) [Sac00, Sac06, ST09] are a brand of faceted search, where a multidimensional taxonomy is used instead of facets and values. In fact, facets and values form a two-level taxonomy, with facets at the first level, and values at the second level. Using taxonomies of arbitrary depth allows for features at different granularity levels. For instance, a facet of date can be used at the levels of days, weeks, months, years, etc. Weeks and months can be combined because taxonomies need not be trees but can be directed acyclic graphs. Features are also called *concepts*, and the generalization ordering between features is called *subsumption*. Taxonomies are multidimensional, in that several features, even under a same facet, can be attached to a same item. This is useful with a facet of topics as a same item can match several topics. The term "dynamic taxonomy" stands for the fact that the summary is now a subset of the taxonomy, which dynamically adapts to the focus.

Logical Information Systems (LIS) [FR00, FR04] are another brand of faceted search that has been developed in our team since 1999, on the basis of Formal Concept Analysis [GW99] and logic-based information retrieval [vR86]. For what concerns us here, logical information systems can be defined as an extension of dynamic taxonomies, where features are the formulas of an ad-hoc logic, and subsumption is defined by logical inference rather than explicitly [FR07]. Using logics enhances the expressiveness of features and queries, as well as the design and engineering of complex taxonomies (see Chapter 8 in [ST09]). In LIS, the focus is defined as the set of answers, called *extension*, of the query, and changes of the focus are done through changes of the query. In addition to navigation, LIS provide direct querying for expert users, and query-by-examples to find items similar to a given set of examples.

3 Semantic Faceted Search: an Informal Presentation

This section gives an informal presentation of semantic faceted search, before formally defining it and proving its properties in further sections. This presentation is based on our prototype implementation, Camelis 2[3], which is an important evolution of Camelis [Fer09], a LIS implementation. Camelis 2 can import RDF/XML files without the need for data preparation or configuration. It gives access to all information in a RDF graph, and even takes into account some of the semantics of RDF(S): the hierarchy of classes (`rdfs:subClassOf`), the hierarchy of properties (`rdfs:subPropertyOf`), and the inheritance of types (`rdf:type`). Camelis 2 gives access to all kinds of entities, including literals such as dates, reified triples, and meta-entities such as classes, properties, and even meta-classes such as `rdfs:Class`.

To illustrate our presentation, we use the genealogical vocabulary introduced in Section 2.1, and we use genealogical datasets converted from GED files[4]. In particular, we use a dataset about the ascendancy of George Washington, which we also used in our user evaluation, reported in Section 7. This dataset has about 400 resources, including 79 persons, and about 4000 triples.

3.1 User Interface

The user interface of Camelis 2 includes the components of both faceted search systems, and querying engines, and introduces a few additional controls, e.g., for introducing variables, disjunction, and negation. Figure 2 shows a screenshot of Camelis 2. From top to bottom, and from left to right, it is composed of a menu bar (M), a toolbar (T), a query box (Q), query controls (QC), feature controls (FC), an answer list or extension box (E), a facet hierarchy (F), and a set of value boxes (V). A query engine can be derived from Camelis 2 by retaining only the components Q and E. A standard faceted search system can be derived by retaining only the components E, F, and V. Camelis 2 can effectively be used under those restrictions, but with a degraded behaviour: difficult and unsafe search in the first case, incomplete (and hence inexpressive) navigation in the second case.

The query box (Q) contains the current LISQL query that determines the current selection, also called the extension (E). LISQL is explained informally in the following, through examples (see Section 4 for formal definitions). In the figure, the query can be translated literally as "something that is a person and whose sex is M (male) and whose lastname is Washington". The words '`a`' and '`and`' are keywords, while the words '`person`', '`sex`', '`M`', '`lastname`', and '`Washington`' are abbreviated URIs, implicitly using the namespace `gen:`. 'M' and 'Washington' could have been represented as string literals. The underline indicates which entity in the query should be used for answers, here the person. The extension box (E) presents answers page by page.

The facet hierarchy (F) is a list of the classes and properties that match some answers, i.e., that some answers are subject of. Classes are here considered as unvalued facets, which are still selective because a facet need not apply to all resources. For example, in Figure 2, we see that all 17 answers are persons (class facet `a person`), and have a firstname (property facet `firstname : ?`). 11 answers are the husband of some family (property facet `husband of`), and hence married with somebody (property facet `married with ?`). The keyword '`of`' forms the inverse of a property, and '`with`' forms the symmetric closure of a property. Facets can be expanded, revealing a facet hierarchy. The expansion of a class facet uncovers subclasses according to the subclass relationship. For example, the expansion of `a person` would uncover the facets `a man` and `a woman`. The expansion of a property facet is more complex. First, it uncovers subproperties according to the subproperty relationship, like for classes. For example, the expansion of `opt trans parent : ?` ("has an ancestor") would uncover `parent : ?`, which would uncover `father : ?` and `mother : ?`. The keyword '`trans`' (resp. '`opt`') forms the transitive (resp. reflexive) closure of a property. Second, the expansion of a property facet uncovers the facets of its values: both classes and properties. For example, the expansion of `father : ?` would uncover the facets `father : a person`, and `father : birth : ?`. In the interface, the ellipsis ___ is used in place of repeated complex properties. As another example, the expansion of `birth : ?` uncovers the facet `birth : place : ?`, because some or all birth events have a place. The expansion of `birth : place : ?` further reveals the facet `birth : place : in ?`, where '`in`' is a reflexive

[3] downloadable at `http://www.irisa.fr/LIS/ferre/camelis/camelis2.html`

[4] `http://jay.askren.net/Projects/SemWeb/`

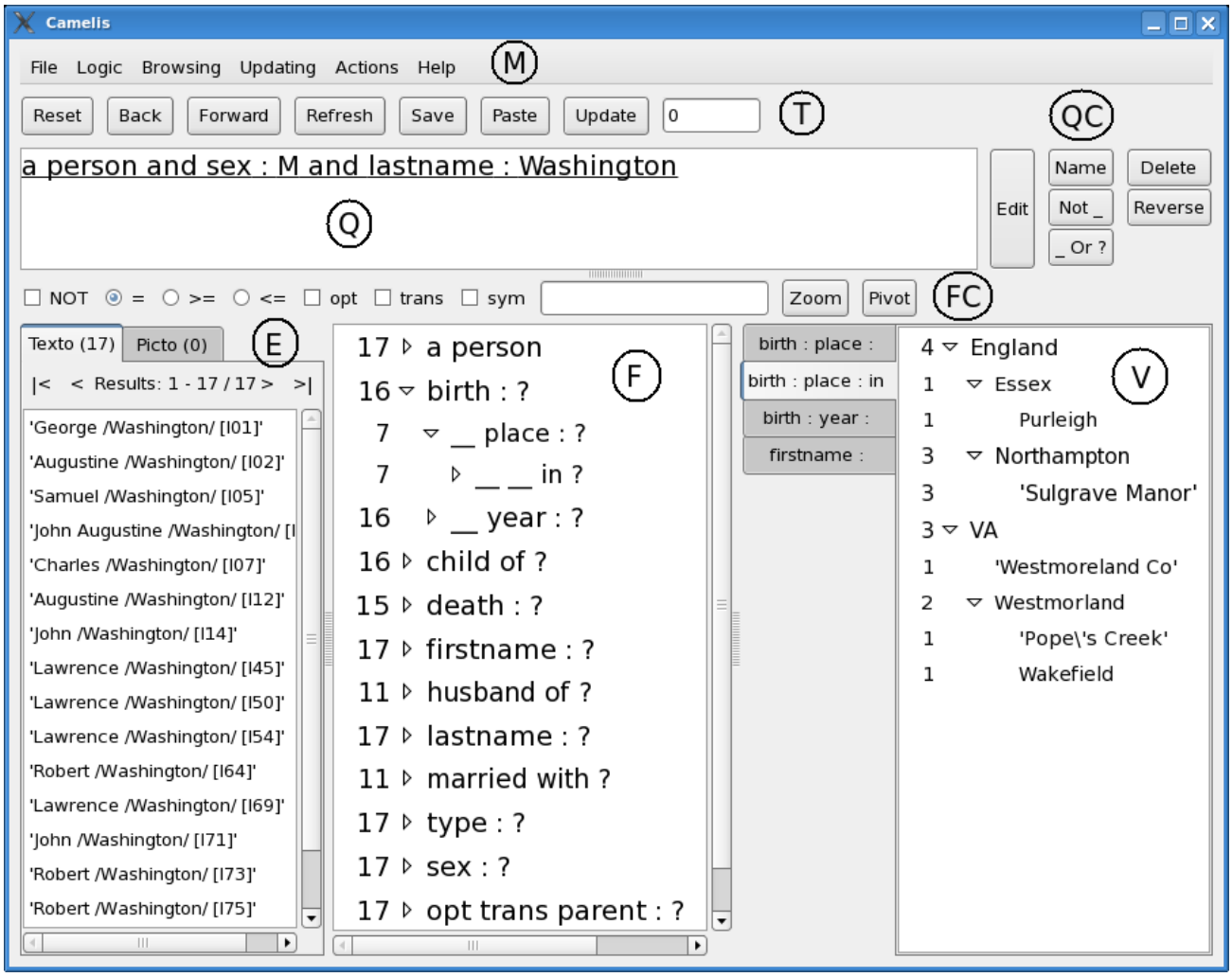

Figure 2: A screenshot of the user interface of Camelis 2. It shows the selection of male persons whose lastname is Washington.

and transitive property meaning "is contained in". The last two facets make the distinction between the exact birthplace (e.g., "born at Wakefield"), and an approximate birthplace (e.g., "born in VA"). Finally, the expansion of a property facet opens a new value box (V), where it displays the relevant values for the facet. This also works for subfacets, and when the last property of a property chain is reflexive and transitive, i.e., a partial ordering, this partial ordering is used to organize the values as a taxonomy. For example, Figure 2 (V) shows the value box for the expansion of the facet `birth : place : in ?`. We can read that 3 persons among the answers were born in VA, among which 1 was born in Wakefield. Similarly, not illustrated here, a descendancy chart of the ancestors of the selected people can be obtained under the index term `opt trans parent : ?`, showing under each individual its children, and this recursively.

All navigation controls modify the query, which entails a change of the extension, and hence of the facet hierarchy and of the set of value boxes. All the elements of the extension box, the facet hierarchy and the value boxes are features. A feature has the same syntax and semantics as a query, and can be used as a restriction value. Feature controls (FC) are used to apply modifiers on selected features: e.g., applying negation to exclude resources having a feature, rather than selecting them. Query controls (QC) apply changes to the query, without using any feature. The 'Edit' button lets the user manually edit the query. Section 6.2 proves that it is never necessary, but it may be desirable for expert users. Other navigation controls are illustrated in the following through a navigation scenario.

3.2 A Navigation Scenario

We demonstrate, through a concrete scenario, how complex queries can be reached by navigation only, without typing anything. Table 1 details the navigation steps of our scenario. At each step, the first line gives the activated navigation controls, separated by +, and the remaining lines give the resulting query. After the first 10 steps, a complex query that

0	*Reset* ?
1	*Select* a person a person
2	*Expand* birth : ? + *Expand* birth : year : ? + *Select* birth : year : 1601 a person and birth : year : 1601
3	*Focus on* birth a person and birth : year : 1601
4	*Cross* place : ? + *Name* a person and birth : (year : 1601 and place : ?X)
5	*Expand* in ? + *Select* in England a person and birth : (year : 1601 and place : (?X and in England))
6	*Focus on* 1601 + (_ *Or ?)* a person and birth : (year : (1601 or ?) and place : (?X and in England))
7	*Select* 1649 a person and birth : (year : (1601 or 1649) and place : (?X and in England))
8	*Focus on* person + *Cross* father : birth : place : ? a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : ?
9	*(Not _)* a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : not ?
10	*Select* ?X + *Focus on* person a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : not ?X
11	*Focus on* year + *Delete* a person and birth : (year : ? and place : (?X and in England)) and father : birth : place : not ?X
12	*Focus on* father + *Reverse* father of (a person and birth : (year : ? and place : (?X and in England))) and birth : place : not ?X

Table 1: A navigation scenario in Camelis 2 on the genealogy of George Washington.

includes all features of LISQL is reached. Its meaning in English is "which person was born in 1601 or 1649 at some place X in England, and has a father who was born at another place than X". At step 0, the button 'Reset' in the toolbar (T) resets the query to the most general query ?. The extension box (E) lists all resources, and the facet hierarchy (F) lists the most general classes (e.g., a person, a family), and the most general properties (e.g., birth : ?, death of ?, in ?).

Steps 0-2 use only navigation controls of standard faceted search, i.e., facet expansion and restriction value selection. At step 1, the class a person is selected by double-click. The extension (E) is now the set of persons, and the facet hierarchy (F) is restricted to facets relevant to persons. At step 2, the facet birth : ? is first expanded to show its subfacets, e.g., birth : place : ? and birth : year : ?. The latter is then expanded with the effect of opening a new value box listing the relevant years. The value 1601 is selected in this value box, adding the constraint birth : year : 1601 on selected persons.

Steps 3-10 introduce the navigation controls of our semantic faceted search, i.e., focus change, naming, disjunction and negation introduction. At step 3, in order to put constraints on the birthplace, the focus is first changed to the birth event

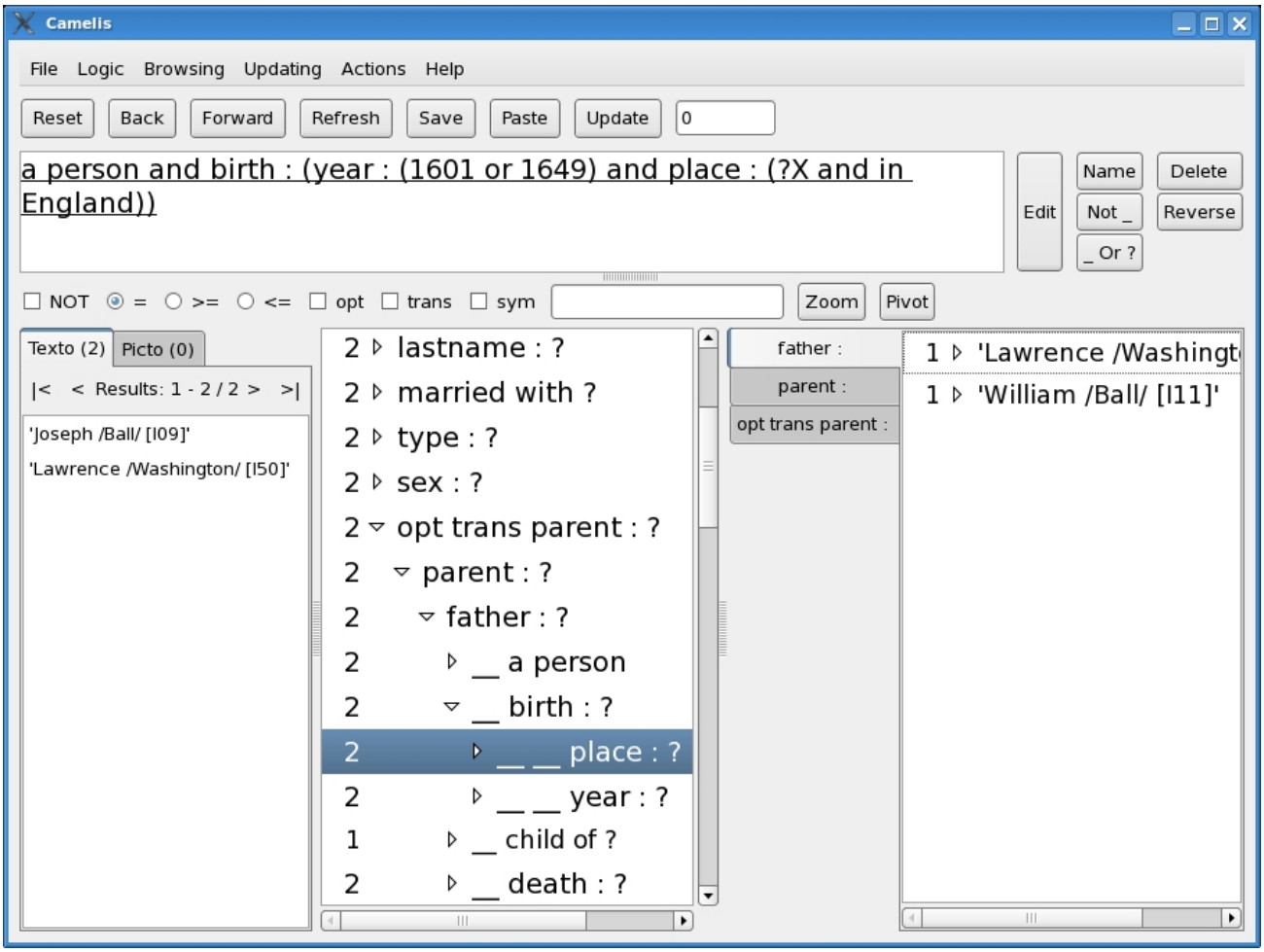

Figure 3: Screenshot of Camelis 2 at the step 9 of the navigation scenario, before the crossing.

by clicking the property `birth` in the query box. Because of the focus change, visible facets are now about the birth of persons born in 1601 (e.g., `year : ?`, `place : ?`). At step 4, the property facet `place : ?` is crossed, which means that the property facet is first selected, and that the focus is then moved to the object of the property, here the place of birth of persons born in 1601. Then, in order to refer to this birthplace later, we name it by pushing the button 'Name' in the query controls (QC). This introduces a variable `?X` in place of `?` (which can be seen as an anonymous variable). At step 5, to further constrain the birthplace, the facet `in ?` is expanded. This opens a new value box that displays a taxonomy of the relevant places. Among them, the value `England` is selected, adding the constraint `in England` to the birthplace.

At step 6, in order to relax the birthdate constraint by allowing other years, the focus is first set on the year by clicking the number `1601` (alternately, the word `year`) in the query box. Then, a disjunction is introduced by pushing the button '_ Or ?' in the query controls (QC). The focus is automatically set on the new alternative, and the extension box lists all relevant years for the birth of persons born in England. At step 7, the additional year `1649` is selected. At step 8, in order to put a constraint on the father of the person, the focus is first set on the person by clicking the word `person` in the query box. Then, we successively expand in the facet hierarchy the property facets `opt trans parent : ?`, `parent : ?`, `father : ?`, `father : birth : ?` to finally uncover and cross the subfacet `father : birth : place : ?`. Figure 3 shows a screenshot at this step, just before the crossing. The extension box lists the two remaining persons matching the query, and the visible value box lists their fathers. The facet hierarchy reveals the hierarchy under `opt trans parent : ?`: e.g., (`parent :`) is a subproperty of (`opt trans parent :`), which means "self or ancestor", (`father :`) is a subproperty of (`parent :`), `___ a person` is a class subfacet of `father : ?`, and `___ birth : ?` is a property subfacet of `father : ?`.

After the crossing, the focus is on the "father's birthplace of persons born in 1601 or 1649 at some place in England". The facet hierarchy contains the variable `?X` with a count that indicates that some but not all of the father's birthplace are the same as the person's birthplace. At step 9, in order to constrain the two birthplaces to be different, the button 'Not _' is first pushed in the query controls (QC) to introduce negation, moving the focus in the scope of the negation, and then the

variable `?X` is selected. Finally, the focus is set again on the person to get the results of our target query. There is only one answer in our dataset, namely 'Joseph Ball'.

Steps 11-12 show navigation controls that are useful in practice for the reversability of user actions, but that are not necessary for expressive navigation. Step 11 shows how constraints can be relaxed anywhere in the query. The constraint on the birthdate is relaxed by setting the focus on the year, and by pushing the button 'Delete' in the query controls (QC). Step 12 shows how the query can be reformulated from a different perspective. The query is reformulated, we say *reversed*, from the perspective of the person's father by setting the focus on the father, and by pushing the button 'Reverse' in the query controls. This is useful, for example, to delete all constraints about the person, while retaining the constraints about the father.

To complete our scenario, note that it is possible to navigate to a disjunction of complex queries, e.g., `(a man and lastname : Washington) or (a woman and father : lastname : Washington)`, and to the negation of a complex query, e.g., `a person and not (father : birth : year : ?X and mother : birth : year : ?X)`.

The buttons 'Back' and 'Forward' in the toolbar (T) provide navigation in the history of queries. The feature controls (FC) provide shortcuts for simple negations and disjunctions. When several features are selected in a same box, they are aggregated into a disjunction. If the checkbox 'NOT' is activated, then negation is further applied on this disjunction of features. To perform a selection of the resulting complex class, the button 'Zoom' has to be pushed in place of the double-click. Other controls allow for introducing inequalities in front of selected values, and relation closures in front of properties; the button 'Pivot' replaces the current focus, instead of refining it.

The navigation scenario covers all the navigational capabilities of Camelis 2 and semantic faceted search. Section 6 formalizes navigation as a navigation graph, and proves safeness and completeness w.r.t. LISQL, which is defined in Section 4.

4 The LIS Query Language (LISQL)

In this section, we first motivate the introduction of the new syntax LISQL for SPARQL queries, before we formally define its syntax and its translation to SPARQL. Because of the central role of queries in semantic faceted search, which is the key to expressive navigation, there are two constraints on their syntax.

The first constraint is an aesthetic one. The syntax is not only used for the query itself, but also for the facets, and restriction values. If we compare the query at step 10 in Table 1 with the focus on the father, first expressed in SPARQL:

```
SELECT DISTINCT ?f
WHERE {
    ?p a person.
    ?p birth ?b.
        ?b year ?y FILTER (?y=1601 || ?y=1649).
        ?b place ?X. ?X in England.
    ?p father ?f.
        ?f birth ?bf.
            ?bf place ?pf FILTER ?pf != ?X }
```

and then in LISQL:

```
a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father :
birth : place : not ?X,
```

it can be noted that the latter is more concise, and makes a minimal use of variables, and replaces a number of logical and algebraic symbols (curly brackets, dot, UNION, FILTER, `=`, `!=`, `&&`, `||`, and `!`) by keywords for the 3 Boolean operators (`and`, `or`, `not`) plus brackets. The LISQL syntax follows the usual syntax for expressions (infix operators and brackets to fix priorities), and we think that this makes it easier to read and learn.

The second constraint is a functional one. Given that queries are obtained by successive applications of query transformations, it is necessary to go through intermediate queries that have no obvious syntax in SPARQL. For example, in the query at step 6 in Table 1, how would be written the query part `year : (1601 or ?)` ? Also, when introducing a disjunction, which form should be chosen: the algebraic UNION between graph patterns, or the logical `||` between constraints ? And suppose the first alternative is a filter, and the second alternative is a graph pattern, like in `year : (1601 or year of birth of Goerges` ? Similar difficulties apply to negation.

The LISQL syntax can be related to the Turtle syntax, and to the abstract syntax of DL complex classes. Like Turtle, LISQL uses spanning trees over graph patterns to hide some variables. The LISQL syntax can be derived from Turtle by replacing the semi-colon `;` by the keyword `and`, and square brackets by round brackets (optional according to priorities),

and by adding disjunction (`or`), negation (`not`), and property modifiers (e.g., `of`, `trans`). Compared to OWL-DL, a LISQL query has a syntax similar to complex classes with two noticeable differences. First, variables can be used as co-references, thus allowing for the expression of cycles in the underlying graph pattern. Second, the focus can be moved on different parts of the complex class, rather than being implicitly set on the root. However, disjunction and negation have an extensional semantics, like in SPARQL and unlike in OWL-DL.

4.1 LISQL Syntax

A LISQL query is defined as the combination of a complex class and a focus. We first define *complex properties*, which enter in the composition of complex classes. They allow for the inverse and various closures of a property, without the need to give them an explicit name.

Definition 2 (complex property) *A* complex property *is any of:*

(*p* `:`) *the property p itself,*

(*p* `of`) *the inverse of the property p,*

(*p* `with`) *the symmetric closure of the property p,*

(`trans` *P*) *the transitive closure of the complex property P ("transitively P"),*

(`opt` *P*) *the reflexive closure of the complex property P ("optionally P").*

Given the property `parent`, the complex properties have the following meaning:

- (`parent :`) "has a parent",
- (`parent of`) "is a parent of",
- (`parent with`) "is in a parent relationship with",
- (`trans parent :`) "has an ancestor",
- (`opt parent :`) "is or has a parent that is".

Applying the three closures, (`opt trans` *p* `with`), defines an equivalence relation, while (`opt trans` *P*) defines a partial ordering if *P* is antisymmetric, and a pre-order otherwise. In the following, we use (`in`) as an abbreviation for the complex property (`opt trans part of`), which defines a spatial ordering between places.

Definition 3 (complex class) *Let V be an infinite set of variables, disjoint with the set of resources R. For every resource $r \in R$, variable $v \in V$, URI $u \in U$, complex property P, and complex classes C, C_1, C_2, the following expressions are also complex classes (in decreasing priority for operators):*

$$r \mid \texttt{?}v \mid \texttt{?} \mid \texttt{a}\ u \mid P\ C \mid \texttt{not}\ C_1 \mid C_1\ \texttt{and}\ C_2 \mid C_1\ \texttt{or}\ C_2.$$

A complex class denotes a set of resources, its *instances*. A resource x is an instance of a complex class C, "x is a C" for short, if:

- (r) "x is equal to r",
- (`?`v) "x is equal to the resource `?`v is bound to",
- (`?`) "x is any resource",
- (`a` u) "x is an instance of the atomic class u",
- (P C) "x is in a P-relation with a y that is a C",
- (p `:` C) "x has a p that is a C",
- (`not` C_1) "x is not a C_1",
- (C_1 `and` C_2) "x is a C_1 and a C_2",
- (C_1 `or` C_2) "x is a C_1 or a C_2.

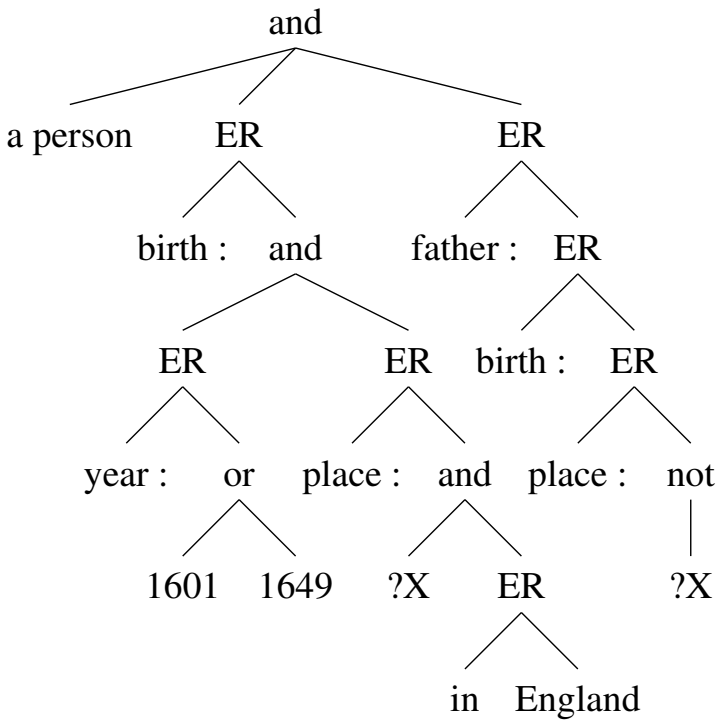

Figure 4: The syntax tree of the complex class `a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : not ?X`.

A complex class $P\ C$ is called an *existential restriction*, a *restriction* for short, in analogy with OWL-DL. Conjunction and disjunction are associative and commutative. The variables `?`v allows for the expression of cyclical graph patterns. The notation $(p\ :\ r)$ is reminiscent of the notation of valued attributes. For example, in the expression (`name : "John"`), `name` is the attribute, and `"John"` is the value. The expression can be read "has John as a name", or "whose name is John".

The complex class C_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : not ?X` uses all class constructs of LISQL, and denotes the set of "persons born in 1601 or 1649 at some place in England, and whose father is born at another place". Figure 4 shows this complex class as a syntax tree, where ER denotes an existential restriction with the complex property as left child.

A same complex class can be used to denote several sets of resources, simply by setting the *focus* on one or another node of the syntax tree of the complex class. The *default focus* is the root of this syntax tree.

Definition 4 (focus) *A* focus *of a complex class C is a complex class node of the syntax tree of C, or equivalently, an occurrence of a complex class as a subexpression of C. The set of foci of a complex class is denoted by $\Phi(C)$. Given a complex class C and a focus $\phi \in \Phi(C)$, $C[\phi]$ denotes the subexpression of C at the focus ϕ.*

In the following, when it is necessary to refer to a focus in a complex class, the corresponding subexpression is underlined with a subscript, like in $P\ \underline{C}_{\phi}$. If a complex class has several occurrences of a same subexpression, they correspond to different foci. Those are distinguished by subscripts when necessary. In Figure 4, every node is a focus, except properties (left children of ER nodes).

A query is then defined not only by a complex class, but also by the choice of one of the foci of this complex class.

Definition 5 (query) *A* query *is a pair $q = (C, \phi)$, where C is a complex class and $\phi \in \Phi(C)$ is a focus of C.*

The meaning of a query is not simply the meaning of the subexpression at the focus, but takes into account the whole query. For instance, assume the query $q_{ex} = (C_{ex}, \phi_{ex})$, where ϕ_{ex} is the node of the subexpression `(1601 or 1649)`. Its meaning is not "1601 or 1649", but "a year among 1601 and 1649 that is the birthdate of a person born at some place X in England and whose father was born at another place than X". If the focus is on first occurrence of the variable `?X`, the meaning is "a place in England that is the birthplace of a person born in 1601 or 1649, and that is not the birthplace of his father". Several foci can lead to the same meaning if they point to the same entity. The case of foci occurring in the scope of negation or disjunction is more subtle, and is discussed in the next section.

```
SELECT DISTINCT ?x4
WHERE
  ((?x1,rdf:type,gen:person) AND
  (?x1,gen:birth,?x2) AND
  (?x2,gen:year,?x3) AND
  ((R(?x3) FILTER ?x3 = 1601)
     UNION (R(?x3) FILTER ?x3 = 1649)) AND
  (?x2,gen:place,?x4) AND
  ((R(?x4) AND R(?X)) FILTER ?x4 = ?X) AND
  (?x5,gen:part*,?x4) AND
  (R(?x5) FILTER ?x5 = gen:England) AND
  (?x1,gen:father,?x6) AND
  (?x6,gen:birth,?x7) AND
  (?x7,gen:place,?x8) AND R(?x8))
MINUS
  ((R(?x8) AND R(?X)) FILTER ?x8 = ?X)
```

Table 2: SPARQL translation of the LISQL query `a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : not ?X`.

4.2 LISQL Translation to SPARQL

A semantics for LISQL, and a practical way to compute answers to queries, is obtained by defining a translation to one-dimensional SPARQL queries. Table 2 shows the translation of the query that uses the complex class C_{ex} introduced in previous section, and that sets the focus on the birthplace. The property `gen:part*` denotes the reflexive and transitive closure of the property `gen:part` (recall that `in` = `opt trans part of`). Simplifications and optimizations can possibly be applied on the graph pattern. For instance, the MINUS constraint can be simplified as `FILTER ?x8 != ?X`.

complex class C	simplified complex class $simpl_\phi(C)$
$P\ \underline{C'}$	$P\ simpl_\phi(C')$
$\underline{C_1}$ `and` C_2	$simpl_\phi(C_1)$ `and` C_2
C_1 `and` $\underline{C_2}$	C_1 `and` $simpl_\phi(C_2)$
$\underline{C_1}$ `or` C_2	$simpl_\phi(C_1)$
C_1 `or` $\underline{C_2}$	$simpl_\phi(C_2)$
`not` $\underline{C_1}$	$simpl_\phi(C_1)$
otherwise	C

Table 3: Rules for the simplification of complex classes, depending on the position of the focus ϕ.

Given a query, the complex class is translated into a graph pattern, and the focus indicates which variable comes into the SELECT clause. However, when the focus occurs in the scope of a negation or a disjunction, the complex class is simplified so that the focus is no more in the scope of a negation or a disjunction. Indeed, how to interpret the query `a man and not a doctor`, where the focus is in the scope of a negation ? Assume a complex class C_1 `and not` C_2, which means "a C_1 that is not a C_2" at the default focus. Setting the focus on C_2 can be used to look at which resources in C_1 are excluded from the set of answers. Therefore, it is equivalent to set the focus on C_2 in the complex class C_1 `and` C_2, meaning "a C_2 that is a C_1". About disjunction, assume a complex class C_1 `and (`C_2 `or` C_3`)`, meaning "a C_1 that is a C_2 or a C_3" at the default focus. Setting the focus on C_2 means "a C_2 that is a C_1 and that may also be a C_3", which is equivalent to "a C_2 that is a C_1". Therefore, disjunction alternatives that are not in the scope of the focus can be removed without changing the meaning of the query. Table 3 gives the simplification rules that apply on complex classes depending on the position of the focus. The underline indicates in which part of the complex class the focus stands.

The ability to focus in the scope of disjunctions and negations is crucial for the completeness of the navigation graph (see Section 6.2). A focus in the scope of a disjunction enables to access an alternative while temporarily hiding the other alternatives. A focus in the scope of a negation gives access to the set of resources to be excluded from a larger selection of resources.

expression	graph pattern / graph pattern modifier
P	$\gamma(x, P^\alpha, y)$
p `:`	$(?x, p^\alpha, ?y)$
p `of`	$(?y, p^\alpha, ?x)$
p `with`	$(?x, p^\alpha, ?y)$ `UNION` $(?y, p^\alpha, ?x)$
`opt` P_1	$\gamma(x, P_1^{\{?\}\cup\alpha}, y)$
`trans` P_1	$\gamma(x, P_1^{\{+\}\cup\alpha}, y)$
C	$\gamma(x, C)$
r	$R(?x)$ `FILTER` $?x = r$ / $\lambda g.g$
$?v$	$(R(?x)$ `AND` $R(?v))$ `FILTER` $?x = ?v$ / $\lambda g.g$
`?`	$R(?x)$ / $\lambda g.g$
`a` u	$(?x, \texttt{rdf:type}, u)$ / $\lambda g.g$
$P\ C_y$	g_P `AND` g_y / f_y where y is a fresh variable, $g_P = \gamma(x, P^\emptyset, y)$, $(g_y, f_y) = \gamma(y, C_y)$
C_1 `and` C_2	g_1 `AND` g_2 / $f_2 \circ f_1$
`not` C_1	$R(?x)$ / $\lambda g.(g$ `MINUS` $f_1(g_1))$
C_1 `or` C_2	$f_1(g_1)$ `UNION` $f_2(g_2)$ / $\lambda g.g$

Table 4: Translation from complex classes and properties to graph patterns.

We now define a translation from complex classes and properties to graph patterns. Table 4 defines γ by induction on complex classes and complex properties. In this table, graph patterns are represented in the compositional syntax of SPARQL (see Section 2.1), and for every $i \in \mathbb{N}$, we assume $(g_i, f_i) = \gamma(x, C_i)$. $\gamma(x, P^\alpha, y)$ is a graph pattern representing the complex property P between x and y, under the relation closure α. The relation closure $\alpha \subseteq \{?, +\}$ is a composition of relation closures, where ? (resp. +) denotes the reflexive (resp. transitive) closure of a binary relation. $\gamma(x, C)$ returns a graph pattern g, and a graph pattern modifier f, that together represent the fact that x is an instance of the complex class C. The graph pattern modifier is here only to handle negations, such that the MINUS operator is applied on the largest possible conjunctive graph pattern. Overlooking this, the example query above would always return empty results. The line where $C = P\ C_y$ can be read as follows. A fresh variable y is chosen to represent the object of property P, and the subject of class C_y. The returned graph pattern is the AND of the graph pattern of the complex property P between x and y, and the graph pattern of the complex class C_y on y. The returned graph pattern modifier is the graph pattern modifier of C_y.

Definition 6 *Let $q = (C, \phi)$ be a query. The SPARQL translation of q is defined by*

$$\Gamma(q) = \texttt{SELECT DISTINCT } ?v \texttt{ WHERE } f(g)$$

where $x \in V$ is a fresh variable not occurring in C, $(g, f) = \gamma(x, simpl_\phi(C))$, and v is the variable such that $\gamma(v, C[\phi])$ has been evaluated during the evaluation of $\gamma(x, C)$.

We now define the *extension* of a LISQL query as the set of answers of the SPARQL translation of the query.

Definition 7 (extension) *Let $\mathcal{R}$ be a RDF graph, and q be a query. The* extension *of q in $\mathcal{R}$, noted $ext_\mathcal{R}(q)$, is the set of resources that are answers to its SPARQL translation $\Gamma(q)$. Every element of the extension is called an* instance *of the query q.*

The extension of queries plays a central role in the definition of a safe and complete navigation graph, as it allows to discriminate between inhabited queries and dead-ends.

5 Query Transformations

We here define a collection of query transformations that are the basis in Section 6 for navigation links from query to query. In each definition of a query transformation, we assume a query $q = (C, \phi)$. A query transformation is denoted by an expression into brackets, $[t]$, and the query resulting from the transformation is noted $q' = q[t]$. The composition of transformations can then be noted $q[t_1][t_2]$, or simply $q[t_1; t_2]$, where t_1 is applied first. In order to define query transformations, we use the notation $C' = C[\phi \leftarrow C_1]$ to represent the complex class C' resulting from the substitution of the subexpression of C at

focus ϕ by the complex class C_1. The subexpression $C[\phi]$ can be reused in C_1. In the examples below, queries are displayed as complex classes with the focus underlined.

A *focus change* simply changes the focus of the query, without modifying the complex class. For example, (a person and birth : place : ?X) [*focus* ?X] = (a person and birth : place : ?X).

Definition 8 (focus-change) *Let $\phi' \in \Phi(C)$ be a focus of C.*

$$(C, \phi)[focus\ \phi'] = (C, \phi').$$

An *and-insertion* inserts a conjunction with a given complex class D at the current focus, and sets the focus on this new complex class. For example, (a person and birth : year : 1601) [*and* place : in England] = (a person and birth : (year : 1601 and place : in England)). When the current focus is the default one, this corresponds to the application of a restriction value in faceted search, aka. *zoom-in*.

Definition 9 (and-insertion) *Let D be a complex class.*

$$(C, \phi)[and\ D] = (C[\phi \leftarrow C[\phi]\ \texttt{and}\ \underline{D}_{\phi'}], \phi').$$

In this definition, the focused subexpression $C[\phi]$ is replaced by $(C[\phi]\ \texttt{and}\ D)$, and the focus is changed to the focus ϕ' of D. When $C[\phi] = \texttt{?}$, it is simply replaced by D.

A *crossing* is a sequence of two query transformations. The first transformation is the and-insertion of a restriction $(P\ D)$, and the second transformation sets the focus on the value of this restriction. For example, (a person) [*cross* birth : ?] = (a person and birth : ?).

Definition 10 (crossing) *Let P be a complex property, and D be a complex class.*

$$(C, \phi)[cross\ P\ D] = (C, \phi)[and\ P\ \underline{D}_{\phi'}; focus\ \phi'].$$

An *or-insertion* inserts a disjunction with a given complex class D at the current focus, and sets the focus on this new complex class. For example, (a person and birth : year : 1601) [*or* 1649] = (a person and birth : year : (1601 or 1649)).

Definition 11 (or-insertion) *Let D be a complex class.*

$$(C, \phi)[or\ D] = (C[\phi \leftarrow C[\phi]\ \texttt{or}\ \underline{D}_{\phi'}], \phi').$$

A *not-insertion* inserts a negation at the current focus, and sets the focus on the negated complex class, rather than setting it on the negation itself. For example, (a person and birth : place : in England) [*not*] = (a person and birth : place : not in England).

Definition 12 (not-insertion)

$$(C, \phi)[not] = (C[\phi \leftarrow \texttt{not}\ \underline{C[\phi]}_{\phi'}], \phi').$$

A *minus-insertion* is an and-insertion followed by a not-insertion. For example, (a person and birth : place : ?X) [*minus* in England] = (a person and birth : place : (?X and not in England)).

Definition 13 (minus-insertion) *Let D be a complex class.*

$$(C, \phi)[minus\ D] = (C, \phi)[and\ D; not].$$

An important property of user interfaces is the reversability of user actions. A focus change can obviously be reversed by another focus change. For and-insertion, or-insertion, and not-insertion, we respectively introduce the respective query transformations: and-deletion [*delete and*], or-deletion [*delete or*], and not-deletion [*delete not*]. Or-deletion can only apply on disjunction alternatives, and not-deletion can only apply under a negation. Because of the equivalence $C \equiv \texttt{?}\ \texttt{and}\ C$, and-deletion can apply to every focus.

As an example of a sequence of query transformations, the sequence [*and* a person; *cross* birth : ?; *cross* year : ?; *and* 1601; *or* ?; *and* 1649; *focus* (year : ..); *cross* place : ?; *and* ?X; *cross* in ?; *and* England; *focus* (a person); *cross* father : ?; *cross* birth : ?; *cross* place : ?; *minus* ?; *and* ?X; *focus* (a person)] leads from the most general query ? to our example query a person and birth : (year : (1601 or 1649) and place : (?X and in England)) and father : birth : place : not ?X. This provides a formalization of the 10 first steps of the navigation scenario presented in Section 3.2.

6 Navigation Graph

In this section, we define a navigation graph that is fully derived from a RDF graph and the selection of a *vocabulary* of complex classes (candidate facets and value restrictions). Each node of this graph is a LISQL query, and each edge is a couple of queries (q, q') such that q' is the result of applying a query transformation on q. We prove the safeness of a navigation graph in Section 6.1, and its completeness in Section 6.2.

We first define the set of candidate transformations of a query, parametrized by a *vocabulary*. In this definition, $vars(C)$ denotes the set of variables that occur in a complex class C.

Definition 14 (candidate transformation) *Let $\mathcal{C}$ be a set of complex classes, called* vocabulary*, and $q = (C, \phi)$ be a LISQL query. The set of* candidate transformations *of q, noted $T_{\mathcal{C}}(q)$, is made of:*

- *a focus-change [focus ϕ'], for every focus $\phi' \in \Phi(C)$;*
- *an and-insertion [and ?v], for one fresh variable $v \notin vars(C)$;*
- *an and-insertion [and D], for every complex class $D \in \mathcal{C}$ s.t. $vars(D) \subseteq vars(C)$;*
- *a crossing [cross P D], for every restriction P $D \in \mathcal{C}$ s.t. $vars(P\ D) \subseteq vars(C)$;*
- *an or-insertion [or ?];*
- *a not-insertion [not];*
- *a minus-insertion [minus ?];*
- *an and-deletion [delete and];*
- *an or-deletion [delete or], when applicable on q;*
- *a not-deletion [delete not], when applicable on q.*

Having only one transformation with a fresh variable is sufficient because inserting a fresh variable or another makes no difference from a semantic point of view. A general or-insertion [*or* D] would be redundant, because it can be decomposed into [*or* ?; *and* D]. Similarly, a general minus-insertion [*minus* D] can be decomposed into [*minus* ?; *and* D].

For practical use, it is important to have a finite number of candidate transformations for each query. Because candidate transformations are derived from a vocabulary, we define finiteness on the vocabulary itself. A vocabulary can be *locally-finite* even if it is an infinite set of complex classes, because there are constraints on which complex classes can be used in the transformations of a given query (Definition 14).

Definition 15 (locally-finite vocabulary) *A vocabulary $\mathcal{C}$ is said* locally-finite *if, for every query q, the set $T_{\mathcal{C}}(q)$ is finite.*

In order to avoid transformations that lead to a dead-end, we define *safe* transformations as a subset of the candidate transformations. The safeness of a transformation depends on the actual dataset, a RDF graph.

Definition 16 (safe transformation) *Let $\mathcal{C}$ be a vocabulary of complex classes, $\mathcal{R}$ be a RDF graph, and q be a query. A candidate transformation $t \in T_{\mathcal{C}}(q)$ is* safe *w.r.t. $\mathcal{R}$ iff both q and $q[t]$ have an answer: $ext_{\mathcal{R}}(q) \neq \emptyset$, $ext_{\mathcal{R}}(q[t]) \neq \emptyset$. The set of safe transformations of a query q is noted $ST_{(\mathcal{C},\mathcal{R})}(q)$, or simply $ST(q)$ when there is no ambiguity.*

We now define a generic navigation graph, parametrized by a *vocabulary* of complex classes, and a RDF graph. This is the graph induced by safe transformations of LISQL queries.

Definition 17 (navigation graph) *Let $\mathcal{C}$ be a vocabulary of complex classes, and $\mathcal{R}$ be a finite RDF graph. The navigation graph $G(\mathcal{C}, \mathcal{R})$ is a graph whose set of nodes is the set of LISQL queries, and whose set of edges is defined as*

$$\{(q, q') \mid \exists t \in ST_{(\mathcal{C},\mathcal{R})}(q) : q' = q[t]\}.$$

As a consequence, a query that has no answer, i.e., whose extension is empty, has no outgoing edge in the navigation graph. Therefore, it is a dead-end, and it provides no navigation link. A navigation graph based on a locally-finite vocabulary is also said locally-finite. A locally-finite navigation graph has a finite number of outgoing edges at every node.

6.1 Safeness

A navigation graph is *safe* if no path can lead to a dead-end, unless it starts at a dead-end.

Theorem 18 (safeness) *Let $\mathcal{C}$ be a vocabulary, and $\mathcal{R}$ be a RDF graph. The navigation graph $G(\mathcal{C},\mathcal{R})$ is* safe*, i.e., for every path going from the query q to the query q', $ext_{\mathcal{R}}(q) \neq \emptyset$ implies $ext_{\mathcal{R}}(q') \neq \emptyset$.*

This is true by definition of the navigation graph because only safe transformations are used for navigation links, ensuring the target query is not a dead-end. There remains the issue of choosing a starting query that is not a dead-end. A natural choice is the *top query* $\top = (\underline{?}_{\phi}, \phi)$. Its extension is the set of all resources in $\mathcal{R}$, which is never empty in practice (otherwise, there is nothing to explore).

6.2 Completeness

A navigation graph is *complete* if there is a path to every query that is not a dead-end, starting from the most general query $\top$. Proving that there is a finite sequence of transformations from a finite vocabulary to every query would be easy. What is more difficult, but more interesting, is to prove there is a *safe* sequence of transformations, i.e., a path in the navigation graph. There are some queries for which there is no such path, but we show that this happens only if there is an unsafe focus-change in the query. For example, the query `a person and mother : not a man` results in the set of persons having a known mother, but a focus-change on `a man` results in the empty set: no man is the mother of a person. The navigation path that would lead to it is `[`*and* `a person; `*cross* `mother : ?; `*minus* `?; `*and* `a man]`, but the last query transformation is not safe because it leads to a dead-end. In fact, the above query is equivalent to `a person and mother : ?` w.r.t. the extension. A similar observation can be made with disjunction. In the query, `a person and mother : (a man or a woman)`, a focus-change on `a man` would result in a dead-end. The query can be simplified into the equivalent query, w.r.t. the extension, `a person and mother : a woman`. Therefore, we characterize what a *safe complex class* is.

Definition 19 (safe complex class) *A complex class C is said to be* safe under $\phi \in \Phi(C)$ *if for every focus $\phi' \in \Phi(C)$ that is under ϕ in the syntax tree of C, we have $ext_{\mathcal{R}}((C,\phi')) \neq \emptyset$. C is said* fully safe *if it is safe under its default focus.*

Before stating and proving the main theorem on completeness, we need a few lemmas on the conservation of the safeness of complex classes when they are simplified. The proofs, only sketched here, are based on the translation of queries to SPARQL (Definition 6).

Lemma 20 *If a complex class C is safe under a focus ϕ, then $C' = C[\phi \leftarrow \underline{?}_{\phi}]$ is safe under ϕ.*

Proof: Assume C is safe under ϕ, and C' is not safe under ϕ. Then $ext((C',\phi)$ is empty. As C has the complex class $C[\phi]$ instead of the most general complex class `?` at the the focus ϕ, the translation to SPARQL of (C,ϕ) has more constraints than (C',ϕ), and the additional constraints are not in the scope of a MINUS. Therefore, $ext((C,\phi))$ is empty too, and C is not safe under ϕ. This contradicts our hypothesis, and proves the lemma. □

Lemma 21 *If a complex class C is safe under a focus ϕ and $C[\phi] = C_1$ `and` C_2, then $C' = C[\phi \leftarrow \underline{C_1}_{\phi}]$ is safe under ϕ.*

Proof: Assume C is safe under ϕ, and C' is not safe under ϕ. Then, there exists a focus ϕ' under ϕ s.t. $ext((C',\phi')$ is empty. As C has the complex class C_1 `and` C_2 instead of complex class C_1 at the the focus ϕ, the translation to SPARQL of (C,ϕ') has more constraints than (C',ϕ'), and the additional constraints are not in the scope of a MINUS. Therefore, $ext((C,\phi'))$ is empty too, and C is not safe under ϕ. This contradicts our hypothesis, and proves the lemma. □

Lemma 22 *If a complex class C is safe under a focus ϕ and $C[\phi] = C_1$ `or` C_2, then $C' = C[\phi \leftarrow \underline{C_1}_{\phi}]$ is safe under ϕ.*

Proof: Assume C is safe under ϕ, and C' is not safe under ϕ. Then, there exists a focus ϕ' under ϕ s.t. $ext((C',\phi')$ is empty. Now, (C,ϕ') has the same simplification as (C',ϕ') because ϕ' is in the first alternative of the disjunction C_1 `or` C_2. Hence, (C,ϕ') has the same SPARQL translation and the same extension as (C',ϕ). Therefore, $ext((C,\phi'))$ is empty too, and C is not safe under ϕ. This contradicts our hypothesis, and proves the lemma. □

We also need the fact that the complex class `?` is a neutral element for conjunction, and an absorbing element for disjunction: i.e.,

- `? and` $C \equiv C$ `and ?` $\equiv C$,
- `? or` $C \equiv C$ `or ?` $\equiv$ `?`.

We now prove that it is enough to use a vocabulary restricted to *atomic* complex classes (i.e., resources, variables, classes, and unqualified restrictions) to generate navigation graphs that are both locally-finite and complete.

Definition 23 (minimum vocabulary) *Let $\mathcal{R}$ be a RDF graph. The* minimum vocabulary $\mathcal{C}_0$ *is made of resource names of $\mathcal{R}$ (i.e., URIs, literals), variables, the complex class* `?`*, class names of $\mathcal{R}$ (*`a` *u), and unqualified restrictions (*P *?) using property names of $\mathcal{R}$.*

Theorem 24 (finiteness) *The minimum vocabulary $\mathcal{C}_0$ is locally-finite.*

Proof: In a RDF graph, there is a finite number of resources, literals, classes, and properties. Therefore, the only source of infiniteness is variables. However, the candidate transformations of a query use only the variables of the query (Definition 14). □

Theorem 25 (completeness) *Let $\mathcal{C}$ be a vocabulary containing the minimum vocabulary $\mathcal{C}_0$, and $\mathcal{R}$ be a RDF graph. The navigation graph $G(\mathcal{C},\mathcal{R})$ is complete for fully safe queries, i.e., for every query $q=(C,\phi)$ s.t. C is a fully safe complex class, there is a path in G from the top query $\top=(\underline{?}_{\phi},\phi)$ to q.*

Proof: We first prove that for every queries $q=(C,\phi)$ and $q'=(C',\phi')$ s.t. $q'=q[and\ D]$ (i.e., $C'=C[\phi \leftarrow C[\phi]$ `and` $\underline{D}_{\phi'}]$), $ext(q)\neq\emptyset$, and q' is safe under ϕ', there is a navigation path from q to q'. Because C' is safe under ϕ', we know that q' is not dead-end, hence $ext(q')\neq\emptyset$. We proceed by induction on the complex class D:

$D=r$**:** we have $q'=q[and\ r]$, and $r\in\mathcal{C}$, hence $[and\ r]\in T_{\mathcal{C}}(q)$. Moreover, as q and q' are not dead-ends, $[and\ r]\in ST(q)$. Therefore, (q,q') is a navigation link.

$D=?v$**:** same as above with $q'=q[and\ ?v]$, whether $?v$ occurs in C or not.

$D=?$**:** same as above with $q'=q[and\ ?]$.

$D=$ `a` u**:** same as above with $q'=q[and$ `a` $u]$.

$D=P\ \underline{D_1}_{\phi_1}$**:**
Let $q_1=q[cross\ P\ ?]=(C_1,\phi_1)$, where $C_1=C[\phi\leftarrow C[\phi]$ `and` $P\ \underline{?}_{\phi_1}]$. Because C' is safe under ϕ', C' is also safe under ϕ_1, and by Lemma 20, C_1 is safe under ϕ_1. Therefore, q_1 is not a dead-end, and hence (q,q_1) is a navigation link.

Let $q_2=q_1[and\ D_1]=(C_1[\phi_1\leftarrow$ `? and` $D_1],\phi_1)=(C_2,\phi_1)$. We have $C_2\equiv C'$. Because q_1 is not a dead-end and C' is safe under ϕ_1, there is path from q_1 to q_2, by induction on D_1.

Finally, $q'=q_2[focus\ \phi']$ and (q_2,q') is a navigation link because q' is not a dead-end.

$D=\underline{D_1}_{\phi_1}$ `and` $\underline{D_2}_{\phi_2}$**:**
Let $q_1=q[and\ D_1]=(C_1,\phi_1)$. Because C' is safe under ϕ', C' is safe under ϕ_1, and by Lemma 21, C_1 is safe under ϕ_1. Therefore, there is a path from q to q_1, by induction on D_1.

We have $q_2=q_1[and\ D_2]=(C',\phi_2)$. Because q_1 is not a dead-end, and C' is safe under ϕ_2 (Lemma 21), there is a path from q_1 to q_2, by induction on D_2.

Finally, $q'=q_2[focus\ \phi']$ and (q_2,q') is a navigation link because q' is not a dead-end.

$D=$ `not` $\underline{D_1}_{\phi_1}$**:**
Let $q_1=q[minus\ ?]=(C_1,\phi_1)$, where $C_1=C'[\phi_1\leftarrow ?]$. Because C' is safe under ϕ', C' is also safe under ϕ_1, and hence, C_1 is safe under ϕ_1 (Lemma 20). Therefore, q_1 is not a dead-end, and (q,q_1) is a navigation link.

Let $q_2=q_1[and\ D_1]=(C_2,\phi_1)$. We have $C_2\equiv C'$, and hence, C_2 is safe under ϕ_1. Therefore, there is a path from q_1 to q_2, by induction on D_1.

Finally, $q'=q_2[focus\ \phi']$ and (q_2,q') is a navigation link because q' is not a dead-end.

$D=\underline{D_1}_{\phi_1}$ `or` $\underline{D_2}_{\phi_2}$**:**
Let $q_1=q[and\ D_1]=(C_1,\phi_1)$, where $C_1=C'[\phi'\leftarrow D_1]$. Because C' is safe under ϕ', C' is safe under ϕ_1, and by Lemma 22, C_1 is safe under ϕ_1. Therefore, there exists a path from q to q_1, by induction on D_1.

Let $q_2=q_1[or\ ?]=(C_2,\phi_2)$, where $C_2=C'[\phi'\leftarrow(D_1$ `or` $?)]\equiv C'[\phi'\leftarrow ?]\equiv C$. Therefore, q_2 is not a dead-end, and (q_1,q_2) is a navigation link.

Let $q_3=q_2[and\ D_2]=(C_3,\phi_2)$: $C_3=C'[\phi_2\leftarrow$ `? and` $D_2]\equiv C'$. Because C' is safe under ϕ', C' is safe under ϕ_2. Therefore, there is a path from q_2 to q_3, by induction on D_2.

Finally, $q'=q_3[focus\ \phi']$, and (q_3,q') is a navigation link because q' is not a dead-end.

We now prove that there exists a path from the top query to every query $q = (C, \phi)$ s.t. C is fully safe. By induction on C, there is a path from the top query to the query (`? and` $\underline{C_\phi}, \phi$), which is equivalent to q. □

This proof also provides an algorithm for finding a path from the top query to the target query. It exhibits a linear complexity, i.e., the path has a length that is linear in the size of the target query.

6.3 Complexity Issues

Navigation boils down to compute a set of transformations of the current query $q = (C, \phi)$, and to present them to the user. For a safe navigation, only safe transformations should be presented; and for a complete navigation, at least the minimum vocabulary $\mathcal{C}_0$ should be used. In this section, we discuss the cost of computing those navigation links. The first aspect is the number of candidate transformations of the current query. The second aspect is verifying whether a candidate transformation is safe.

From Definition 14, the number of candidate transformations $T_{\mathcal{C}}(q)$ is linear in the number of foci of C plus the size of the vocabulary $\mathcal{C}$ restricted to variables occurring in C. As the number of foci of a complex class is clearly finite, if the vocabulary, restricted to variables in C, is finite, then the set of candidate transformations is also finite. The size of the minimum vocabulary restricted to variables in C is linear in the number of variables in C plus the number of resources of the RDF graph (classes and properties are resources themselves). As the number of resources of a RDF graph is finite, the minimum vocabulary, restricted to variables in C, is also finite. As a conclusion, it is possible to have finite sets of candidate transformations while preserving navigation completeness. Every finite vocabulary that contains the minimum vocabulary is acceptable.

We now discuss the verification of the safeness of candidate transformations. By default, to verify that a transformation is safe, the extension of the target query must be computed and compared to the empty set. We show that some of them are always safe, saving the cost of verifying them. For other transformations, the verification can be reduced to a less costly computation. In the following lemmas, we have $q = (C, \phi)$ the current query, and $q' = (C', \phi')$ the target query obtained by applying a transformation on q. We assume that $ext(q) \neq \emptyset$, which is ensured by safe navigation, and we are interested in judgments about the fact that $ext(q') \neq \emptyset$.

Lemma 26 *Assume $q' = q[focus\ \phi']$, with $\phi' \in \Phi(C)$. If $simpl(q) = simpl(q')$, then $ext(q') \neq \emptyset$.*

When two queries have the same simplification (see Table 3), they are translated into the same graph pattern. As there are no optional graph pattern, the change of the variable in the SELECT clause cannot entail an empty extension. A focus-change does not change the simplified query when the focus does not enter or exit the scope of a disjunction or a negation. For other foci, the extension of q' has to be computed.

In fact, it is acceptable in practice not to check the safeness of a focus change. For example, consider the navigation to the query `a person and father : not a doctor and mother : a doctor`, where no dead-end is encountered, but where a focus on the first occurrence of `a doctor` is unsafe. In fact, this unsafe focus-change is an information in itself, revealing that nobody has both his father and mother that are doctors. The first part of the navigation leads to the query `a person and father : not a doctor`, which is not a dead-end because there are doctors who are fathers, as well as there are non-doctors who are fathers. Then, the second part of the navigation selects, among the persons whose father is not a doctor, those whose mother is a doctor. This is the second part that produces the unsafe focus-change, because of the exclusion, in the RDF graph, between having a doctor father and a doctor mother. Moreover, it may be useful to make this focus change in order to delete `a doctor` and replace it by something more general.

Lemma 27 *Assume $q' = q[and\ ?v]$, with $?v \notin vars(C)$. We have $ext(q') = ext(q)$, and hence, $ext(q') \neq \emptyset$.*

As the fresh variable does not occur in q, it is equivalent to ?. Therefore, q' is equivalent to q, and has the same extension. The benefits of this transformation is to give a name to an entity, so as to refer to it at another focus.

Lemma 28 *Assume $q' = q[and\ D]$ or $q' = q[cross\ D]$, with $D \in \mathcal{C}$, and $vars(D) = \emptyset$. We have $ext(q') = ext(q) \cap ext((\underline{D_{\phi''}}, \phi''))$.*

This lemma is very important because the transformations [*and D*] and [*cross D*] are the most numerous. Instead of computing the extension of q', it is possible to store the extensions of the complex classes $D \in \mathcal{C}$ that have no variable, and to check that they share instances with the current query. The cost of verifying a transformation is reduced to an intersection between two sets of resources. It is interesting to note that this is the same criteria as used for the selection of restriction values in faceted search (Section 2.2).

Lemma 29 *Assume $q' = q[t]$, where [t] is either of [or ?], [minus ?], [not], [delete and], [delete not]. We have $ext(q') \neq \emptyset$.*

In the last lemma, the five query transformations produce a graph pattern that is the same or more general than for q. This implies the extension of q' is equal or larger than for q, and hence is not empty.

To summarize, the transformations that require to compute the extension of q' are only the and-insertions of complex classes sharing variables with q, and the potential or-deletion. Other and-insertions need only the computation of a set intersection. All other transformations need no computation at all because they are necessarily safe, or related to an unsafe focus change. Therefore, the additional cost compared to standard faceted search, given a same vocabulary, is only the checking of the potential or-deletion, and the computation of the and-insertion of each variable occuring in the query, where the number of those variables is bounded by the number of cycles in the query, which is very low in practice.

6.4 From Theory to Camelis 2

From theoretical results presented in previous sections, we can now explain and justify the user interface of Camelis 2, as presented in Section 3. Given the definition of the navigation graph, the user interface is composed of the current query and a set of query transformations. The query determines the current selection, and the query transformations determine the available navigation links.

The query box (Q in Figure 2) displays the query as a complex class with an underlined part that indicates where the focus stands. A click at any position in the query box triggers a focus-change, unless it is the current focus that is clicked.

Other query transformations that do not require the selection of a complex class are provided as buttons at the right of the query box (QC in Figure 2). The button 'Name' provides the and-insertion of a fresh variable, which is automatically generated starting with `?X`, `?Y`, `?Z`, `?A`, etc. The button 'Not _' performs either a not-deletion or a not-insertion, whether the focus is under a negation or not. The button '_ Or ?' performs an or-insertion. The minus-insertion has been introduced during the writing of this paper, and will be implemented as an additional button '_ and not ?'. However, it can be decomposed into an and-insertion and a not-insertion. Finally, the button 'Delete' performs either an or-deletion or an and-deletion, whether the focus is a disjunction alternative or not.

The remaining query transformations, and-insertions and crossings, require the selection of a complex class that belongs to the vocabulary. A crossing is just a particular case of and-insertion, when the complex class is a restriction. We define the set of complex classes for which there is a safe and-insertion as the *index*. Its elements are called *features*.

Definition 30 (index) *Let $\mathcal{C}$ be a vocabulary, $\mathcal{R}$ be a RDF graph, and q be a LISQL query. The* index *of q in the vocabulary $\mathcal{C}$ and in the RDF graph $\mathcal{R}$ is defined as*

$$index_{(\mathcal{C},\mathcal{R})}(q) = \{D \mid (and\ D) \in ST_{(\mathcal{C},\mathcal{R})}(q)\}.$$

As traditionally done in faceted search, each feature is labeled by its *count*, i.e., the number of answers of the current query that match the feature. It is also the number of answers that would result from the application of an and-insertion with this feature. Because of safeness, count values are strictly positive integers.

Camelis 2 works with an infinite vocabulary, but the index is presented as an expandable tree so that only a finite subset of the vocabulary is used at any time. This allows for rich faceted summaries, while letting users in control of how much is displayed.

Definition 31 (Camelis 2 vocabulary) *In Camelis 2, the vocabulary is the set of complex classes that use none of the Boolean operators (*`and`*,* `or`*,* `not`*), and such that variables are only used as atomic complex classes.*

As in dynamic taxonomies and logical information systems (see Section 2.2), features can be organized into a generalization ordering by defining subsumption between them. Our definition is not based on logical inference as in OWL-DL, but merely on the computation of extensions of LISQL queries. Examples of subsumption relationships can be found in Section 3.

Definition 32 (subsumption) *Subsumption $\sqsubseteq$ between complex classes and between complex properties of the Camelis 2 vocabulary can be defined by the following inference rules:*

1. *if $r_1 = r_2$, then $r_1 \sqsubseteq r_2$;*
2. *if C is a complex class, then $C \sqsubseteq$* `?`*;*
3. *if $u_1 \in ext_{\mathcal{R}}($*`rdfs:subClassOf :` *$u_2)$,*
   then `a` *$u_1 \sqsubseteq$* `a` *u_2;*
4. *if $p_1 \in ext_{\mathcal{R}}($*`rdfs:subPropertyOf :` *$p_2)$,*
   then (p_1 `:` *$\sqsubseteq p_2$* `:`*), (p_1* `:` *$\sqsubseteq p_2$* `with`*), (p_1* `of` *$\sqsubseteq p_2$* `of`*), (p_1* `of` *$\sqsubseteq p_2$* `with`*), (p_1* `with` *$\sqsubseteq p_2$* `with`*);*

5. *if* $P_1 \sqsubseteq P_2$,
   then ($P_1 \sqsubseteq$ `trans` P_2), ($P_1 \sqsubseteq$ `opt` P_2), (`trans` $P_1 \sqsubseteq$ `trans` P_2), (`opt` $P_1 \sqsubseteq$ `opt` P_2);

6. *if* $P_1 \sqsubseteq P_2$ *and* $C_1 \sqsubseteq C_2$,
   then $P_1\ C_1 \sqsubseteq P_2\ C_2$;

7. *if* $r_1 \in ext_{\mathcal{R}}(\overline{\texttt{opt trans}\ P\ r_2})$,
   then $\overline{\texttt{opt trans}\ P\ r_1} \sqsubseteq \overline{\texttt{opt trans}\ P\ r_2}$.

Figure 2 shows how the index is dispatched between the extension box (E), the facet hierarchy (F), and the value boxes (V). The extension box contains all features that are resource names. Indeed, a resource r is a feature of a query q iff r is in the extension of q. Rule *(i)* accounts for the fact that the extension box is a flat list. Features in the form (P^+ r), where P^+ denotes a property chain, are grouped into value boxes. There is one value box for each expanded property chain. Rule (*vii*) accounts for taxonomies of values as visible in Figure 2 (recall that `in` = `opt trans part of`). All other features, those that do not contain any resource, are placed into the facet hierarchy. Rule *(ii)-(vi)* account for their hierarchy. Instead of displaying up to 12 complex properties for every basic property p, we only show (p `:`) and (p `of`), and possibly their reflexive and transitive closures when this produces taxonomies of values. Other closures are accessible by toggling each closure on/off and pushing the button 'Zoom' in the feature controls (FC). When double-clicking a feature in the form (P^+ ?), a sequence of crossings is performed on the properties of the property chain; otherwise, an and-insertion is performed.

7 Usability Evaluation

This section reports on the evaluation of semantic faceted search in terms of usability. we have measured the ability of users to answer questions of various complexities, as well as their response times. Results are strongly positive and demonstrate that semantic faceted search offers expressiveness and ease-of-use at the same time. There was no difficulty with the LISQL syntax. The main difficulty was about focus change, and other difficulties were about negation and inverse because of minor flaws in the user interface and in the tutorial. Details about the evaluation and results can be found on our website [5].

7.1 Participants

The subjects consisted of 20 persons, 15 males and 5 females. They were computer science graduate students (13 from the University of Rennes 1, 7 from the INSA engineering school). All participants use Internet daily and are familiar with information searching. They had prior knowledge neither of Camelis 2, nor of faceted search or semantic web. None was familiar with the dataset used in the evaluation and they had no particular interest in the concerned field.

7.2 Methodology

The evaluation was conducted in three phases : tutorial, test and feedback.

Tutorial. Before beginning the test, participants first had 20 minutes to learn how to use Camelis 2 with an oral tutorial, then they had 10 minutes to navigate freely and to ask questions. A paper tutorial, with the same information as the oral tutorial, was given to the participants. The tutorial detailed the interface and the actions, the genealogy vocabulary and an example of scenario that has been accomplished with the subjects. The scenario was a succession of queries that show, select, or count individuals according to their properties and relationships.

During the training, participants had a different knowledge base from the test, but both were about genealogy[6]. The knowledge bases were chosen such that all participants had at least some familiarity with the concepts.

Test. Participants were asked to answer a set of questions, consisting in listing or counting persons having some features and relationships. We recorded their queries and answers, and the time they spent on each question. Each question was written in French and in English. The test was composed of 18 questions and it was not limited in time.

Table 5 shows the questions and the minimum number of navigation steps necessary to answer them, when using the minimum vocabulary only (see Definition 23). The answer difficulty was smoothly increasing. Some questions had similar difficulty, they are grouped together in the same category. There are 7 categories: the first 2 categories are covered by standard faceted search, while the 5 other categories are not in general. The questions of a given category were not always presented successively. The first category, "Visualization", did not require the creation of a query. The exploration of the

[5] http://www.irisa.fr/LIS/alice.hermann/camelis2.html

[6] For the training, Benjamin Franklin's genealogy, and for the test, George Washingtons genealogy.

Category	Question	#navig. links
Visualization	1 How many persons are there?	0
	2 How many men are there?	0
	3 How many persons have a birth's place in the base?	0
Selection	4 How many women are named Mary?	4
	5 Who was born at Stone Edge?	4
	6 Which man was born in 1659?	5
	7 Who is married with Edward Dymoke [I33]?	3
Path	9 Which man has his father married with Alice Cooke [I30]?	5
	11 Which man is married with a woman born in 1708?	7
Disjunction	8 Which women have for mother Jane Butler [I67] or Mary Ball [I23]?	6
	12 Which men are married with a woman whose birth's place is Cuckfields or Stone Edge?	9
Negation	10 How many men were born in the 1600 or 1700 years, and not in Norfolk?	12
	13 How many women have a mother whose death's place is not Warner Hall?	7
Inverse	14 Who was born in the same place as Robert Washington [I64]?	6
	15 Who died during the year when Augustine Warner [I18] was born?	6
Cycle	16 Which persons died in the same area where they were born?	9
	17 How many persons have the same firstname as one of their parent?	8
	18 Which persons were born the same year as their spouse?	10

Table 5: Questions of the test, by category, and the minimum number of navigation links to answer them.

facet hierarchy was sufficient to lead to the answer. In the second category, "Selection", we asked to count objects that have a particular feature. In the third category, "Path", subjects had to follow a path of properties. We asked to select objects in relation with another object that has a particular feature. The fourth category, "Disjunction", required to use disjunction. We asked to select objects in relation with another object that have a feature A or a feature B. The fifth category, "Negation", required to use negation. We asked to count objects that do not have a particular feature. The sixth category, "Inverse", required to use the inverse of a property. We asked to select objects that have a feature that another known object have. In the seventh category, "Cycle", we asked for objects that have a feature shared by a related object. The three questions of this category required to use variables. The two first categories of questions, Visualization and Selection, are in the expressive scope of all faceted search systems, while the other categories are beyond most of them.

Questionnaire. After the test, each user filled in the SUS questionnaire [Bro96]. It contains 10 standardized questions on the user's feeling. The questions can be found on Table 6 together with the results. Each item ranges on a 0-4 scale from "strongly disagree" to "strongly agree".

We have added 5 open questions: 11) "What did you find especially difficult?", 12) "What did you find especially easy?", 13) "Which difficulty have you met?", 14) "What can be improved?", 15) "What should be kept unchanged?".

7.3 Results

A query could be correct while the answer was not. Indeed, a participant could forget to change the focus and answer related objects in place of the searched objects. Moreover, to find the correct answer, several queries were in general possible.

Figure 5 shows the number of correct queries and answers, the average time spent on each question and the number of participants who had a correct query for at least one question of each category:

Visualization : The first two questions had 20 correct answers and queries. The third question had 10 correct answers and 13 correct queries. All the 20 participants had a correct query for at least one question of the category. The average response times were respectively 43, 21, and 55 seconds.

Selection : The first question had 19 correct answers and 20 correct queries. The three other questions had 20 correct answers and queries. All the 20 participants had a correct query for at least one question of the category. The average response times were respectively 49, 77, 65, and 103 seconds.

Path : The first question had 14 correct answers and 15 correct queries. The second question had 18 correct answers and 19 correct queries. All the 20 participants had a correct query for at least one question of the category. The average response times were respectively 127 and 89 seconds.

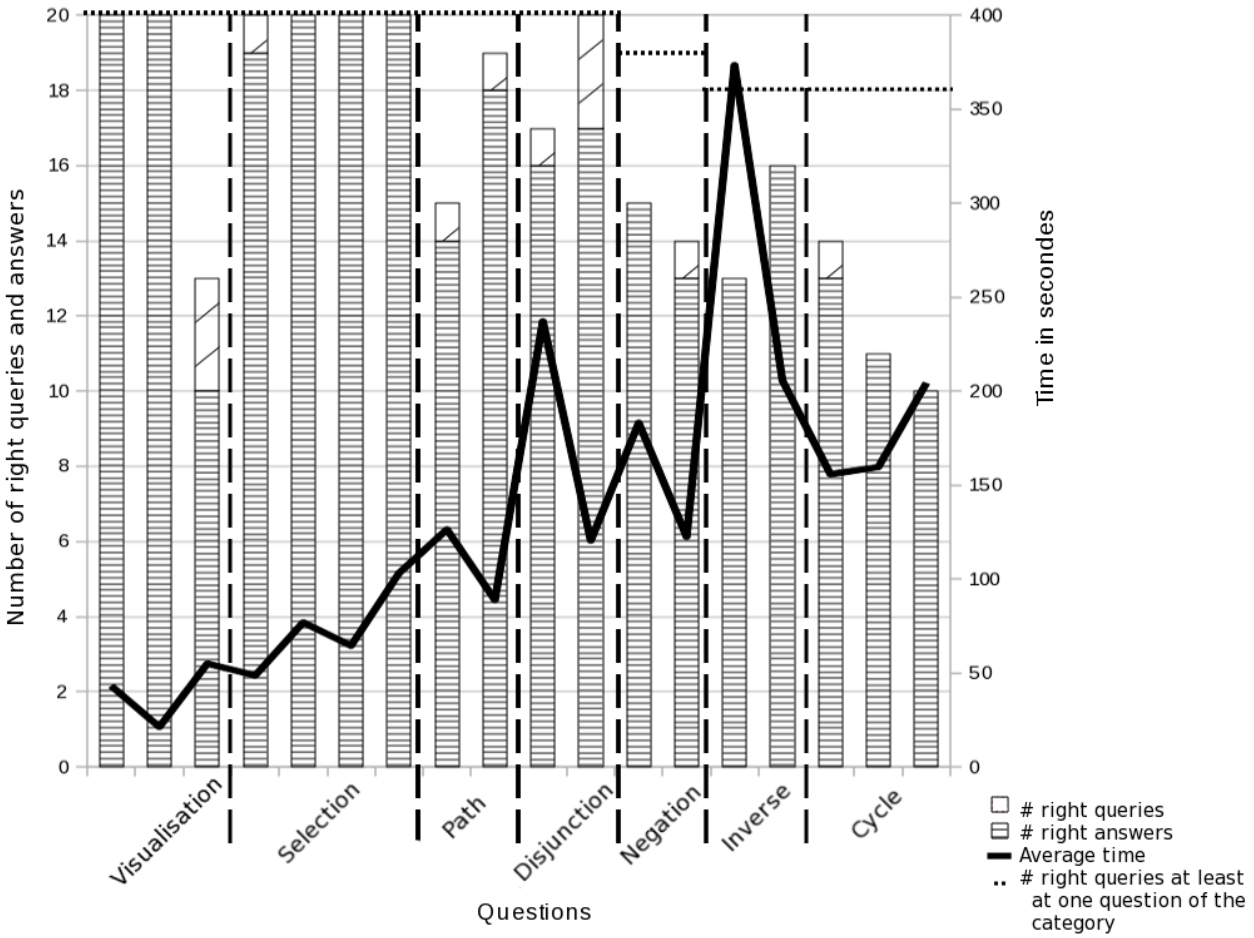

Figure 5: Average time and number of correct queries and answers for each question

Disjunction : The first question had 16 correct answers and 17 correct queries. The second question had 17 correct answers and 20 correct queries. All the 20 participants had a correct query for at least one question of the category. The average response times were respectively 237 and 120 seconds.

Negation : The first question had 15 correct answers and queries. The second question had 13 correct answers and 14 correct queries. 19 participants had a correct query for at least one question of the category. The average response times were respectively 183 and 122 seconds.

Inverse: The first question had 13 correct answers and queries. The second question had 16 correct answers and queries. 18 participants had a correct query for at least one question of the category. The average response times were respectively 373 and 206 seconds.

Cycle : The first question had 13 correct answers and 14 corrects queries. The second question had 11 correct answers and queries. The third question had 10 correct answers and queries. 18 participants had a correct query for at least one question of the category. The average response times were respectively 156, 160, and 204 seconds.

Figure 6 shows the distribution of subjects by number of correct answers and correct queries. The differences between the two distributions come from problems with the focus change. All subjects but one had correct answers to more than half of the questions. Half of the subjects had the correct answers at least to 15 questions out of 18. Two subjects answered correctly to 17 questions, their error was on a disjunction question for one and on a negation question for the other. All subjects had the correct query for at least 11 questions.

7.4 Interpretation

Test. For all questions, there is at least 50 percent of success. The subjects spent an average time of 40 minutes on the test, the quickest one spent 21 minutes and the slowest one 58 minutes.

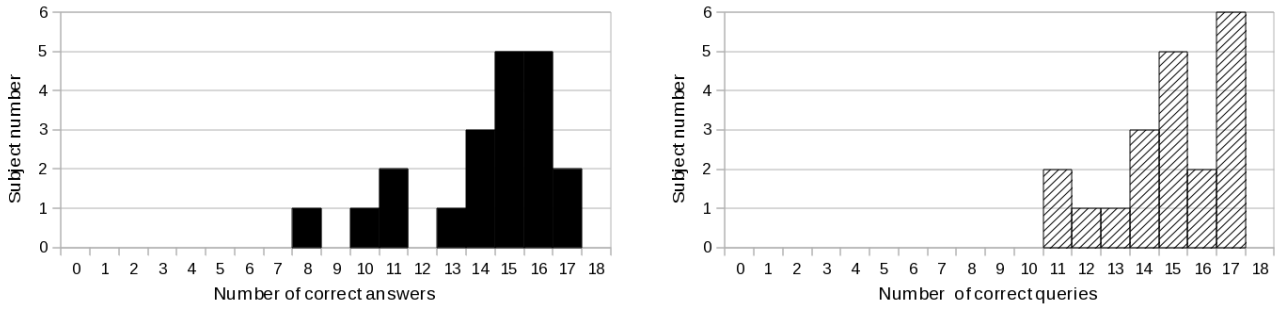

Figure 6: Distribution of subjects by number of correct answers, and by number of correct queries.

The first 2 categories corresponding to standard faceted search, visualization and selection, had a high success rate (between 94 and 100) except for the third question. The most likely explanation for the latter is that the previous question was so simple (`a man`) that subjects forgot to reset the query between the questions 2 and 3. All questions of the first two categories were answered in less than 1 minute and 43 seconds on average. Those results indicate that the more complex user interface of semantic faceted search does not entails a loss of usability compared to standard faceted search for the same tasks.

For other categories, there is a high diversity in the subject queries. Within each category, we observed that response times decreased, except for the "Cycle" category. At the same time, for "Path", "Disjunction" and "Inverse", the number of correct answers and queries increased. Those results suggest a quick learning process of the subjects.

For the negation, there is a small decrease of the number of correct answers. We think that this comes from a problem in the interface. Subjects checked on the NOT checkbox and double-clicked on the feature, expecting that it would have the same effect as clicking on the 'Zoom' button. This will be effective in the next version.

For the cycle questions, there are a lot of errors during navigation. We conjecture some lassitude at the end of the test because the three cycle questions have similar complexity and all but two subjects answered correctly to at least one of these questions.

There is a peak of response time for the first "Inverse" question. We realized afterwards that there were no related examples in the tutorial. It is noticeable that subjects, nevertheless, managed to solve the questions with a reasonable success rate. Furthermore the second question was much quicker.

The focus change problem was important for the first questions and the last ones. Between questions 5 and 12, no one failed except for a single person who regularly did.

SUS Question	Score (on a 0-4 scale)	
I think that I would like to use this system frequently	2.8	Agree
I found the system unnecessarily complex	0.8	Strongly disagree
I thought the system was easy to use	2.6	Agree
I think that I would need the support of a technical person to be able to use this system	1.5	Disagree
I found the various functions in this system were well integrated	2.9	Agree
I thought there was too much inconsistency in this system	0.6	Strongly disagree
I would imagine that most people would learn to use this system very quickly	2.5	Agree
I found the system very cumbersome to use	1.0	Disagree
I felt very confident using the system	2.8	Agree
I needed to learn a lot of things before I could get going with this system	1.7	Neutral

Table 6: Results of SUS questions.

SUS and open questions. Table 6 shows the answers to the SUS questions. Subjects agree that there is a need for teaching, however results show that they quickly learned, even by themselves for one category. They find the system complex but without useless functions and without inconsistency. People found all questions before the 14^{th} one easy and the questions 14 to 18 difficult with cycles and inverse properties. Some questions were found harder, but nobody consulted the paper tutorial. People found the navigation and the query language easy. They said that they had a problem regarding the focus change, in particular to know which part of the query they should select to modify the focus. As improvements, people wished a better integration of the negation with the double-click. Subjects wished to keep the navigation and the query box.

They also asked for some functions that are present in Camelis 2 but were not presented during the tutorial session (e.g., query edition).

8 Related Work

We discuss other approaches for applying and extending faceted search to the Semantic Web. We also compare the expressiveness of LISQL with two expressive query languages of the Semantic Web: SPARQL and SPARQL-DL.

8.1 Faceted Search for the Semantic Web

As faceted search is becoming widespread, a number of proposals have been made to apply it on the Semantic Web (SW). They all have in common to assume that data is represented in a SW format, either RDF(S) or OWL. Most of them, such as Ontogator [MHS06], mSpace[7], and Longwell[8], do not claim for a contribution in term of expressiveness, and contribute either to the design of better interfaces and visualizations, or to methods for the rapid or user-centric configuration of faceted views [SVH07]. Therefore, their contributions are somewhat orthogonal to ours, and could certainly complement ours. Other approaches, such as SlashFacet [HvOH06] and BrowseRDF [ODD06], extend faceted search towards a more expressive navigation.

The most essential ingredient for an expressive and flexible semantic search in RDF graphs is *focus change*. It allows to change the perspective without changing the underlying graph pattern. To the best of our knowledge, no faceted search system offers this in a general way. SlashFacet has the *crossing* operation that selects the images of the items in the current selection through a property. Crossing includes a focus change, but crossing back a property is not equivalent to a focus change, because it introduces an additional restriction: starting from C and crossing p `:` `?` and then p `of` `?` leads to p `:` p `of` C instead of C `and` p `:` `?` (they are not equivalent). Other systems allow to focus on different types of items, but this focus cannot be changed in the course of a search. For example, in a dataset about publications, a choice has to be made between authors and documents.

It is generally considered that the query should be hidden from the interface. In fact, in most faceted search systems, the query *is* displayed as the list of the restriction values users have already selected in the course of their search. This is important so that users do not feel lost, and can easily reverse previous selections. On our case, the query is also important to specify focus changes. Of course, displaying the query in SQL would ruin those benefits: the display of the query is part of the design of the user interface. Now, when the expressiveness is raised to SPARQL with graph patterns, disjunction, and negation, it becomes necessary to introduce syntax. While, in Camelis 2, the query is simply rendered as a sentence following some grammar, nothing prevents to render syntax through graphical widgets (e.g., lists for conjunction, trees for restrictions, tab panels for disjunction). In our approach, SPARQL is used behind the scene, and LISQL is used to render the query in a way that fits semantic faceted search (see Section 4).

Disjunction and negation are either absent or strongly limited in existing approaches. Disjunction is restricted to build sets of values or sets of items, e.g., in SlashFacet. Negation is restricted to restriction values, and also applies to unqualified restrictions (`not` P `?`) in BrowseRDF. No other system allows to form cycles as we do with variables.

The value boxes of SlashFacet can handle only one taxonomy of values, whereas we can use the reflexive and transitive closure of any property that link the values together. For instance, when values are persons, we can use either `parent :` (descendancy chart), or `parent of` (ancestry chart).

8.2 Query Languages for the Semantic Web

We compare our query language LISQL to SPARQL, as the reference query language for the Semantic Web, and to SPARQL-DL [SP07] for the syntactic similarity of complex classes and complex properties with LISQL.

8.2.1 Comparison with SPARQL

Haase *et al.* [HBEV04] define a set of 14 use cases for comparing the expressiveness of RDF query languages. We use them to evaluate and compare the expressiveness of SPARQL and LISQL. First, a significant difference is that LISQL has mono-dimensional queries, i.e., LISQL queries are translated to SPARQL queries having a single variable after SELECT. This constraint comes from the nature of faceted search, not from LISQL itself as several foci could be selected to have several variables after SELECT. The facet hierarchy, the value boxes, and a highlighting mechanism compensate for this constraint. Assume users want to know who is the mother of each male Washington. They first navigate to the query `a man and lastname : Washington`. Then, they expand the facet `mother : ?` in the facet hierarchy, which opens a value box that

[7] see `http://mspace.fm/`

[8] see `http://simile.mit.edu/wiki/Longwell`

lists the mothers of male Washingtons, and for each mother, tells how many children she has among them. The associations between male Washingtons and their mothers are accessible by a dynamic highlighting mechanism. When selecting a male Washington (in the extension box), his mother is highlighted in the value box. Symmetrically, when a mother is selected in the value box, her children are highlighted in the extension box.

The use cases SPARQL and LISQL have in common are path expressions (e.g., "the name of the author of some publication X"), union/disjunction, partial support for collections and containers, support for literals, and entailment through class and property hierarchies. Compared to SPARQL, LISQL has not the OPTIONAL construct because it is useless in one-dimensional queries. The difference/negation use case is covered in extensions of SPARQL with the operator MINUS of Angles and Gutierrez [AG08], or the operator NOT EXISTS of SPARQL 1.1. The recursion use case (transitive closure in LISQL) is covered in nSPARQL [PAG08], an extension of SPARQL with *nested regular expressions*. LISQL supports restricted forms of aggregation and grouping, but enough to cover the aggregation use case. The restriction is that only the count aggregator can be used, and grouping can be done only along one dimension. For example, Figure 2 shows in the value box (V) "the number of male Washingtons by birthplace". The reification use case is covered by SPARQL: e.g., "the person who has classified the publication X". As defined in Section 4, LISQL does not cover it, but its implementation in Camelis 2 does. The LISQL query for the previous example is (`a publication and ?X and topic (classifier : ?) : ?`), where the complex class into brackets after `topic` put a constraint on the reified triple whose predicate is `topic`. This complex class can be navigated to, like other complex classes.

In total, SPARQL scores 9.5/14, LISQL scores 8/14, as defined in Section 4, and scores 10/14, as implemented in Camelis 2. In fact, SPARQL and LISQL have a similar expressiveness, and most differences can be removed by extending either language: adding difference/negation and recursion to SPARQL; adding multiple foci and optional pattern to LISQL.

8.2.2 Comparison with SPARQL-DL

Syntactically, LISQL complex classes are similar to complex classes as defined in OWL-DL. This suggests that SPARQL-DL [SP07] could be used instead of SPARQL to translate from the LISQL syntax. However, this is not possible because SPARQL-DL is restricted to conjunctive queries, and variables cannot occur in complex classes. On one hand, a LISQL query that contains disjunctions and negations but no variables (hence no cycles) and the default focus, can be translated to a SPARQL-DL query in the form `Type(?x,`C`)`, where C is a complex class that has the same abstract syntax as the LISQL query. For example, the LISQL query `a man and birth : (year : (1601 or 1649) and place : not in England)` can be translated to

```
Type(?x, and(
   man,
   some(birth, and(
     some(year, or(\{1601\},\{1649\})),
     some(place, not(some(in, \{England\})))
))))).
```

On the other hand, a LISQL query that contains variables but neither disjunction nor negation, can be translated in a similar way to SPARQL, using in fact the common subset between SPARQL and SPARQL-DL. For example, the LISQL query `a man and father : ?X and mother : spouse of ?X` can be translated to

```
Type(?z,man),
PropertyValue(?z,father,?x),
PropertyValue(?z,mother,?y),
PropertyValue(?x,spouse,?y).
```

The two kinds of translations cannot be reconciled in the general case, in particular when variables occur in the scope of negation of disjunction.

In fact, SPARQL-DL and LISQL work at different level, and might complement each other by benefiting from a comparable syntax. SPARQL-DL, like OWL-DL, works at the *intentional* level, whereas LISQL and SPARQL work at the *extensional* level. The intentional level is associated to open world assumption, and ontological reasoning. The extensional level is associated to closed world assumption, and query answering over a unique and finite interpretation, namely a RDF graph.

9 Conclusion

We have introduced *semantic faceted search* as a search paradigm for Semantic Web knowledge bases, in particular RDF graphs. It combines the expressiveness of the SPARQL query language, and the benefits of exploratory search and faceted search. Exploratory search is formalized as a graph, where nodes are queries, and edges are query transformations. The

navigation graph is proved to be safe, because whatever the path of query transformations, the set of answers of the current query is never empty. It is proved locally-finite, because the number of available query transformations is always finite. It is also proved complete w.r.t. the query language, because for every query, there is a navigation path that leads to it, unless this query is not safe. Finally, it is as efficient as standard faceted search w.r.t. the computation of facets and restriction values. The completeness proof is the key result here because it draws an equivalence between expressive querying and exploratory search, therefore totally liberating users from editing queries, even the most complex ones.

The user interface of semantic faceted search includes the user interface of other faceted search systems, and can be used as such. It adds a query box to tell users where they are in their search, and to allow them to change the focus or to remove query parts. It also adds a few controls for applying some query transformations such as insertion/deletion of disjunction, negation, and variables. We have introduced a new query syntax, LISQL, to better fit with a faceted interface and query transformations. The LISQL syntax can be seen as an extension of either Turtle or OWL-DL complex classes. Beside the list of selected items, the user interface has a hierarchy of facets organizing classes and properties by subsumption, and value boxes that can be displayed as flat lists or as various taxonomies automatically derived from the dataset.

Semantic faceted search has been implemented as a prototype, Camelis 2. Its usability has been demonstrated through a user study, where, after a short training, all subjects were able to answer simple questions, and most of them were able to answer complex questions involving disjunction, negation, or cycles. This means semantic faceted search retains the ease-of-use of other faceted search systems, and reaches the expressiveness of query languages such as SPARQL.

We think that semantic faceted search is not tied to LISQL/SPARQL, and can be adapted to other query languages. Indeed, the definition of a navigation graph only requires the definition of the extension of a query, i.e., its set of answers, and the definition of query transformations. The hard part is then to prove that the resulting navigation graph is safe, locally-finite, and complete, which we have successfully done here for LISQL/SPARQL.

Acknowledgments. We would like to thank the 20 students, from the University of Rennes 1 and the INSA engineering school, for their volunteer participation to the usability evaluation.

References

[AG08] R. Angles and C. Gutierrez. The expressive power of SPARQL. In A. P. Sheth *et al*, editor, *Int. Semantic Web Conf.*, LNCS 5318, pages 114–129. Springer, 2008.

[BCM+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BKK05] A. Bernstein, E. Kaufmann, and C. Kaiser. Querying the semantic web with Ginseng: A guided input natural language search engine. In *Work. Information Technology and Systems (WITS)*, 2005.

[Bro96] J. Brooke. SUS: A quick and dirty usability scale. In P. Jordan, B. Thomas, B. Weerdmeester, and A. McClelland, editors, *Usability evaluation in industry*, pages 189–194. London: Taylor and Francis, 1996.

[Fer09] S. Ferré. Camelis: a logical information system to organize and browse a collection of documents. *Int. J. General Systems*, 38(4), 2009.

[FHH04] R. Fikes, P. J. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the semantic web. *J. Web Semantic*, 2(1):19–29, 2004.

[FR00] S. Ferré and O. Ridoux. A file system based on concept analysis. In Y. Sagiv, editor, *Int. Conf. Rules and Objects in Databases*, LNCS 1861, pages 1033–1047. Springer, 2000.

[FR04] S. Ferré and O. Ridoux. An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419, 2004.

[FR07] S. Ferré and O. Ridoux. Logical information systems: from taxonomies to logics. In *Int. Work. Dynamic Taxonomies and Faceted Search (FIND)*, pages 212–216. IEEE Computer Society, 2007.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations*. Springer, 1999.

[HBEV04] P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A comparison of RDF query languages. In S.A. McIlraith *et al.*, editor, *Int. Semantic Web Conf.*, LNCS 3298, pages 502–517. Springer, 2004.

[HEE+02] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Communications of the ACM*, 45(9):42–49, 2002.

[HKR09] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[HvOH06] M. Hildebrand, J. van Ossenbruggen, and L. Hardman. /facet: A browser for heterogeneous semantic web repositories. In I. Cruz *et al*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 272–285. Springer, 2006.

[Mar06] G. Marchionini. Exploratory search: from finding to understanding. *Communications of the ACM*, 49(4):41–46, 2006.

[MHS06] E. Mäkelä, E. Hyvönen, and S. Saarela. Ontogator - a semantic view-based search engine service for web applications. In I. F. Cruz *et al.*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 847–860. Springer, 2006.

[ODD06] E. Oren, R. Delbru, and S. Decker. Extending faceted navigation to RDF data. In I. Cruz *et al*, editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 559–572. Springer, 2006.

[PAG08] J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. In A. P. Sheth *et al*, editor, *Int. Semantic Web Conf.*, LNCS 5318, pages 66–81. Springer, 2008.

[Sac00] G. M. Sacco. Dynamic taxonomies: A model for large information bases. *IEEE Transactions Knowledge and Data Engineering*, 12(3):468–479, 2000.

[Sac06] G. M. Sacco. Some research results in dynamic taxonomy and faceted search systems. In *Faceted Search Work. at ACM SIGIR 2006*. ACM, 2006.

[SP07] E. Sirin and B. Parsia. SPARQL-DL: SPARQL query for OWL-DL. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *Work. OWL Experiences and Directions (OWLED)*, volume 258. CEUR-WS, 2007.

[ST09] G. M. Sacco and Y. Tzitzikas, editors. *Dynamic taxonomies and faceted search*. The information retrieval series. Springer, 2009.

[SVH07] O. Suominen, K. Viljanen, and E. Hyvönen. User-centric faceted search for semantic portals. In E. Franconi, M. Kifer, and W. May, editors, *Eu. Semantic Web Conf.*, LNCS 4519, pages 356–370. Springer, 2007.

[vR86] C. J. van Rijsbergen. A new theoretical framework for information retrieval. In *Int. ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 194–200. ACM, 1986.