Reconciling Expressivity and Usability in Information Access

Sébastien Ferré

Habilitation Thesis

University Rennes 1

Reconciling Expressivity and Usability in Information Access

from File Systems to the Semantic Web

Sébastien Ferré

Associate Professor
Team LIS, IRISA, University Rennes 1
ferre@irisa.fr
http://www.irisa.fr/LIS/ferre/

Defended on November 6th, 2014
in front of the defense committee, composed of:

Olivier Pivert	Professor at Univ. Rennes 1 / *president*
Norbert E. Fuchs	Senior researcher at Univ. Zurich / *referee*
Marianne Huchard	Professor at Univ. Montpellier 2 / *referee*
Eero Hyvönen	Professor at Univ. Aalto / *referee*
Karell Bertet	Associate professor (HDR) at Univ. La Rochelle / *examiner*
Fabien Gandon	Researcher (HDR) at INRIA Valbonne / *examiner*

To Sterenn, Yann, and Eléonore

Foreword

Since the start of my PhD in 1999, I have worked on various problems, and produced a number of different contributions pertaining to several domains and research communities. This habilitation thesis is an opportunity for me to summarize them, and to open perspectives. However, that synthesis work is made difficult by the fact that the set of my research tracks look more like a bush than like a tree. In fact, I see the research process as an evolution process where ideas, theories, softwares, and applications compete for survival, and are transformed by mutation and crossover. Some of my research tracks have gone extinct (e.g., logic functors), while others have flourished (e.g., Semantic Web). Some have appeared by mutation (e.g., guided edition from guided search), while others have merged (e.g., conceptual navigation and faceted search). At every time, there is not a single active research track, nor a "best" research track, but a population of diverse and complementary research tracks. That evolution process makes it hard to synthesize the different research results into one "all-inclusive" result. First, merging different existing results in a coherent way is a research process in itself, and a never-ending process. Second, the result of that merging would be a static snapshot hiding the evolution process, and hence hiding both the past and the future of the research activity, and that snapshot would soon become outdated. An alternative approach would be to write a *history* of research tracks and results, in chronological order. However, people generally do not want to revive the past, which tends to be boring, but rather to know the past in order to understand the present, and envision the future.

What I therefore chose to do is to write a synthesis of my research work over the last 15 years such that it is rooted in the present, and yet, shows the evolution process from past to future. To this purpose, I "datamined" my different research results, and discovered a pattern that provides an unifying framework for a number of quite different results: Abstract Conceptual Navigation (ACN). Each result is presented as an instance of the ACN framework, without trying to reformulate the original result, only using ACN as a point of view over it. The presentation of successive results through a

same framework makes it easier to compare them, and therefore to highlight progress over time. As an unifying framework, ACN helps to understand the specificity of my research work, and what drived it from file systems to the Semantic Web. The generality of ACN also opens the room for imagination, and helps suggest and structure a number of perspectives. ACN is not a "hook in the sky", and builds on strong information access paradigms: query languages, navigation structures, and interactive views. In fact, ACN is a proposal for a synthesis of those three information access paradigms.

This habilitation thesis is made of a synthesis chapter, and four selected papers published in international journals as appendices. The selected papers represent important milestones in my research work, and are shown exactly as they were published. The synthesis is centered around Abstract Conceptual Navigation (ACN), and is made of three parts: state-of-the-art, main contributions, and perspectives. The state-of-the-art is an overview over existing approaches to information access for structured data. It tries to be complete at a coarse level of detail, but it ignores many details and specificities of existing approaches. This is justified, I think, by the objective to introduce and motivate ACN, which is itself at a high level of abstraction. A more detailed account of related work can be found in published papers. The ACN framework is then introduced, and used as a basis for the presentation of the main contributions of my work. Secondary contributions are associated to their closest main contribution, and only briefly presented. Finally, the main contributions are summarized, and ACN is used again as a basis for discussing a number of perspectives. In the synthesis, self-references are distinguished by a boldface font. Each appendix paper has its own reference list.

Even though I use the first-person singular in this foreword because I here express my point of view, most of *my* work is *our* work. First of all, all this work would not even exist without the original ideas of Olivier Ridoux in the late nineties, and the opportunity he gave me to work on them. Then, I was lucky to work with many colleagues and students, who largely contributed to the "population growth" of my research tracks. This is why the use of the first-person plural in the synthesis is not only for scientific tradition, but also an acknowledgement of their contributions on many results. I will not try and list all of them here, but the reader will see them all along the synthesis, and as co-authors in self-references.

Contents

Synthesis

1 Introduction

In many domains where information access plays a central role, there is a gap between expert users who can ask complex questions through formal query languages, and lay users who either are dependent on expert users, or must restrict themselves to ask simpler questions. In everyday practice, tools like folder trees, search engines, and spreadsheets are far more often used than databases and semantic knowledge bases (formal information systems), even though the later tools offer much more expressivity and precision in search. In fact, formal information systems are so tedious to use that even expert users do not generally use them for their everyday activities. When a database is used, only a limited number of pre-defined queries are generally made available to end users through interfaces. We think this is because there are two bottlenecks in the use of formal information systems. The first bottleneck is the input of data, which requires the formal representation of data. The second bottleneck is the expression of formal queries (e.g., in SQL). The common point between those two bottlenecks is the use of formal languages, an update language for the former, a query language for the latter. Because of the formal nature of those languages, there seems to be an unescapable trade-off between expressivity and usability in information systems. The objective of this synthesis is to present a number of results and perspectives that show that the expressivity of formal languages can be reconciled with the usability of widespread information systems. The final aim of this work is to empower people with the capability to produce, explore, and analyze their data in a formal way.

We focus on formal languages and systems because they are key to a high expressivity and precision. Therefore we restrict the scope of our work to structured and semantic information, i.e. information representations that are amenable to (logical) reasoning. This includes relational databases (RDB) [Codd, 1970], spreadsheets, classification hierarchies, and the Semantic Web (SW) [Berners-Lee et al., 2001, Hitzler et al., 2009]. This

a priori excludes unstructured texts, images and other multimedia objects as those would be seen as mere strings and bitmaps in our approach. However, a large number of works aim at automatically extracting structured and semantic information from such contents (e.g., natural language processing, clustering/classification, segmentation). A notable example is DBpedia [Lehmann et al., 2013], a Semantic Web version of Wikipedia that extracts and formally represents about 2 billions facts. More generally, the extension of the Web of documents with a Web of data, which includes the Semantic Web, means that more and more information is available in structured formats, ranging from simple CSV files to Linked Open Data (LOD)[1]. Because of its structured and semantic nature, its openness, and its support by the W3C, the Semantic Web is the playground of choice for our work, even though similar results could have been obtained on the more established relational databases.

Our work focuses on two important properties of information access: expressivity and usability. We here propose idealized definitions of them whose purpose is more to clarify their meaning than to drive practical evaluations. *Expressivity* is a measure of how many and which questions can be answered by a system. The set of expressible questions is generally defined as a query language, which helps when comparing the expressivity of different systems. A system A is more expressive than a system B if every B-question can be translated into an *equivalent* A-question. This definition shows the importance of semantics in the definition of query languages, as it is necessary to define query equivalence. All of this can also be said for update languages, which define the set of expressible statements. SPARQL [Angles and Gutierrez, 2008, SPARQL11, 2012] is the reference query and update language for the Semantic Web, and plays the same role as SQL for relational databases. Both SPARQL and SQL are recognized as very expressive, and are therefore used as a gauge for evaluating and comparing existing systems, including ours.

Usability is a measure of how many and which people can use effectively a system. Because of human factors, it is more difficult to define usability than to define expressivity. We propose to define that a system A is more usable (for some task) than a system B if every user that can perform the task in system B can also perform it in system A. This definition implies that systems can only be compared on their common tasks. When system A is more expressive than system B, it supports tasks that are not supported by system B, and is therefore "more usable" on those tasks by definition. However, "more usable than impossible" is not very informative as only a few

[1] http://www.w3.org/DesignIssues/LinkedData.html

people may be able to leverage the additional expressivity. A way to compare them is then to evaluate the proportion of B-users (defined as all persons that can perform all tasks supported by B) that can effectively perform the additional tasks supported by system A. Beyond the objective measure of performance (success rate and time), the subjective experience of users is also important when comparing different systems. For example, system properties such as response times, reliability, understandability of results, serendipity strongly affect the user experience.

Since the beginning of my PhD (1999), we have proposed a number of theories and implementations to better reconcile expressivity and usability, and applied them to a number of contexts going from file systems to the Semantic Web. In this synthesis, we introduce a unifying framework to factor out the main ideas of all those results: Abstract Conceptual Navigation (ACN). The principle of ACN is to guide users by letting them *navigate* in a conceptual space where places are *concepts* connected by navigation links. Concepts are characterized by a formal expression (e.g., a query, an update), and are made of two parts: an *extension* and an *intension*. The extension is made of concrete objects (e.g., entities, values) while the intension is made of formal expressions. The conceptual space is not static but is induced by concrete data, and evolves with it. Navigation promotes usability by guiding users, and freeing them from the burden of writing formal expressions. Because those formal expressions are generated through navigation, instead of being parsed from user input, it is easier to improve their display and understandability. The formal characterization of concepts promotes expressivity because the conceptual space results from the combination of a formal language and a dataset. Therefore, ACN can be made more expressive by using a more expressive language. ACN is *safe* if every navigation path avoids unsafe places (e.g., empty results). ACN is *complete* if there is a navigation path to every safe place. ACN is abstract in the same way an abstract class is in Java: it only defines a structure to be filled with a concrete definition of the conceptual space. Our theories and implementations qualify as instances of ACN, reaching increasing degrees of expressivity and usability, and covering different kinds of tasks.

This synthesis is made of three parts. Section 2 shortly presents the different kinds of solutions that have been proposed for information access. Those solutions are organized into three main categories, and compared to each other according to a number of criteria. Section 3 organizes and presents most of our contributions into a unifying framework: Abstract Conceptual Navigation (ACN). ACN is a contribution of this thesis, and helps at the same time to understand past work (*What they have in common? How do they compare?*), and to sketch future work. Section 4 summarizes our

core contribution to information access, i.e. the reconciliation of expressivity and usability. It also discusses less addressed properties such as *readability* and scalability. Finally, it opens a number of theoretical and applicative perspectives for future research.

2 State of the Art

In this section, we try and cover the different kinds of existing solutions to access structured information. Given the importance of information access as a research problem, a huge number of works has been done, and is still being done. We only cite a few of them, but after more than ten years working on this problem, we are pretty confident that the vast majority of existing works fall in at least one of the categories that we have identified. Those categories are:

- query languages: e.g., SQL, SPARQL, Natural Language Interfaces (NLI),
- navigation structures: e.g., file systems, hypertext,
- interactive views: e.g., faceted search, OLAP.

In the following, we define each category, and illustrate them with concrete and well-known approaches that fit in the category. We evaluate each approach w.r.t. our objective to reconcile expressivity and usability. At a finer level, we use the following properties in our evaluations and comparisons:

expressivity: What range of questions can be answered?

reliability: What degree of confidence can be put on answers?

readability: How easy is it for users to read and understand questions, answers, and user interface controls?

guidance: Whether users are guided in the expression of their user needs?

safeness: Whether guidance prevents users to fall in dead-ends?

specificity: Whether guidance is made specific to actual data?

scalability: What amount of data can be accessed?

The complexity of usability is reflected by its decomposition into reliability, readability, guidance, safeness, and specificity (this list is certainly not exhaustive). In practice, those properties are partially entangled, so that it may be difficult to evaluate each property independently. For example, a limited guidance may be the cause or the consequence of a limited expressivity.

2.1 Query Languages

This category includes all systems in which the user composes a query, and the system returns query answers. They rely on the definition of a query language (syntax), a domain of answers, and a mapping from queries to answers (semantics). The query language can range from simple (e.g., keywords) to complex (e.g., SQL); and from formal (e.g., SQL) to natural (e.g., English). Answers are most often sets of objects, but can also be single values (e.g., query answering [Lopez et al., 2011]) or tables (e.g., SQL). Query languages can be classified in three classes, even though there is a continuum between them [Kaufmann and Bernstein, 2010]: formal languages, controlled natural languages, and natural languages.

Formal languages. An archetypal formal query language is SQL for querying relational databases. Other examples are SPARQL for querying (and modifying) RDF graphs [SPARQL11, 2012], and XQuery for querying and processing XML trees [XQuery3, 2013]. Formal languages are based on logic, relational algebra, and structural matching. Their main advantages are a high expressivity and reliability, and are only superseded on expressivity by programming languages. Decades of research on the optimization and execution of formal queries (in particular SQL) has led to highly scalable solutions. However, the response time can vary drastically between different queries, depending on their complexity. The main drawback of formal languages is their formal syntax that exhibits low-level notions such as logic and relational algebra. It takes a trained computer scientist to write SQL or SPARQL queries. It is important to note that the difficulty is not only about learning a particular syntax, but also about learning the data schema, and understanding the low-level and technical notions.

Natural languages. The use of natural languages as query languages is attractive because everybody masters at least one natural language since youth, and because natural languages are very expressive. This gives the promise of direct and natural information access. Unfortunately, despite a vast amount of work in the past decades [Lopez et al., 2011], existing solutions are gen-

erally either specific to a limited domain, or unreliable. In order to improve reliability, systems are generally designed and trained for a restricted domain, and for a restricted set of questions. Therefore, despite the high expressivity of natural language, natural language interfaces (NLI) offer a limited expressivity, generally much lower than that of expressive formal languages. In the Semantic Web, most systems (e.g., PowerAqua [Lopez et al., 2012], QAKiS [Cabrio et al., 2012], FREyA [Damljanovic et al., 2010]) generate only Basic Graph Patterns (BGP) [SPARQL11, 2012] with at most a few triple patterns.

Controlled natural languages. Controlled natural languages (CNL) have been introduced to eliminate or reduce the ambiguity problem of natural languages. They have been used to improve communications among humans (e.g., Caterpillar Fundamental English [Verbeke, 1973]), or to provide natural notations for formal languages (e.g., Attempto Controlled English (ACE)[2] [Fuchs et al., 1999, Fuchs et al., 2006]). According to Kuhn [Kuhn, 2013], *"A controlled natural language is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax and/or semantics while preserving most of its natural properties"*. The main advantage of CNLs is to regain the expressivity and reliability of formal languages, while retaining the readability of natural languages. An analogy can be made with high-level programming languages, which offer the same expressivity than machine language by translating high-level constructs into low-level instructions (e.g., translating a loop into a combination of tests and jumps). While CNLs are generally designed to be readable without training, it is generally more difficult to write correct sentences, similarly to formal languages. Indeed, precise syntax and semantic conventions have to be followed so that machine understanding agrees with user's intended meaning. Most CNLs are designed for the expression of statements and rules, and few of them for the expression of queries [Kuhn, 2013].

Discussion. A common weakness of all query language approaches is the lack of user *guidance*. Even though it is arguably simpler to use natural languages than formal languages, formulating information needs as queries is in all cases a difficult task. Even if users master the syntax of the query language, they may not know the relevant vocabulary. Even if they know the vocabulary, nothing prevents them to get empty results, which is frustrating for users and force them to search by trial-and-error. Flexible querying approaches mitigate those difficulties by returning not only exact answers,

[2] `http://attempto.ifi.uzh.ch/site/`

but also approximate answers. Approximation can be based, for instance, on semantic similarity [Corby et al., 2006, Hurtado et al., 2008] or on fuzzy logic [Bosc and Pivert, 1995]. However, it remains necessary to have some knowledge of the vocabulary, and users have generally little control on the approximation process.

Moreover, users may not have a precise idea of what they are looking for, and need to get some feedback from the system in order to refine their information needs. Blank entry fields, the usual user interface for query language approaches, can trigger a "writer's block" on users. Indeed, the lack of feedback makes it hard for users to predict the system behaviour, and delays the detection of errors well after they have been entered. Query forms, query builders, and auto-completion are improvements to the entry field that offer some level of *guidance*. Query forms prevent syntax errors by using different fields and menus, but *expressivity* is limited to the simplest queries or to canned queries. Query builders (e.g., Semantic Crystal [Kaufmann and Bernstein, 2010], a query constructor for fuzzy databases [Smits et al., 2013a]) allow for the graphical construction of formal queries. They also prevent syntax errors, and provide more *expressivity* than forms, but they do not entirely solve the problem of *readability* and *understandability* because the graphical representations remain close to formal languages. They make it easier for specialists to build syntactically correct queries, but not much easier for non-specialists. Auto-completion can also be used to avoid syntax errors (e.g., Ginseng [Bernstein et al., 2005], Aggrego Search [Smits et al., 2013b]), or even to account for semantic relationships between terms and concepts [Hyvönen and Mäkelä, 2006], but it is mostly used to avoid vocabulary errors (e.g., typos, synonyms). However, those improvements are not sufficient to provide a fine-grained *guidance* at both lexical and syntactic levels, i.e., a *guidance* avoiding empty results (*safeness*) and giving feedback about the actual contents of the dataset (*specificity*).

2.2 Navigation Structures

This category includes all systems in which users navigate from place to place by following navigation links (e.g., file systems). They rely on the definition of a navigation graph where nodes are navigation places (e.g., folders), and edges are navigation links (e.g., links to sub-folders). At each navigation step, the user interface shows some contents associated to the current navigation place (e.g., the local files), and links to neighbour places (e.g., the sub-folders). Those systems can be classified in (at least) three classes, according to the shape of their navigation structure: graphs, hierarchies, and concept lattices. In order to formally compare navigation structures to

query languages, we evaluate the expressivity of each category by analyzing the kind of questions that can be answered through navigation.

Graphs. The historic example of a graph used as a navigation structure is the hypertext that structures the World Wide Web (WWW) [Berners-Lee et al., 1992], and maybe the Memex of V. Bush [Bush, 1945]. In WWW's navigation structure, navigation places are Web pages, and navigation links are hypertext links. A more recent example is the Semantic Web [Hitzler et al., 2009], whose RDF graphs can be used as navigation structures: navigation places are here entities (e.g., people, locations, organizations), and navigation links are semantic relationships between entities (e.g., "has parent", "is an employee of", "was born in"). When RDF graphs are published according to the Linked Open Data (LOD) principles, the Semantic Web can be navigated with a standard Web browser[3]. Questions are limited to starting from a particular place, and following a number of navigation links. Answers are limited to a single place, which in general represents a single entity. For example, it is possible to answer questions like "What is the birth date of the father of Einstein?". When there are several outgoing links with same type (e.g., "child of"), only one of them can be followed so that only one of several answers can be retrieved in one navigation path. For example, for the question "What are the birth dates of the children of Einstein?", the birth date of only one child can be retrieved at a time. Questions like "Who was born in Berlin, and died in Vienna?" cannot be answered because they involve two starting points, here "Berlin" and "Vienna", and a join between two criteria.

Hierarchies. The archetypal hierarchy used as a navigation structure is the file system, where navigation places are folders, and navigation links are moving to a child folder or to the parent folder. The contents of a folder is a set of files. Hierarchies of folders are also found in a number of applications, where files are replaced by emails or bookmarks. Other examples of hierarchies are found in Web directories, library catalogues, and as the product of clustering algorithms. At first sight, one could think that a hierarchy, as a tree, is just a particular case of a graph, but there are important differences. The first difference is that the objects of interest are not the folders, but the items in them (e.g., files). Therefore, each folder materializes a question, whose answers are the items in the folder. The second difference is that an item does not only belong to its folder, but also indirectly to the

[3]You can give it a try by starting, for example, at http://dbpedia.org/resource/Rennes.

parent folders, recursively. Therefore, the item set of a folder is included in the item set of its parent folder, and the former can be considered as a specialization or a refinement of the latter. For example, a folder "French films" would be a subfolder of "European films", which in turn would be a subfolder of "films". As a consequence, navigation is mostly top-down, moving from the most general folder towards more specific folders. Advantages compared to graphs are that several items can be found as answers, and that several criteria can be combined. For example, photos can be classified and retrieved according to both date and location by using navigation paths like `/2013/Europe/France/Rennes/`, where photos are classified first by year, then by successively smaller locations. However, a hierarchy suffers from "the tyranny of the dominant decomposition" [Tarr et al., 1999] because it forces a fixed ordering of the different criteria, and therefore limits the set of questions that can be answered, and hence expressivity. In the above example about photos, photos can be retrieved for the combination of a year and a location, for a year alone, but not for a location alone. The choice of folder also impacts the granularity of criteria. In the above example, photos can be retrieved by year, but neither by month nor by decade (using a single navigation path). Using a different hierarchy like `/Europe/France/2013/July/` only displaces the problem because it changes the set of possible questions without making it larger. Because of those constraints on criteria ordering and granularity, a hierarchy can only answer a small subset of conjunctive queries. A drawback of hierarchies compared to graphs is the lack of navigation links between items. For example, in a hierarchy of people classified by, say, nationality, occupation, and birth date, it is not possible to represent and navigate relationships between persons (e.g., "is a child of", "was influenced by").

Concept lattices. Concept lattices are a less known navigation structure [Godin et al., 1993] that comes from the theory of Formal Concept Analysis (FCA) [Wille, 1982, Ganter and Wille, 1999]. They are similar to hierarchies in that: (1) navigation places (called *formal concepts*) are ordered from the more general to the more specific, (2) every formal concept contains a set of items (called *objects*), and (3) every formal concept is characterized by a conjunction of criteria (called *attributes*). A first advantage over hierarchies is that concept lattices are formally derived from data. In FCA, data is organised into a *formal context*, i.e. a binary relation between objects and attributes. Therefore, users need only to describe objects by attributes, not to classify them. Because concept lattices are derived from formal contexts, they can be automatically maintained up to date. In this way, the

navigation structure can be maintained consistent and complete, properties that are notoriously difficult to establish in graphs and hierarchies. A second advantage of concept lattices is to remove the expressivity restrictions of hierarchies by allowing arbitrary conjunctions of attributes as queries. This is possible because a concept lattice can be understood as an overlaying of all possible hierarchies for a given set of criteria [Guillas et al., 2008]. To be more precise, all questions (i.e., sets of attributes) that share a same set of answers (i.e., sets of objects) are merged into a same formal concept. For a given concept, the set of answers is called its *extension*, and its most precise question is called its *intension*. A consequence is that there are often several possible navigation paths to a same concept. The main problem is that the number of concepts can be as high as exponential in the number of objects or attributes. Even though it is much lower in practice, it is still generally too high (e.g., a million concepts for a thousand objects) for the visualization of the whole concept lattice. Therefore, a commonly followed approach is to display only parent and child concepts of the current concept (recall that here navigation places are concepts), which can be computed and displayed in polynomial time [Lindig, 1995, Ducrou and Eklund, 2008]. To our knowledge, navigation based on concept lattices has only been applied to small formal contexts with tens to hundreds of objects. When applied to a larger collection, an initial user query is generally used to extract a small formal context [Carpineto and Romano, 1996]. Navigation in concept lattices is interoperable with querying as it is possible to start with any query (a set of attributes), to locate the corresponding concept by computing its answers, then to navigate from it. When the query has no answer, a solution [Messai et al., 2008] is to add a dummy one to the formal context, so that parent concepts contain the objects most similar to the query.

Discussion. We here discuss how navigation structures comply with the above properties (Section 2, p. 12). The main advantage of navigation structures is their *usability*. The navigation paradigm is a very natural one, and takes its roots in human cognition from navigation in the geographical space. Analogies can be drawn between graph navigation structures and road maps, and between hierarchies and the physical storage of books in a library. Only concept lattices do not have obvious analogues in the physical world, and are indeed more difficult to interpret by users. The user interface of navigation structures always follow the same structure: the identifier of the current place, a view of the current place contents, and controls to navigate to related places. The identifier can be a URL, a folder path, or a set of attributes. The view strongly depends on the kind of contents. It can be a single document,

a list of files, a mozaic of pictures, or a combination of those. The navigation links can be hypertext links, folders, or buttons. This kind of user interface provides a high *readability* because every element is easy to understand and use. It also provides *guidance* because each navigation link can be seen as a system suggestion to go on in the search. It is possible to navigate from every place, at least if we consider that going back in the navigation history or moving up to the parent folder are valid navigation links. *Specificity* is good in navigation structures, because they are built on top of data. Graphs can only connect existing places (e.g., Web pages), hierarchies are generally grown incrementally upon the addition of new items (e.g., file systems), and concept lattices are automatically derived from data and adapt to changes. Finally, *safeness* is guaranteed, except in hierarchies where there may be empty folders. One may contend that *guidance* is not *safe* because it is a common experience to follow a navigation path, and not to find the expected information, while present in the navigation structure. However, this is the *expressivity* of the navigation structure that should be blamed for that, rather than its *guidance* properties.

The main limitation of existing navigation structures is their low *expressivity*. It is limited to relation paths in graphs, and to conjunctions of attributes in concept lattices. In principle, it is easy to combine the two navigation structures, like in the Web that combines hypertext links and Web directories (in addition to search engines). However, the two kinds of queries cannot be combined as such, and therefore the set of expressible queries is only the set union of relation paths and conjunctions of attributes. For example, assuming a hierarchy of films and people, and relationships between them, it is not possible to find in one navigation path "all directors of French films" but only "the director of a particular film" or "all French films".

Navigation structures offer a very good *scalability*, when they are explicitly represented like graphs and hierarchies, because all computations are local to the current place (e.g., using HTTP links in the current Web page, reading the contents of the current folder). The best proof of that is the World Wide Web, which counts tens of billions pages[4]. In the case of concept lattices, the need to compute the navigation structure from data makes it much less scalable, and in practice, they have only been applied on small datasets.

[4]See http://www.worldwidewebsize.com/

2.3 Interactive Views

This category includes all systems in which users interact with a dynamic view over data. They follow the Model-View-Controller (MVC) paradigm: the model is the underlying dataset, the view is a representation of a subset of the dataset, and the controller is a set of available interactions that let users modify or transform the view. An analogy can be made with navigation structures with views playing the role of navigation places, and with controls playing the role of navigation links. However, users do not feel that they move in a navigation space, but on the contrary that it is data that moves in front of them. A better analogy is that of a scientist looking at an object through a combination of lenses and filters. The object is the dataset, and does not change. The combination of lenses and filters defines the current view, which can be controlled by adding or removing individual lenses or filters. This category is not so common, but it is valuable because it offers a rich user experience. For example, it is found under a simple form in email tools, where lists of emails can be sorted according to different criteria (e.g., by descending date or by ascending sender name), and filtered by some keyword. It is also found in interactive maps (e.g., Google maps), where a number of layers can be overlaid, and where it is possible to zoom in and out at every position and at every scale (within some range). We discuss in more details two techniques that have been well studied, and played an important role in our own research: faceted search and OLAP.

Faceted search. Faceted Search (FS) [Hearst et al., 2002, Sacco and Tzitzikas, 2009] 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: e.g., multimedia information bases [Hyvönen et al., 2002, Sacco, 2008, Amato and Meghini, 2008] (see Chap. 9 in [Sacco and Tzitzikas, 2009]). The data model is generally a collection of items described along a number of facets. This is similar to a table in a relational database but FS supports heterogeneous data in that some items may have no value for some facets, or on the contrary have several values for a same facet. A couple (facet,value) is called a *filter*, because it can be used to filter the collection to items having that value for that facet. The view is made of a set of items (called *selection*), and the subset of filters that overlap with the selection (called *restrictions*). This set of restrictions can be seen as an index or a summary of the selection. Assuming the selection is a set of photos taken in Australia, the restrictions are the dates of those photos, people visible on them, more precise locations in Australia, etc. Each restriction is generally displayed with the number

of items in the selection that match it. The selection is analogue to query answers, and restrictions are an important enrichement of the view that supports user feedback and understanding-at-a-glance of the dataset. In fact, when FS is used to explore a dataset without looking for a particular item, restrictions *are* the useful answers, and the selection is only a means to get them. Restrictions are also useful to modify the selection. Indeed, each restriction covers a subset of the selection, and that subset becomes the new selection when the restriction is selected by the user. In some systems, it is also possible to exclude a restriction, or to select/exclude an union of restrictions. For example, in the previous selection of Australian photos, it is possible to select photos taken in Sydney, or to exclude photos with an animal, or to select those showing Alice or Bob.

The query of which the selected items are the answers is not always explicit in FS, but it can be derived from the sequence of selection transformations chosen by the user. If only selections of restrictions are available, the reachable queries are conjunctions of filters. The expressivity of FS is then similar to concept lattices, where attributes play the same role as filters. If exclusions and unions are also available, the reachable queries are conjunctions whose conjuncts are either filters, negated filters, or unions of filters. This raises expressivity a little beyond concept lattices but far below querying languages like SPARQL. In particular, like for hierarchies and concept lattices, FS misses relationships between items.

Semantic faceted search covers a number of approaches that aim to apply faceted search to Semantic Web data. 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 [Mäkelä et al., 2006], mSpace[5], and Longwell[6], do not claim for a contribution in term of FS expressiveness, and contribute either to the design of better interfaces and visualizations, or to methods for the user-centric configuration of faceted views over semantic data [Hyvönen et al., 2005, Suominen et al., 2007]. The latter is important for the readability of faceted views given the frequent heterogeneity of semantic data. Therefore, their contributions are somewhat orthogonal to ours, and could certainly complement them. Other approaches extend faceted search in order to offer more expressive power w.r.t. relationships: e.g., /facet [Hildebrand et al., 2006], BrowseRDF [Oren et al., 2006], SOR [Lu et al., 2007], gFacet [Heim et al., 2010], VisiNav [Harth, 2010]. In this setting, facet-value pairs are property-object pairs, and other filters are introduced such as classes, property domains, and property ranges. In addi-

[5]mSpace `http://mspace.fm/`
[6]Longwell `http://simile.mit.edu/wiki/Longwell`

tion to selection and exclusion, an additional selection transformation is to *cross* a relation. For example, starting from a selection of French films, the relation "has director" can be crossed to reach the selection of all directors of a French film. This is precisely the kind of query that could not be answered with navigation structures because navigation was either from one item to one related item, or from a set of items to a subset of items. However, even though all ingredients of the core SPARQL are present (i.e., relationships, join, union, negation), existing systems have a number of limitations that makes them significantly less expressive than SPARQL. As we have shown in detail [**Ferré and Hermann, 2012**], the main cause for such limitations is that selection transformations are defined as set operations between the selection and restrictions. In particular, this prevents to reach tree-shaped queries like "Who directed a film produced in the US and starring a woman born in France?". This also prevents cycle-shaped queries like "Who directed a film in which he/she also starred?".

OLAP. On-Line Analytical Processing (OLAP) [Codd et al., 1993] was introduced by E.F. Codd to facilitate the analysis of data warehouses by people that are data specialists but not computer specialists. It is an important technique in Business Intelligence (BI), and often works on top of relational databases [Codd, 1970]. The OLAP model is the data *cube* [Pedersen and Jensen, 2001]. A cube has a number of *dimensions* (e.g., city, date, and product), where each dimension has a finite domain of values (e.g., a set of cities, a set of dates, and a set of products). The Cartesian product of those domains of values defines a set of *cells*, one for each combination of a value for each dimension (e.g., ("Rennes", "24 July 2013", "X-shoes")). Each cell contains one or several values, called *measures* (e.g., quantity). Each cell represents a *fact* (e.g., "a sale"), i.e. a combination of values for dimensions and measures (e.g., "12 X-shoes were sold in Rennes on 24 July 2013"). In fact, a cube is equivalent to a relational table, where dimensions and measures are attributes, and the set of dimensions is a key for the table. OLAP cubes are often defined as views over a relational database (R-OLAP), and several OLAP cubes may be defined on the same data to support various data analyses. An important OLAP ingredient that is not native in relational databases is the notion of *hierarchies* of values on dimensions. A dimension hierarchy enables the grouping of dimension values at different levels of granularity. For example, dates can be grouped by month and years. Similarly, cities can be grouped by regions and countries, and products can be grouped by categories. When different values are grouped on a dimension, some cells are merged, and an *aggregation* operator needs to

be applied to still have one value per measure in each cell. In our example, quantities can be summed when grouping dates by month or when grouping products by category.

An OLAP view is simply an OLAP cube, possibly projected to a subset of dimensions and measures, and with a chosen granularity level for each dimension. As the presentation is generally 2-D, the first two dimensions are presented as a two-dimensional table, and other dimensions are nested in various forms: nested tables, histograms, pie charts, maps, etc. Users can interact with the OLAP view by applying a number of possible operations. The granularity level on each dimension can be made finer (*drill-down*) or coarser (*roll-up*). Dimensions and measures can be shown or hidden. The order of dimensions can be changed (*pivot*). On each dimension, subsets of values can be selected in order to zoom on sub-cubes (*slice* and *dice*). In some cases, it is also possible to change the aggregation operator on measures, and to exchange the role between dimensions and measures.

We now relate OLAP views to SPARQL 1.1 queries in order to evaluate their expressivity. A data cube translates to a fixed graph pattern that retrieves facts by relating values for dimensions and measures. Each granularity level translates to another graph pattern that relates the finest-level value to the coarser-level value. Dimensions translate to GROUP BY clauses and projected variables, and measures translate to aggregations. Hidden dimensions and measures are simply removed from the SELECT and GROUP BY clauses. This translation shows that OLAP supports complex analytic queries that are not covered at all by faceted search and navigation structures. However, an important limitation is that the main graph pattern is fixed for a given data cube. In practice, this means that users depend on the managers of the information system for the definition of new cubes, and therefore can only get answers for predefined sets of questions. Another limitation is that data cubes are mono-valued, i.e. each cell has only one value for each measure. That limitation corresponds to the first normal form (1NF) in databases. It is generally fine for numerical data, but can be problematic for relational data. For example, one may want to visualize a cube of the actors of films per director and year. In this case, the measure is a set of actors, which could easily be displayed as a list in each cell of a two-dimensional table. In fact, 1NF is seldom contested in databases, but deserves to be so here because OLAP cubes are *views* over data, and not *models* of data.

Discussion. We here discuss how interactive views comply with the same above properties. Interactive views have a *usability* that is similar to that of navigation structures while offering more *expressivity*. The idea that an

initial view is presented, and that users can trigger controls to transform this view iteratively, is arguably a *readable* metaphor. For complex views and transformations, user training may be necessary, unlike with navigation structures. However, the spreading use of faceted search in e-commerce websites (see [Sacco and Tzitzikas, 2009], p. 265) demonstrates that interactive views can be *readable* from scratch by users. Because views and controls that transform them can be seen as a virtual navigation structure, interactive views offer *guidance* to users, and can even be qualified as guidance-based, where available controls play the role of system suggestions. Indeed, only available controls are applicable, and every applicable control returns a valid view. Therefore interactive views satisfy *safeness* and *specificity*.

The greater *expressivity* compared to navigation structures comes from the fact that views are not defined explicitly and *a priori*, but are the result of a computation, i.e. the successive application of view transformations chosen by users. This enables a set of reachable views that can be larger than any manually-defined navigation structure, be it the World Wide Web. The more reachable views there are, the more *expressivity* is available. This is interesting because it suggests that the *expressivity* of a system can be increased "simply" by adding transformations, and possibly by adding information into views. Because it is necessary to decide dynamically for each view which transformations are applicable, and to compute the selected transformations, the *scalability* of interactive views is lower than for explicit navigation structures. However, a good deal of research in both faceted search and OLAP has been devoted to *scalability* with results satisfactory enough to allow for industrial application (in e-commerce for faceted search, and in business intelligence for OLAP).

2.4 Summary and Comparison

Table 1 summarizes the approaches presented above by comparing them w.r.t. our four main properties: expressivity, readability, guidance, and scalability. The objective here is not to make a precise and technical comparison between them, but rather to draw conclusions on how to combine them in order to add their strengths. We have adopted for each property a coarse scale that ignores many details, but is sufficient to exhibit the strengths and weaknesses between approaches. For example, three levels of expressivity have been retained: low, medium, and high. In fact, even in a same approach, different systems may have very different expressivity (e.g., CNL, FS). Furthermore, different systems may not be comparable in that each has features that the other misses (e.g., OLAP vs FS). FL exhibit the highest expressivity, and no other approach gets close to them. At the other end

Approach	expressivity	readability	guidance	scalability
Formal languages (FL)	high	difficult	unspecific	database
Natural languages (NL)	low	natural	unspecific	database
Controlled NL (CNL)	medium	easy	unspecific	database
Graphs (G)	low	natural	specific	Web
Hierarchies (H)	low	natural	specific	Web
Concept lattices (CL)	medium	easy	specific	database
Faceted search (FS)	medium	easy	specific	database
OLAP	medium	easy	specific	database

Table 1: Comparison of the different approaches to information access w.r.t. key properties.

of the scale, low expressivity concerns approaches where direct navigation in data is used (G, H), or where queries must belong to a fixed number of patterns (NL), or where the size of those patterns is bounded (NL). All other approaches allow for combinatorial queries, generally conjunctive queries, possibly with additional features (e.g., crossing relations in semantic FS, aggregations in OLAP, conjunctive queries in FCA), and are given a medium expressivity. CNL are scored higher than NL because their syntactic and semantic analysis is easier, which enables the combinatorial aspect.

Readability is qualified as *natural* when no user training is necessary and every user input is accepted, *easy* when some short training may be necessary and most people can manage to use it, and *difficult* when a longer training is necessary and only a few specialists can use it. Graphs, hierarchies, and NL are qualified as natural because they are already well-known by everybody. FL are obviously the most difficult to use approaches. Other approaches (CNL, CL, FS, OLAP) have precisely been designed to be easy to catch on, and offer a good usability.

Guidance comes in various forms, but can be classified in only two levels, depending on the specificity (and safeness) of the guidance. For query languages, only unspecific syntax-based and ontology-based guidance exists, and there is often no guidance at all. In other approaches, guidance is always available, and only leads to valid navigation places or views.

Scalability is tricky to compare between systems as different as a query engine and faceted search, and it does not play a crucial role in our analysis. Therefore we simply discriminate between, on one hand, graphs and hierarchies that can work at Web-scale, and, on the other hand, other approaches that can work at the scale of a large local database. Because of their dynamic computations of navigation links and view transformations during

interaction, FS, OLAP, and CL are more demanding in data access and computation, and are therefore more difficult to scale than query languages.

From Table 1, four classes of approaches can be identified with their strengths and weakness:

1. FL: formal languages offer the highest expressivity but the least usability;
2. NL/CNL: (controlled) natural languages offer a trade-off between expressivity and readability;
3. G/H: explicit navigation structures are the most usable and scalable, but offer only low expressivity;
4. FS/OLAP/CL: interactive views offer a balance between expressivity and readability, plus specific guidance.

The classes FL and G/H are the more contrasted, and also the more widespread today. The class FS/OLAP/CL is the more balanced by combining medium expressivity, easy readability, and specific guidance. Those observations are made graphically visible in Figure 1. The approaches NL and CNL are displayed with dash-dot lines because they are respectively included in G/H and FS/OLAP/CL classes. That inclusion is mostly the consequence of our coarse scales, and NL/CNL remain the approaches of choice for reconciling expressivity and readability.

3 Main Contributions

From the beginning, our objective has been to reconcile querying and navigation, i.e., to make querying more *usable*, or equivalently, to make navigation more *expressive*. Therefore, our work has concentrated on combining and increasing *expressivity* and *guidance*. However, we have also been concerned with *readability* that ought to be at least *easy*, and with *scalability* that should be similar to FS. Our contributions started with concept analysis (FCA), and then integrated principles from faceted search (FS), the Semantic Web (SW), OLAP, and controlled natural languages (CNL). The result is a quite diverse collection of theoretical results and softwares. In order to present a synthesized view over our main contributions, we introduce a general framework that encompasses them all: *Abstract Conceptual Navigation* (ACN). In Section 3.1, we first define ACN, and discuss how it integrates the three categories of information access discussed in Section 2 (query languages, navigation structures, and interactive views), and how it promotes

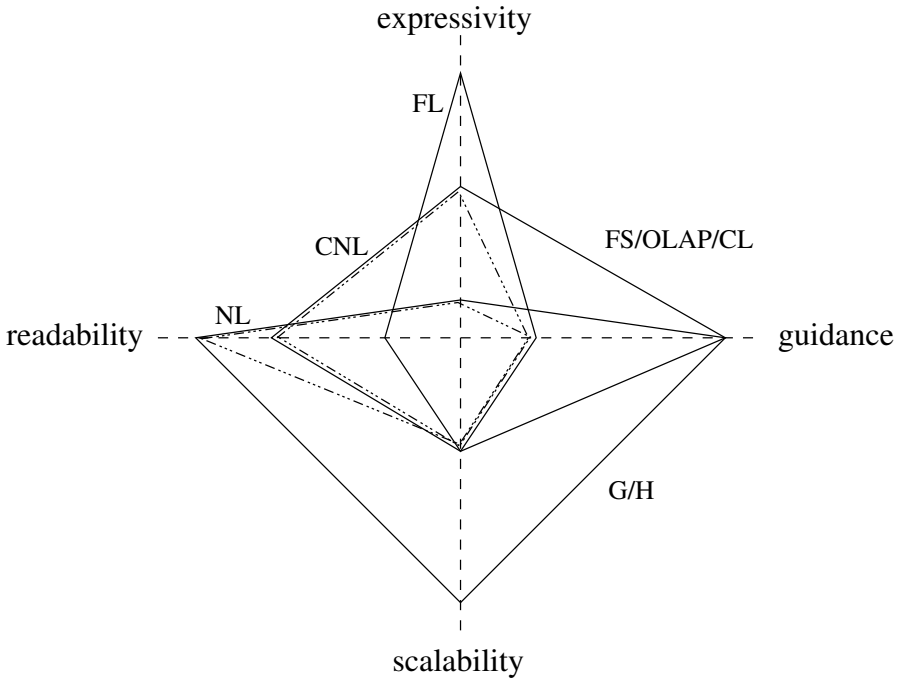


Figure 1: Radial diagram comparing the different approaches.

the desired properties (expressivity, readability, guidance, and scalability). We then shortly describe our main contributions as different instances of the ACN framework. The last section, Section 3.7, describes SQUALL, a CNL for the Semantic Web that is so far only a partial instance of ACN, but that will help improve readability and expressivity in ACN.

3.1 Abstract Conceptual Navigation (ACN)

Abstract Conceptual Navigation (ACN) is analogous to an abstract class in object-oriented programming in that it is made of a number of components to be defined in order to build concrete information systems. ACN is useful as a general framework for reconciling expressivity and usability, but not as a concrete theory because no precise theoretical judgement can be made about it. We reuse the term *Conceptual Navigation* from FCA because the key idea is still to navigate from concept to concept, where each concept defines a navigation place, and where each concept is made of an extensional part, and an intensional part. The qualifier *Abstract* emphasizes the fact that those extensional and intensional parts can be very different from the object sets and attribute sets of FCA.

ACN is defined by the following components (see Figure 2 for a schematic

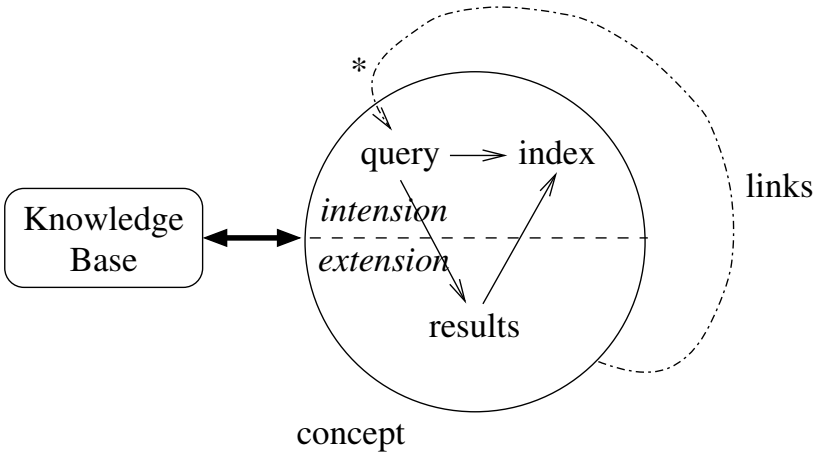


Figure 2: Schema showing the different components of Abstract Conceptual Navigation (ACN), and their interactions.

representation):

knowledge base (K) K is a *knowledge base* that contains the formal representation of facts, rules, ontological axioms, taxonomies, etc. Its content is constrained by the needs of the application, and the availability of data and domain knowledge. It is an implicit parameter of all ACN operations defined below.

query language (Q) Q is a *query language*, i.e. a set of expressible queries. Here, *query* should be understood in a broad sense, and may include closed and open questions, analytical questions (OLAP), sets of keywords, folder paths, updates, commands, ontological assertions, etc. The current query characterizes the current concept, and is part of its intension.

extension (E and $ext \in Q \to E$) E is the set of all possible extensions (a.k.a., query results), and *ext* is a mapping from queries to their extension, given a knowledge base K. The query extension is any piece of data returned to users from the evaluation of the query. It can be a set of values, an answer list, a table, etc. For some queries, e.g. updates, the knowledge base may be modified as a side effect. The extension of the current query makes up the extension of the current concept.

index (I and $index \in Q \times E \to I$) I is the set of all possible indexes over the extension, and *index* is a mapping from queries and their extension to their index. An index is at the same time a summary over the

extension, and a set of query refinements. It is part of the intension of the current concept because it is made of query elements. Its role is to provide feedback and guidance.

links (*links* $\in Q \times E \times I \rightarrow 2^Q$) component *links* defines a set of navigation links from the current query to neighbour queries. Links can be derived from any component of the current concept: the query, the extension, or the index.

The advantage and *raison d'être* of ACN is to subsume the three categories of information access discussed in Section 2: query languages, navigation structures, and interactive views.

- *ACN as a query language.* Q is the query language, and *ext* defines query results for each query. Indexes and links are void. For instance, SPARQL editors can be seen as partial ACN instances where the query language is SPARQL, and where extensions are tables of results or RDF graphs.

- *ACN as a navigation structure.* Navigation places are specified by queries in Q, and navigation links are given by the ACN component *links*. Extensions and indexes may be used to compute the links, and may be used to define the contents of navigation places. For instance, file system hierarchies can be seen as partial ACN instances where queries are directory paths, extensions are file lists, and navigation links lead to children directories and the parent directory.

- *ACN as an interactive view.* For recall, interactive views follow the MVC architecture (model-view-controller). In ACN, the model is the knowledge base K. The view is the composition of a query, its extension, and its index (an element of $Q \times E \times I$). The controller is made of the links, which are derived from the view contents. Each link activation generates a new view by computing a new extension (*ext*) and a new index (*index*) from the target query. For instance, faceted search can be seen as an ACN instance where a query is a set of facet-values, an extension is a selection of items, an index gives the frequency of facet-values among those items, and links allow the addition and removal of facet-values to the query.

On the positive side, ACN inherits *expressivity* from query languages, and *guidance* from navigation structures and interactive views, which achieves our main objective to reconcile *expressivity* and *usability*. However, it must be noted that the actual *expressivity* may be less than the *expressivity* of

the query language in the case where navigation links are not rich enough to reach all valid queries. Ideally, links should be both *safe* and *complete*. *Safeness* means that no navigation path leads to dead-ends (e.g., empty results). *Completeness* means that every safe query can be reached through a finite navigation path. *Safeness* is important for the quality of *guidance* as it avoids users to "bump in the walls", and *completeness* is important to leverage the *expressivity* of the query language. Therefore, a critical issue in ACN is the definition of links, and hence the definition of extensions and indexes because links are derived from them.

On the negative side, ACN inherits the *un-readability* of query languages, and the *un-scalability* of interactive views. *Readability* was not a big issue in our first work because the query language was simple enough, but became a bigger issue in more recent work with SPARQL-like *expressivity*. We have started addressing this issue with the help of CNLs (see Section 3.7). *Scalability* is the least addressed property in our work, but we have always excluded approaches that could not scale in principle (e.g., computing concept lattices), and strived to stick to linear-to-quadratic time complexities in the size of the dataset (e.g., number of objects in a collection).

3.2 Logical Concept Analysis (Camelis)

The first ACN instance is based on the theory of Logical Concept Analysis (LCA), and has been implemented as two systems, Camelis and LisFS, and three Web interfaces on top of them: Geolis, Abilis, Portalis (more details below). We have introduced LCA [**Ferré and Ridoux, 2000b**, **Ferré, 2002**, **Ferré and Ridoux, 2004**] as a logical generalization of FCA for the purpose of defining Logical Information Systems (LIS) [**Ferré and Ridoux, 2000a**, **Ferré, 2002**, **Ferré and Ridoux, 2004**, **Ferré, 2009a**]. LCA generalizes FCA by replacing sets of attributes, which are used for the representation of object descriptions and concept intents, by the formulas of ad-hoc logics. The benefit is a gain in expressivity when using concept lattices as navigation structures because this allows for more concepts and more links between concepts. Whereas FCA only offers binary attributes and conjunction, LCA can be plugged in with logics that offer valued attributes, patterns over values (e.g., intervals of dates, string patterns), disjunction, and negation [**Ferré and Ridoux, 2001b**, **Ferré, 2006**]. To facilitate the design of new ad-hoc logics, we have built a rich toolbox of logic components, called *logic functor* [**Ferré and Ridoux, 2001a**, **Ferré and Ridoux, 2006**], that can be composed in a plug-and-play fashion, while automatically verifying that the composed logic satisfies properties that are necessary

for LIS (e.g., consistency and completeness of entailment). The notion of functor has been extended to LCA knowledge bases, *logical contexts*, to improve the efficiency of LIS [**Ferré, 2009b**].

A number of other generalizations of FCA have been proposed. Generalized FCA [Chaudron and Maille, 2000] and pattern structures [Ganter and Kuznetsov, 2001] independently discovered the same core definitions. However, those approaches have sticked to the computation and display of concept lattices for data mining purposes, and have therefore been limited in terms of expressivity and scalability for navigation purposes. We have also pushed farther the generalization with generic pluggable implementations and the toolbox of logic functors. This genericity allowed us to develop quickly new LIS applications with complex and very different logics (e.g., function/method type signatures in programming libraries with Soazig Bars [**Bars et al., 2002**], sequence motifs in bioinformatics with Ross D. King [**Ferré and King, 2004**], lexicalized grammars in linguistics with Annie Foret [**Foret and Ferré, 2010**]).

Computing and displaying concept lattices in FCA as navigation structures does not scale well. This is even worse in LCA because a higher expressivity implies a higher number of concepts. However, the concept lattice can still be used as a navigation structure by computing, on demand, links to neighbour concepts only. However, our experiments have shown that an interactive view is more readable than a large navigation structure, in which users can get lost. The user experience in interactive views is the building of more and more complex selections of objects, with a clear view of the current selection at each step. In our approach, the index shows every feature that occurs in the extension, and at which frequency (e.g., 50% of the photos in the extension are portraits). Moreover, the index can be organized and displayed as a partial ordering according to logical entailment (e.g., all photos taken in France are also taken in Europe). The index provides understanding at a glance, and navigation through the selection of features.

Interestingly, other researchers converged to very similar ideas at about the same time: dynamic taxonomies [Sacco, 2000] and faceted search [Hearst et al., 2002] are closely related. Faceted search is more advanced in terms of usability and scalability, and we are more advanced in terms of expressivity and formalization. We made contributions to dynamic taxonomies and faceted search by using logics for the design of taxonomies [**Ferré and Ridoux, 2007**], and by allowing for more complex selections involving disjunction and negation [**Ferré, 2008**]. This led to a collaboration with G.M. Sacco and Y. Tzitzikas that resulted in the edition of a book [Sacco and Tzitzikas, 2009], in which we contributed to five chapters.

We now sketch a formalization of LCA-based information systems as an ACN instance. Full details are available in Appendix A [**Ferré and Ridoux, 2004**] (for theoretical aspects) and Appendix B [**Ferré, 2009a**] (for practical aspects).

knowledge base $K = (\mathcal{O}, \mathcal{L}, d)$: a *logical context*, where:

- $\mathcal{O}$ is a set of objects,
- $\mathcal{L}$ is a logic, defined as a set of formulas partially ordered by an entailment relationship $\sqsubseteq$, called *subsumption*,
- $d \in \mathcal{O} \to \mathcal{L}$ is a mapping from objects to their logical description.

query language $Q = \mathcal{L}$: queries are logical formulas.

extension $E = 2^{\mathcal{O}}$ and $ext(q) = \{o \in \mathcal{O} \mid d(o) \sqsubseteq q\}$: extensions are sets of objects, and the results of a query are the objects whose description entails, i.e. "matches", the query.

index $I = (\mathcal{L} \to \mathbb{N})$ and $index(q, e) = \{x \mapsto n \mid x \in Features(K) \subseteq \mathcal{L}, n = \#(e \cap extent(x)) > 0\}$: the index of a query is the set of *features* x that are "matched" by a number $n > 0$ of query results. Given a context K, the features are automatically extracted from object descriptions (indexation), and can also be manually specified by users.

links $links(q, e, i) = \{q \text{ and } x \mid (x \mapsto n) \in i\} \cup \ldots$: the main links consist in conjuncting the query with a feature from the index. Other links enable to reset the query, remove some feature from the query, or to introduce disjunctions and negations of features in the query.

LCA-based information systems were first implemented as a standalone application, Camelis[7]. It was first implemented in λProlog [Miller and Nadathur, 1986, Belleannée et al., 1999] in 1999, and then re-implemented in OCaml [Chailloux et al., 2000] in 2002. At the beginning, it had a command-line interface that reused the `cd/ls` paradigm of UNIX file systems. In 2005, it was equipped with a GUI that offered richer visualizations, and better usability. Since then, I have been using it for various purposes (e.g., bibliography, personal photos, homedir files), and it has also been used by other members of the LIS team and beyond (e.g., program execution traces, linguistic data, mushrooms, birds, calls for papers). The efficiency of Camelis was progressively increased, and can

[7] http://camelis.gforge.inria.fr/

now manage logical contexts up to 10 million object×feature, e.g. 100,000 objects with 100 descriptors each.

The second implementation, LisFS (Logical File System) [Padioleau and Ridoux, 2003], was developped by Yoann Padioleau during his PhD [Padioleau, 2005]. The originality of LisFS is that it is implemented as a real Linux file system, behind the VFS (Virtual File System) interface. This implies that the standard `cd/ls` shell commands behave according to LIS principles, and not according to the classical hand-made folder hiearchy. For example, a file path can be `/author:Padioleau/author:Ridoux/title:"A Logic File System"/year:2003/PadRid2003.pdf`, where directory names are object descriptors, whose order is not relevant. A directory path represents a query, subdirectories correspond to index elements and navigation links, and local files are the objects in the extension that do not belong to subdirectories. For example, from the directory `/title:*Logic*/year:>2000/` (i.e., files whose title contains "Logic", and whose year is after 2000), some valid subdirectories are `author:Padioleau`, `year:2003`, `title:*File*`. LisFS was also extended in order to refine the granularity from files to lines of textual files [Padioleau and Ridoux, 2005]. This was applied to the browsing of source code, e.g., selecting method definitions with a given class in their signature.

Geolis [Bedel et al., 2006][**Bedel et al., 2008b**, **Bedel et al., 2012**] is a Web interface working on top of LisFS, and was developped by Olivier Bedel during his PhD [Bedel, 2009]. It extends LisFS as a Geographical Information System (GIS) in order to browse geographical data. New logic components were designed to represent spatial geometries (e.g., points, lines, polygons), and the objects of the extension are displayed on a map instead of as a list. When index features are spatial geometries themselves, they are displayed on the map and can be selected to refine the selection, like any other feature. Like for date intervals or string patterns, custom geometries can also be drawn on the map for refining the query, and the index and navigation links can still be computed by the system afterwards. The main contribution of Geolis w.r.t. other GIS is to abstract over the distinctions that are usually made between the different layers on one hand (e.g., water layer, building layer), and between geographical data and thematic data on the other hand. Indeed, each layer is simply represented as a descriptor for its objects, and geographical/thematic data simply use different logics for their representation. This abstraction allows for a more flexible and more uniform exploration of geographical data. Another contribution is the first introduction of relationships between objects in a LIS system [**Bedel et al., 2008a**], which were used for the representation of Euclidian and topological relations

between objects (more about relationships in Section 3.4).

Abilis[8] is a Web application on top of Camelis. Its purpose is to make Camelis available without the burden of software installation, and also to enable collaborative work and group decision on logical contexts [**Ducassé and Ferré, 2008**]. It was developped by Benjamin Sigonneau, Véronique Abily, and Vincent Alleaume, and is based on the Ocsigen framework[9]. In Section 3.3, we discuss how Abilis was extended by Pierre Allard with OLAP views and maps.

Portalis is a Web portal for managing and accessing online Camelis repositories. Its objective is to make it easy to define all sorts of clients on top of Camelis, e.g. Web applications, mobile applications, APIs. It encapsulates Camelis as a Web service, and manages all aspects related to available services, user sessions, and access rights. It has been developed by Yves Bekkers and Benjamin Sigonneau under a FEDER funding from Europe and the Brittany region.

3.3 Cubes of Concepts (Abilis)

Search results are not always expected to be lists of objects. OLAP views (see page 22) are data cubes, i.e., multi-dimensional arrays of aggregated measures. Such views are key to analytical questions such as "What are the total sales per product category and per year?". LCA-based information systems have been extended with OLAP views and controls by Pierre Allard during his PhD [Allard, 2011]. This relies on a clear distinction in LCA between attributes and values, which is not required by LCA, but was already done in practice anyway. Attributes play the role of dimensions and measures, and values play the role of cube coordinates in the case of dimensions, and cell contents in the case of measures.

A first contribution to existing work in FCA and OLAP is that we allow attributes to be multi-valued, i.e. that an object may have several values for a same attribute. This implies that a same fact can belong to several cells of the cube, and that a cube cell may contain several values. This is useful, for example, in a bibliography context when facts are documents, because documents may have several authors, and several topics. It becomes possible to answer questions like "What is the number of documents per author and per topic?", where each document will be counted once for each combination of an author and a topic. In fact, considering multi-valued attributes is equivalent to consider attributes as relations instead of as functions (e.g.,

[8] Try it at http://ledenez.insa-rennes.fr/abilis/

[9] http://ocsigen.org

the attribute "author" is a relation from documents to people). This is a departure from the often undisputed first normal form of databases.

A second contribution to most OLAP approaches is that dimensions and measures are symetrical, and their roles can be exchanged. All attributes can be used in three different roles: as a dimension, as a measure, and as a query element to select a subset of facts. Furthermore, a same attribute can be used in the three roles at the same time, like in the question "Among the documents written by Smith or Jones, what is the number of co-authors per author?":

- in the query: only documents with *author* Jones or Smith are considered,
- as a dimension: those documents are grouped by *author* (each document may fall in several groups),
- as a measure: in each group, the *authors* are collected and counted.

A third contribution to OLAP is that drill-down and roll-up, which usually apply to dimensions, can also apply to measures. For example, in the question "What are the publication decades per author?", the measure is the publication year (e.g., 1996) rolled-up at the level of decades (e.g., nineties), and hence, the cell contents are sets of decades.

The principle of this ACN instance is to extend a LCA query with dimensions and measures. The selection of a bare attribute (no specified value) is interpreted as selecting all values of this attribute in parallel, and hence reaching a row of concepts. The selection of a second bare attribute is interpreted as selecting all possible values of this attribute, for each value of the first attribute, and hence reaching a 2D-array of concepts. More generally, after selecting several bare attributes, we obtain a *cube of concepts*, whose dimensions are those attributes. Navigating to cubes of concepts is like following many navigation paths in parallel, and organizing the results into a cube, like in OLAP. Measures then serve to define the rendering of concepts, i.e., the contents of the cells of the cube. We here sktech this ACN instance by reusing the definitions of Camelis (more details are available in a paper at ICFCA'12 [**Ferré et al., 2012**]).

knowledge base K is a *multi-valued logical context*, i.e., a logical context where objects are described by a number of attributes, where each attribute may have multiple values (zero, one, or more). Equivalently, K is a set of triples (object, attribute, value). Value domains are LCA logics, hence partial ordering of values, and are extended with granularity levels (e.g., days, months, years for dates), and with aggregators (e.g., "sum" for numbers).

query language A query is here made of three components:

1. a *logical formula* that selects a subset of objects, like in CAMELIS;
2. a vector of *dimensions*, where each dimension is an *axis*, i.e., the combination of an attribute and a granularity level;
3. a vector of *measures*, where each measure can be any of: the extension, the count of the extension, the index of an axis, an aggregation of the index of an axis.

extension The logical formula of the query determines a selection of objects, the *global extension*. The dimensions project this global extension into a cube of *local extensions*, which may overlap each other because of multi-valued attributes. Each cell of the cube corresponds to a particular valuation of the dimension attributes, and hence to a particular refinement of the logical formula, and hence to a particular concept. A local extension is the extension of a local concept. The measures define the contents of each cell, and each measure is defined as a function of the local concept. The extension (a set of objects) and the index (a set of values with their frequency) of a local concept are defined like in CAMELIS. The count of a concept is simply the cardinal of its extension. The aggregation of an index, is the aggregation of its values, taking into account their frequency.

index The index is the same as in CAMELIS, based on the global extension. Dimensions and measures have therefore no impact on it.

links The links of CAMELIS are safe, and modify the logical formula of the query, and hence the global extension. Additional links are the addition/removal of attributes as dimensions and measures, the change of granularity levels (drill-down and roll-up), and the choice of aggregations on measures.

Cubes of concepts have been implemented in ABILIS by Pierre Allard. In addition to the computation of extensions, indexes, and links, it deals with the question of richly displaying cubes of concepts. Dimensions are rendered according to their number, value domains, and user choices. If there are two dimensions, they are rendered as a 2D table. If there are more dimensions, those are nested as sub-tables. Innermost dimensions with a numeric measure can be rendered as histograms or pie charts. Temporal dimensions can be rendered as timelines, and spatial dimensions are rendered as maps. Measures are rendered according to their type. Extensions are rendered as lists, and indexes are rendered as tag clouds, based on the frequency of values.

Cubes of concepts have been applied to geographical data analysis [Allard, 2011], and to group decision [**Ducassé et al., 2011**]. The geographical application demonstrated that geometries (here, polygons representing administrative areas) can be organized into a logic of values. Those geometries define either a dimension of the cube on a map, or a measure through geometric aggregations (e.g., union) or metric measures (e.g., area). The group decision application helped a research team to define a publication plan for the next year based on a collection of about 1000 conferences. Each conference was described by its topics, deadline, and rank (A, B, C, or none). ABILIS was useful to collaboratively define a selection of interesting conferences, and then to assign team members as the main author for each of those conferences. A concrete workplan was obtained in ABILIS by reaching the query "What are the selected conferences per deadline month and per main author".

Interestingly, we also found cubes of concepts useful for the discovery of functional dependencies and association rules [**Allard et al., 2010**]. Let us assume that the attributes $x_1, \ldots x_n$ have been chosen as dimensions, and the index of the attribute y has been chosen as the measure. The rule $x_1, \ldots, x_n \rightarrow y$ is a functional dependency if every cell of the cube has at most one value. Each cell that has exactly one value defines an exact association rule. When a cell has several values, each value defines an approximate association rule. In ABILIS, values are displayed in a font size proportional to their frequencies. Thus, the support and confidence of association rules can be directly visualized. Cubes of concepts therefore allow to visualize large sets of rules at once. The roll-up and drill-down operations are also useful to explore functional dependencies and association rules. More functional dependencies and exact association rules can be obtained by a drill-down on a dimension (making the rule premise more precise) and by a roll-up on a measure (making the rule conclusion more general). The main limitation is that users are not guided in the selection of the premise and conclusion attributes, whereas the number of discovered rules may vary a lot depending on those selections.

3.4 Query-based Faceted Search (SEWELIS)

The main limitation of Formal Concept Analysis (FCA) and Faceted Search (FS), and hence of our LCA-based ACN, is the lack of relationships between objects. Each object can be related to a number of features, and features can be related together by subsumption relationships thanks to logic, but objects cannot be related to each other. This restricts the application of ACN to collections of one type of objects, e.g. photos, documents, people. For

instance, in a bibliographical context, one has to choose whether documents (as objects) are described by people (as author, editor, etc.), or the reverse. In each case, not all useful queries can be answered, which is reminiscent of hierarchies where no hierarchy could answer all useful queries (see page 16). The importance of relationships is reflected by the fact that widespread query languages like SQL and SPARQL not only support them, but are based on them. SQL is the standard query language for *relational* databases, where every input and output is a relation. SPARQL is the standard language for the Semantic Web, which is relational by nature (RDF graphs).

There have been a number of proposals to extend both FCA and FS with relationships. Relational Concept Analysis (RCA) [Hacene et al., 2007, Rouane-Hacene et al., 2013] extends FCA by considering a context for each type of objects and a context for each relation. A concept lattice is produced for each type of object, and take into account relationships with other kinds of objects. This is used for unsupervised data-mining, and applied in software engineering, e.g. to model transformations [Dolques et al., 2010]. RCA is not directly suitable to search because there is one concept lattice per type of objects, and hence independent navigation spaces. We have independently proposed a similar extension of LCA (RLCA) [**Ferré et al., 2005**] except that a single concept lattice is defined that encompasses both object descriptions and relationships. Like for LCA, this relational concept lattice is not computed but serves as a navigation structure for the purpose of search and exploration. However, the expressivity in RLCA is still quite limited compared to SQL or SPARQL, as it is mostly restricted to linear navigation paths through relations. This expressivity is in fact similar to extensions of FS for the Semantic Web (see page 21 about semantic faceted search). In both cases, neither tree patterns, nor graph patterns with cycles, can be expressed, and even less their combination with disjunction and negation.

The ACN instance presented here enables to reach through navigation arbitrary combinations of graph patterns (including cycles), disjunctions, and negations, i.e. a large fragment of SPARQL [**Ferré, 2010**]. It is founded on *Query-based Faceted Search* (QFS) [**Ferré and Hermann, 2011**, **Ferré and Hermann, 2012**], a generalization of FS where navigation links are defined as query transformations instead of set operations on the extension. This generalization is in fact the essence of Abstract Conceptual Navigation, and the key to higher expressivity. We now sketch a formalization of QFS as an ACN instance. Full details are available in Appendix C [**Ferré and Hermann, 2012**].

knowledge base K is an RDF graph, i.e. a set of triples. The triples may be explicitly stated, or implicitly derived by logical inference. Inference

maybe based on an RDF schema, OWL axioms, or rules.

query language Q is based on LISQL, a high-level syntax for our SPARQL fragment that minimizes the use of variables, and facilitates the insertion of disjunctions and negations (see examples in Appendix C). An ACN query $q \in Q$ is the combination of a LISQL expression (a query in the classical sense) and a distinguished sub-expression, called *focus*. That focus is taken into account in the definition of the extension and the index, and serves as an insertion point for query transformations. It plays an essential role in the construction of complex queries, and hence in the expressivity of QFS. Intuitively, different foci correspond to different word ordering or intonation breaks in NL sentences. For example, the sentence "The mouse, the cat plays with it." has the same meaning as the sentence "The cat plays with the mouse.", but it puts the focus on the mouse rather than on the cat.

extension Extensions are still sets of objects, but the membership of an object to the extension depends on its relationships to other objects. The extension $ext(q)$ can be defined in terms of SPARQL. Given that the LISQL expression of q translates to a SPARQL graph pattern P, and that the focus of q translates to a distinguished variable x of that graph pattern, $ext(q)$ is the answer set of the SPARQL query `SELECT` x `WHERE` { P }.

index Given extensions are sets of objects, like in the LCA instance, the index can be defined in a similar way. Here, the main index elements are the types of objects (i.e., classes), and the properties that apply to them.

navigation links Extension objects and index elements can be inserted at the focus. The focus can be moved to another sub-expression of the LISQL expression. Disjunction and negation can be inserted at the focus. The sub-expression at the focus can be removed. This set of navigation links is enough to build arbitrary LISQL expressions.

Our shift from FCA and LCA contexts to RDF graphs is not as big as it seems. Most descriptors in LCA are valued attributes $a = v$, and the association between an object o and a descriptor is then equivalent to a triple (o, a, v), where v is then an RDF literal, and a is a data property. In relational extensions of FCA and LCA, the association of a couple of objects (o_1, o_2) to a relation r is equivalent to a triple (o_1, r, o_2), where r is an object property. Moreover, most subsumption relations between attributes,

values, and relations can be expressed in RDFS with class and property hierarchies. The main loss is the ad-hoc logics of LCA, but there are obvious benefits in using the well-established W3C standards of the Semantic Web (i.e., RDF, RDFS, OWL, SPARQL): e.g., comparability of our work to others' work, reusability of and interoperability with SW tools, availability of datasets.

An important theoretical result of QFS is the proof of the *safeness* and *completeness* of navigation. We informally define safe places and fully safe LISQL expressions. A navigation place is *safe* if its extension is not empty, and a LISQL expression is *fully-safe* if it generates a safe place for every focus. A safe navigation means that every navigation path starting at a safe place leads to a safe place. This implies that users cannot fall into dead-ends by following suggested links. A complete navigation means that every fully-safe LISQL expression can be reached through navigation from the initial place. This implies that users never need to manually edit the query, and can entirely rely on suggested navigation links to express their information needs. This is a significant result given the expressivity of LISQL.

Sewelis[10] (Semantic Web LIS) is our current implementation of QFS, and evolved from Camelis in 2009 (it was first named Camelis 2). It is a desktop application that enables to load and browse RDF files. Its user interface has the same view structure, made of three parts: the query, the extension, and the index. The main differences are a richer query language, and more controls for navigation. A user study was performed by Alice Hermann, and is reported in Appendix C [**Ferré and Hermann, 2011**]. The subjects were students in computer science but not aware of the Semantic Web. The study showed that most students could answer many types of questions on genealogical data after a short training time. There was no problem of *readability* with LISQL, but there was some confusion about the focus. Sewelis has also been applied by Clément Guérin *et al* to the exploration of comicbooks, based on the content of panel images and on metadata [Guérin et al., 2012].

In its most recent developments, Sewelis scales to a few millions triples, which is high enough for many useful applications, but quite small compared to the largest RDF datasets (several billions triples). However, Joris Guyonvarc'h (as an MSc student) demonstrated the *scalability* of QFS up to billions of triples by re-implementing the core of Sewelis as a Web application on top of SPARQL endpoints [**Guyonvarch et al., 2013**]. That was made possible by computing the index on a sample of the extension (e.g., 1000 first results), and because SPARQL engines have been highly op-

[10] http://www.irisa.fr/LIS/softwares/sewelis/

timized. This work was basis for the development of SPARKLIS[11], a SPARQL endpoint explorer that provides all querying features of SEWELIS in a scalable way. SPARKLIS works on one of the largest dataset, DBpedia, which has about 2 billions triples.

Finally, we list the missing SPARQL 1.1 expressivity features in the version of SEWELIS that is presented in Appendix C: multi-dimensional queries (i.e., several variables in the `SELECT` clause), optional graph patterns (only relevant on multi-dimensional queries), solution modifiers (i.e., subset of results, ordering of results), expressions and constraints based on built-in predicates and functions, aggregations, named graphs and external services.

3.5 Updating Through Interaction (UTILIS)

Previous ACN instances only consider queries literally, i.e., the *retrieval* of information. We here consider the *update* of information in a new ACN instance, UTILIS (Updating Through Interaction with Logical Information Systems), which was developed by Alice Hermann during her PhD [Hermann, 2012]. Updating suffers the same trade-off as retrieval between expressivity and usability. In fact, a parallel can be drawn between the different approaches of information retrieval (see Section 2), and the different solutions that exist for information update:

- query languages → update languages, as formal languages or CNL;
- navigation structures → editors of graphs and hierarchies, often through graphical user interfaces;
- interactive views → dialogs based on forms and tables.

Formal update languages like SQL and SPARQL are the most expressive, but also the least usable, exactly like for their retrieval counterpart. Controlled natural languages (CNL), such as GINO [Bernstein and Kaufmann, 2006] or ACE [Fuchs et al., 2006, Kuhn, 2009], are designed to be more readable, and their tools often have an auto-completion mechanism that helps users avoid lexical and syntactic errors. Editors of graphs and hierarchies let users insert one edge at a time, growing an existing structure interactively. For example, Protégé [Noy et al., 2001] is an ontology editor for OWL and RDFS languages, in which users can edit the hierarchy of classes, the hierarchy of properties, and the graph of relationships between individuals. Its use is rather intuitive but mostly for ontology designers who already have a good knowledge of Semantic Web standards. QuiKey [Haller, 2010] requires no

[11] Try it at http://www.irisa.fr/LIS/ferre/sparklis/osparklis.html

such knowledge, and supports the quick creation of chains of triples with an ergonomic auto-completion mechanism. Semantic wikis, such as SweetWiki [Buffa et al., 2008], ACEWiki [Kaljurand and Kuhn, 2013], or Semantic Media Wiki [Völkel et al., 2006], extend well-known wikis with special notations for RDF triples as labelled links. They encourage users to input semantic data along textual data, and use the semantic data for improved querying and navigation. Other systems, like OKM Ontology Management [Davies et al., 2010] or Freebase [Bollacker et al., 2008], use predefined forms that abstract over the structure of semantic data. Forms are well-known, and therefore easy to use, but not flexible, and therefore generally less expressive.

Utilis is defined as an extension of Query-based Faceted Search (QFS), and implemented as an extension of Sewelis [**Hermann et al., 2012**]. It can also be seen as a relational extension of an earlier work about the guided insertion and description of objects in LCA [**Ferré and Ridoux, 2002**]. A key point is that Utilis is an instance of ACN, and can therefore be rendered through the exact same interface as for retrieval. In Sewelis, the distinction between the two modes is indicated by the highlighting color of the focus: green for retrieval mode (existing information), red for update mode (yet to be created information). We now sketch this ACN instance (see paper [**Hermann et al., 2012**] for more details and illustrations):

knowledge base an RDF graph, like in Sewelis.

query language LISQL, like in Sewelis. In update mode, a LISQL expression is understood as an object description. Disjunction and negation are not allowed in such descriptions, because they are not representable semantically in RDF (but they would be representable in OWL).

extension When building the description of a new object, it finally happens that no suggestion fits the new object because no existing object is like it. The *safeness* property is here too strong. Users can relax it by triggering the update mode (red focus). In update mode, the query (i.e., the current description of the object being created) is generalized until results are found. Those results are the existing objects featuring the most similarity with the object being created, i.e. approximate answers. A query is generalized by replacing classes by their super-classes, properties by their super-properties, and by removing sub-expressions.

index The index is defined like for Sewelis, but is based on similar objects in update mode.

links The links of SEWELIS allow the insertion of additional elements to the object description under construction. The insertion of disjunction and negation are not relevant in update mode. The selection of an element can be done by browsing the index, and by auto-completion over the index contents. New elements, such as entities, classes, properties, and literal values, can be created on the fly with ad-hoc widgets (e.g., a calendar for dates).

When a decription is judged as complete by the user, the `Assert` button turns yet-to-be-created information into existing information, and switches from update mode to retrieval mode. In other words, it performs the expected update, and inserts a number of triples into the RDF graph. Conversely, a `Retract` button deletes a number of triples, and switches to the update mode.

We now discuss the advantages of UTILIS, compared to other approaches.

- UTILIS *does not need any preliminary preparation, nor any schema, to give suggestions in the description of objects.* Protégé needs an ontology (e.g., the domain and range of properties) to give any suggestion. Form-based approaches need the preliminary definition of forms by domain experts. CNL approaches require the definition of a lexicon for the application domain. UTILIS uses existing RDF data as a basis for suggestions.

- UTILIS *suggestions are based on individuals rather than on ontology axioms, and are therefore more precise (*specificity*).* For example, after specifying the collection of a comics panel, Protégé suggests all characters of all collections, while UTILIS only suggests the characters of the chosen collection.

- UTILIS *suggestions depend on the full description of the new object.* Other systems, e.g. Protégé, only consider the class of the new object to suggest relevant properties, and then consider the range of properties to suggest relevant values. For example, starting with a film described by its director and an actor, UTILIS would suggest other actors that played for the same director or with the same actor, but Protégé would suggest all actors.

- UTILIS *suggestions are available before any user input (*guidance*).* All systems, except Protégé and ACEWiki, require the user to enter at least a few letters in order to give suggestions. In UTILIS, the index provides immediate suggestions. This is acceptable because suggestions are here more specific, and therefore less numerous. In the case of long lists of

suggestions, or when the user has a clue, QuiKey-like auto-completion offers a powerful way to find the right suggestion.

A user study was performed to compare UTILIS with PROTÉGÉ, in terms of usability. The task assigned to subjects was to describe comics panels about their context and contents: e.g., collection, visible characters, speech bubbles (*who says what to whom*). Subjects found UTILIS more difficult to use, compared to Protégé forms, because of the extra flexibility and novelty of the user interface. However, subjects appreciated UTILIS suggestions and auto-completion, and effectively used them. Finally, the most important result is that UTILIS ensured a better consistency in the produced data, which can be seen as the counterpart of *safeness* in information access. For example, 27 duplicate entities were introduced in Protégé, while none in UTILIS. This means that existing entities are more easily identified and reused in UTILIS than in Protégé. Note that the identification and reuse of resources is what distinguishes five stars linked open data (LOD) from four stars LOD, according to the star scheme of LOD[12].

3.6 Possible World Exploration (PEW)

In all above instances of ACN, there are concrete objects to be returned as the extension of queries. However, it is common in the Semantic Web to start with the design of an ontology that defines classes and properties, logical axioms about them, but no objects (aka. individuals in OWL). OWL semantics is based on Description Logics (DL) [Baader et al., 2003]. From the viewpoint of formal semantics, the axioms of an ontology can be seen as conditions or constraints which a *world* (called an *interpretation* in DL) has to satisfy for matching what the ontology modeler has specified to be "true" in the considered domain. Therefore, an ontology characterizes a set of *possible worlds* (called *models* in DL), which the ontology modeler may wish to explore and modify. Indeed, modeling a domain knowledge as an ontology is an error-prone activity. For example, it has been observed that negative constraints (e.g., "cars are not humans", "animals have no social security ID") are often overlooked because they are less salient for human beings than positive constraints (e.g., "every woman is a person", "every person has a lastname"). Missing constraints entail a discrepancy between what the ontology modeler has in mind, and the formal semantics of the ontology: there are more possible worlds than expected (e.g., a world where "some animal has a social security ID"!).

[12] http://5stardata.info/

In a joint work with Sebastian Rudolph, during his visit at Irisa in March 2012, we derived a new ACN instance from SEWELIS, the Possible World Explorer (PEW[13]), to explore OWL ontologies and enrich them with negative constraints [**Ferré and Rudolph, 2012**]. PEW is very similar to SEWELIS at the syntactic level, but strongly differs at the semantic level because it is based on logical reasoning rather than on query evaluation. We now sketch the principles of PEW as an instance of ACN (see paper [**Ferré and Rudolph, 2012**] for more details):

knowledge base K is an OWL knowledge base, i.e., a set of OWL axioms.

query language A query q is the combination of a *simple* OWL class expression and a focus. Simple OWL class expressions are made of class names, individual names, role names, inverse roles, existential restrictions, conjunction, disjunction, and atomic negation. Those expressions are in fact very similar to LISQL expressions, with the exception that generalized negation is excluded. The difference with LISQL lies in the semantics of queries, i.e., in the definition of the extension and index. The focus is the same as in SEWELIS, and points to a subexpression of the OWL class expression.

extension The extension of the query is simply the yes/no answer to the question "Is the query satisfiable?". A navigation place is safe if the extension is "yes", i.e., if the query is satisfiable. The query is a class expression, and a class expression is *satisfiable* if there exists a possible world (i.e. a DL model) in which that class expression has a non-empty set of instances. A class expression can therefore be seen as a *pattern* that discriminates between possible worlds. The effect of navigation is to refine the query, and hence the pattern, and hence to focus on smaller subsets of possible worlds while maintaining that subset non-empty (safeness).

index The index is the set of *possible adjuncts*. An adjunct is an elementary class expression among: a class name (e.g., "a pizza"), an unqualified existential restriction (e.g., "has some ingredient"), an individual name (e.g., "tomato"), and their negations (e.g., "not a pizza", "has no ingredient", "not tomato"). A *possible adjunct* is an adjunct that, when inserted at the focus of a satisfiable query, produces another satisfiable query. In other words, possible adjuncts ensure navigation safeness by avoiding users to reach unsatisfiable queries. A possible adjunct is also

[13] We adopt the Semantic Web practice of flipping letters in acronyms.

a *necessary adjunct* if its negation is not a possible adjunct (e.g., "has an ingredient" for a pizza).

links The navigation links are a subset of those in SEWELIS: moving the focus, inserting an element of the index, inserting a disjunction, and deleting the sub-expression at the focus.

We have proved the *safeness* and *completeness* of navigation in PEW. This means that navigation only leads to satisfiable queries, and that every satisfiable query is reachable by navigation. In other words, the navigation space is made of all patterns on possible worlds that can be expressed as a simple OWL class expression, and that are satisfiable in at least one possible world of the ontology. Whenever the ontology modeler finds a satisfiable query that should not be satisfiable according to her domain knowledge, a negative constraint can be added to the ontology. The effect of that additional constraint is that the unexpected possible worlds are erased from the set of possible worlds. The negative constraint simply states that the current class expression C must have an empty set of instances ($C \sqsubseteq \bot$ in DL).

PEW is implemented on top of OWL API[14] for handling ontologies, and on the HermiT reasoning engine [Shearer et al., 2008] for checking the satisfiability of class expressions. It reuses the GUI of SEWELIS. For immediate readability by ontology modelers, class expressions (the query and adjuncts) are displayed using standard OWL notations: both the DL notation and the Manchester syntax are available. We experimented PEW on the well-known pizza ontology. This ontology has the advantage of being well-known, covering a large subset of OWL constructs, and being representative of OWL ontologies. While the pizza ontology is often referred to and was subject to a number of refinements, we found in our exploration a number of unexpected possible worlds, and hence missing axioms. For example, we found it is possible to have a vegeterian pizza that contains meat or fish as an ingredient, despite an apparently explicit definition of what is a vegeterian pizza! The explanation is that a vegeterian pizza was defined as not having meat or fish as *topping*, but not all ingredients are toppings, opening the possibility of meat in other kinds of ingredients (e.g., the base).

PEW is user-centered, while most approaches to the discovery of missing negative constraints are automated. Automated discovery is valuable, but is not entirely reliable, and results need to be inspected manually anyway. Moreover, those approaches are generally restricted to disjointness, the simplest form of negative constraints. Another user-centered approach to ontology completion is based on Formal Concept Analysis (e.g.,

[14] http://owlapi.sourceforge.net/

[Völker and Rudolph, 2008]), but it tends to patronize the ontology modeler by forcing her to answer a prescribed row of questions. In contrast, PEW lets users freely explore the space of possible worlds. However, it provides no strategy to cover that space.

3.7 An Expressive CNL for the Semantic Web (SQUALL)

This last contribution, SQUALL [**Ferré, 2012**, **Ferré, 2013**], is only a partial instance of ACN because indexes and navigation links are not defined, like in query language approaches. SQUALL stands for *Semantic Query and Update High-Level Language*, and is a CNL for querying and updating RDF datasets (see Appendix D for more details). The knowledge base is therefore the same as for SPARQL and SEWELIS. The ACN extensions (i.e., query results) are the same as in SPARQL, i.e., tables of results for `SELECT`-queries, and are more general than the lists of results in SEWELIS. Compared to SPARQL, SQUALL has a very similar *expressivity*, and a better *readability*. Indeed, SQUALL covers all major features of SPARQL, and only misses a few minor features: e.g., n-ary `DESCRIBE`-queries, regular expression modifiers. Unlike SPARQL, SQUALL sentences are immediately readable because its grammar is very close to that of English (see examples in Appendix D), while SPARQL is a formal language that exhibits low-level notions such as logic, relational algebra, and a lot of variables. Compared to SEWELIS, SQUALL is at the same time more expressive and more readable than LISQL, SEWELIS' query language. The advantage of SEWELIS is to provide feedback and guidance through navigation. A major objective for future work is then to make SEWELIS and SQUALL converge into a system as expressive as SPARQL, as readable as English, and as guided as SEWELIS (see a more thorough discussion in Section 4).

The syntax and semantics of SQUALL are defined with a Montague grammar [Montague, 1970]. The semantics of a sentence is represented in a logical intermediate representation, which is then translated to a target language, here SPARQL. Therefore, existing SPARQL engines (e.g., Virtuoso) and SPARQL endpoints (e.g., DBpedia) can be used to efficiently execute queries and updates. We have implemented a Web application SQUALL2SPARQL[15] that translates SQUALL queries and updates to SPARQL, and send them to SPARQL endpoints. The Web site also provides a number of examples, in particular the 200 questions from the QALD-3/DBpedia[16] chal-

[15] http://lisfs2008.irisa.fr/ocsigen/squall/
[16] QALD: Question Answering over Linked Data

lenge [Cimiano et al., 2013], reformulated in SQUALL. The results of our participation to the challenge can be found at http://www.clef2013.org/. SQUALL has a full coverage of QALD questions, and a very good precision and recall (F-measure = 0.90), but it relies on a manual reformulation of questions. Sentences are generally natural at the syntactic level, but the main limitation to readability is that there is so far no lexical treatment. This means that URIs, generally in the form of qualified names (e.g. `dbo:Film`), must be directly used as words. The advantage is that SQUALL is directly applicable to any SPARQL endpoint, without requiring the costly design of a domain-specific lexicon. However, an objective is to make the use of such lexicons possible when available.

Many CNLs have been defined in the past decades [Kuhn, 2013], but SQUALL is the only one that targets SPARQL queries *and* updates. Most CNLs for the Semantic Web rather target ontological axioms (e.g., ACE [Fuchs et al., 2006], SOS and Rabbit [Schwitter et al., 2008]). A few CNLs target SPARQL queries (e.g., Ginseng [Bernstein et al., 2005], Aggrego Search [Smits et al., 2013b, Smits et al., 2014]) but cover a limited fragment of SPARQL, typically tree patterns, and some filters. A number of systems have been developed to query semantic data in free natural language (see Section 2.1). Their advantage is that they do not put constraints on the user, contrary to CNL-based systems. They use NLP tools, external resources (e.g., WordNet), and ontologies to try and translate spontaneous questions to SPARQL. Because of ambiguity and synonymy in natural languages, those systems achieve a much lower precision and recall (at QALD-3, CASIA got the best result with an F-measure equal to 0.36). Their main limitation to this date is about expressivity as they can only answer the simplest questions: 1-2 triples, counting, only lists of results, no tables, no comparatives, no superlatives, no disjunctions, no negations, no expressions. In fact, SQUALL occupies a new trade-off in the CNL community. Traditionnaly, CNLs have developed syntactic coverage in as much as semantics follows. With SQUALL, we consider RDF/SPARQL as a target semantics, and we give it a CNL syntax.

4 Conclusion and Perspectives

Abstract Conceptual Navigation (ACN) is a new paradigm of information access that merges the fundamental concepts of query languages (e.g., SPARQL), navigation structures (e.g., file hierarchies, the Web), and interactive views (e.g., faceted search, OLAP). Figure 3 shows the progress achieved by the most recent ACN instances, QFS (Query-based Faceted Search) and

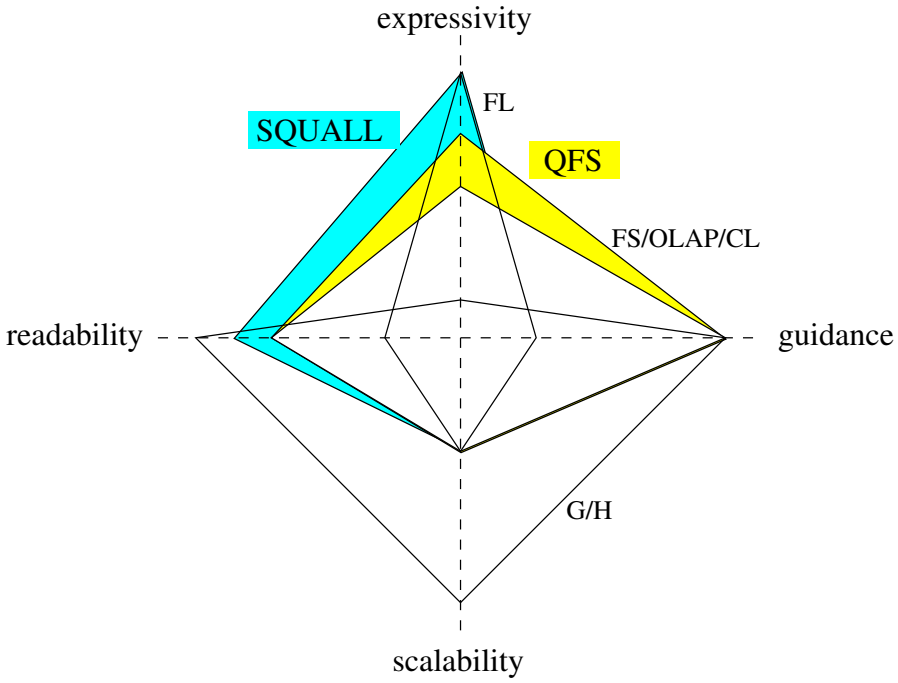


Figure 3: Radial diagram showing the contribution of ACN instances QFS (Query-based Faceted Search) and SQUALL.

SQUALL, compared to existing approaches (see Figure 1 on page 27). QFS inherits the *usability* (*guidance* and *specificity*) of faceted search, which is *de facto* recognized by its adoption in e-commerce websites and others, because QFS is based on the same interaction model. Users only have to make choices among system suggestions (ACN navigation links) in order to iteratively refine a set of results (ACN extension). Like faceted search, QFS supports both classical search where users have a precise idea of what they want, and exploratory search [Marchionini, 2006] where users do not have a precise aim, and rather want to get a global understanding of a dataset. The definition of navigation places by queries instead of selections of objects open the way to very high *expressivity*, compared to classical faceted search. A key ingredient to such an expressivity is the *query focus*, which defines at the same time: (1) a subexpression of the query, (2) an insertion point in the query for query refinements, and (3) a point of view on query results. While QFS does not yet reach the full expressivity of SPARQL, I am confident that that objective is reachable in the medium term, even though each new feature requires an astute combination of query language theory and user interaction design. To conclude, ACN seems to be a generic and effective solution to the combina-

tion of expressivity and usability, which has been the main objective of our research.

The main challenges for ACN regard the properties of readability and scalability. The *readability* challenge, i.e. the user ability to clearly understand the meaning of queries, indexes, and navigation links, increases with expressivity. I think that the use of natural language (NL) is unavoidable to this purpose because it is the only expressive language that everybody knows, and because every other expressive language (e.g., maths, programming languages) is only known by a minority of specialists. A big advantage of ACN in this challenge is that NL understanding is not necessary, unlike in Question-Answering (QA) systems, and instead, NL generation is used to render queries and query refinements. Indeed, NL generation can use a subset of natural language without reducing understandability by people, while NL understanding must cope with full natural language, including syntactic and lexical errors, ellipsis, etc. In this respect, Controlled Natural Languages (CNL) are interesting because they embody a formal language into a subset of a natural language, and this is why I started studying them recently. Figure 3 shows that SQUALL reaches the full expressivity of SPARQL while improving the readability compared to formal query languages (FL) and interactive views (FS/OLAP).

The *scalability* challenge has two sides. The head side concerns the possibly large sets of system suggestions (i.e., large indexes) users may have to choose from. Large indexes are unavoidable when datasets get as large as DBpedia. For exploratory search, indexes should be ranked so as to show first the most *important* elements for the dataset: e.g., the most frequent, the most consistently used. For directed search, intelligent auto-completion, e.g. based on NL understanding, could be used to locate index elements from keywords. The tail side concerns the computation of indexes and extensions over large datasets. While Query-based Faceted Search has been shown to scale to large datasets like DBpedia (about 2 billion triples), the future of semantic search is *federated search*, i.e. search over several datasets and SPARQL endpoints. Some SPARQL endpoints are already quite optimized, but they have only been optimized to the computation of query results, not query refinements. There is probably an important source of improvement there.

4.1 Theoretical Perspectives

We here sketch a number of theoretical perspectives that aim to improve instances of ACN w.r.t. expressivity, readability, guidance, and scalability. They are roughly ordered from short-term to long-term perspectives. Most

of them are deemed to be integrated in Sewelis.

Analytical queries. Analytical queries are typically offered by OLAP and used in Business Intelligence (BI) (see page 22), whose principles have been integrated into Abilis (see page 34), as an extension of Camelis. Sewelis does not yet support such queries, despite the fact that they can be expressed in SPARQL 1.1 thanks to multi-dimensional queries and aggregation operators. Therefore, extending Sewelis to analytical queries can be done by increasing Sewelis' expressivity and guidance to cover those SPARQL features. ACN here provides an original approach that consists in performing analytical queries directly against RDF graphs, while existing approaches in the Semantic Web generally propose to first extract data cubes from RDF graphs, and then apply the OLAP approach to those cubes [Kämpgen and Harth, 2011]. The problem with the latter approach is that domain experts rely on SW experts for the extraction of data cubes, and that it is difficult to anticipate all useful data cubes. Indeed, because of their relational nature, RDF graphs can be the source of many different data cubes, playing as different points of view on data. In our approach, the additional SPARQL features would translate to new query constructs, and OLAP operators would translate to new navigation links. In this way, conceptual navigation would smoothly combine selections, the choice of dimensions and measures, and aggregations.

Visualization of results. On the user interface side, analytical queries often come with visualization components: e.g., tables, charts (e.g., histograms, pie, scatter plot), maps, timelines, and combinations of those. Visualization has already been experimented by P. Allard in Abilis. Our objective is to adapt and improve it for semantic data in Sewelis. There is a lot of interest for the visualization of semantic data, and we aim to contribute with a more systematic way to build complex visualizations from simpler components. ACN principles could be used to suggest the relevant visualization components, and compose them. A difficulty is that a *good* default visualization must be defined at each navigation place so that the system can present the current results, and so as to save as many visualization choices as possible to the user.

Computations in queries. A current limit of Sewelis (and Camelis) is that it can only return values that are explicitly present in data, not computed values. For example, one may want to retrieve the age of people, computed as the duration from their birth date to the current day, or

the distance between places, computed from their geographical coordinates. Of course, ages and distances could be computed in a pre-processing stage and inserted into the dataset as additional triples, but this has two major drawbacks. The first drawback is synchronization with updates (e.g., every day for ages, every time a coordinate is added or modified for distances). The second drawback is that it is impossible to anticipate all computation needs, and it can be costly to store computed information (e.g., the number of distances is quadratic in the number of places). Fortunately, SPARQL 1.1 enables to return computed values through expressions. The standard supports basic computations on numbers and strings, and some SPARQL engines extend them to time and space (e.g., Strabon [Kyzirakos et al., 2012]). Our objective is to extend Sewelis' guidance to the interactive construction of expressions. The main difficulty is that an operation can only be evaluated when all its operands are defined, which makes the usual computation of ACN indexes from results impossible at some navigation places (e.g., what to suggest for the second operand of an addition?). Our idea is to resort to datatypes and operand signatures in those cases to suggest relevant refinements.

Rules. When an expression is often needed, e.g. the age of people, it would be useful to define it once for all as a function, instead of re-building it every time. Such a definition can be seen as a rule saying that: "if a person X has a birth date D, then the age of X is the number of years from D to now". Such rules exist under different forms in various domains such as: databases (triggers, business rules), logic programming and Datalog, knowledge bases, the Semantic Web (e.g., SWRL), dataflow programming and spreadsheets (formula cells). The different kinds of rules differ in what they can do (e.g., infer new facts, perform updates, trigger actions), their semantics and decidability of inference, the kind of inference (e.g., forward chaining vs backward chaining), and the cost of inference and synchronization with updates and events. Therefore, a major difficulty is the choice of a suitable kind of rules for the Semantic Web in general, and for Sewelis in particular. However, whatever kind of rules is considered, ACN could contribute to their authoring by domain experts. A rule can generally be seen as the combination of a query (e.g., "Which person X has which birth date D?") and an update (e.g., "the age of X is the number of years from D to now"), where the query corresponds to the rule premise, and the update corresponds to the rule conclusion. Given that ACN has been instantiated for both queries (Sewelis, Section 3.4), and updates (Utilis, Section 3.5), an objective is to combine the two ACN instances more closely to allow for

the authoring of rules.

NL verbalization of ACN queries. With ACN queries being more and more expressive, it is important to find better and better NL verbalizations of the constructed queries. The task of translating from a formal language to a natural language is known as *natural language generation*, and is the opposite of *natural language understanding*. For recall, NL understanding is not necessary in the ACN approach because queries are constructed in an interactive and guided manner. NL verbalization has been done for SPARQL [Ngomo et al., 2013], for instance. In short, NL verbalization requires to map constructs of the query language to grammatical structures, and to map URIs to words. The former can be done once for each target natural language, using the grammar of a controlled natural language, e.g. SQUALL. The latter, however, requires the definition of a lexicon for each vocabulary, ontology or dataset. A meta-vocabulary, Lemon [McCrae et al., 2011], has been designed for the description of such lexicons but, unfortunately, those are rarely available. An alternative to the manual edition of lexicons is to learn them from a corpus [Walter et al., 2013], which was the second task of the QALD 2013 challenge [Cimiano et al., 2013]. Our first objective is to make use of lexicons, when they are available, to improve the NL verbalization of ACN queries. Our second objective is to consider the edition of lexicons along with the edition of RDF data in UTILIS. Indeed, it should be possible and economical to ask users for the NL form whenever they create a new class or property. At the extreme, we could imagine that users never see or specify a URI, but only NL forms from which the system generates new URIs or retrieves existing URIs.

Machine learning. UTILIS is already a form of machine learning because it tries and predicts relevant elements to be added to the current description of an object, at each step of the conceptual navigation. More precisely, it is a form of instance-based machine learning, like the nearest neighbours approach, because it relies on similar objects to produce such elements. However, it has a number of characteristics that makes it difficult to transpose existing results from the machine learning domain:

- training data is highly relational (the RDF graph),
- there is no fixed target as every entity, class or property is a potential target,
- the type of instances depends not only on the object being described,

but also on the current focus in the current description, which may change at every navigation step,

- the user is tightly integrated to the incremental learning process, and there is no separation between a training phase and an exploitation phase.

A number of FCA-based approaches use concept lattices as the basis for knowledge discovery and machine learning [Fu et al., 2004]. Among them, NAVIGALA [Visani et al., 2011] is particularly interesting because it can be used for incremental learning, where only small parts of the concept lattice need to be computed on-demand. However, it only works on propositional data, and not on relational data. Inductive Logic Programming (ILP) [Muggleton and Raedt, 1994] works on relational data but requires a target and negative examples to work, and is computationally costly, even for a single target, on large data. I am already familiar with ILP because during my postdoc at the University of Wales, Aberystwyth, Ross D. King and I made contributions to ILP using the LCA notion of logic, and applied them to functional genomics [**Ferré and King, 2005**, **Ferré and King, 2006**]. Case-Based Reasoning (CBR) [Aamodt and Plaza, 1994] does not require a fixed target but usually needs a clear separation between *problems* and *solutions*, i.e., between known facts and predicted facts. We are interested in studying the potential cross-fertilizations between our approach and existing work in machine learning, in particular FCA-based approaches, ILP, and CBR.

Exploring the immaterial. All instances of ACN so far, except Pew, have their extensions based on concrete instances, explicitely represented and stored. Those instances are objects in Camelis, and RDF terms in Sewelis. The originality of Pew is to define its extensions as the existence of possible words, which are never explicitly represented, and even less stored. Keeping instances implicit is a necessity when the domain of instances is untractably large or infinite. This is a common situation in computer science, and it would be interesting to investigate the application of ACN to them. We here list a few examples, paraphrasing the sentence "objects as instances of queries":

- models as instances of logical theories (logic semantics),
- execution traces as instances of programs (program semantics),
- (frequent) patterns as instances of a pattern language (e.g., association rules) over a dataset (data-mining),

- solutions as instances of a set of constraints (constraint satisfaction problem).

In each case, instances would not be computed *a priori* as a concrete dataset to explore, but either their existence would be proved, like in Pew, or finite excerpts of them would be computed on-the-need. In each case, several approaches could be followed. For example, in the case of programs and execution traces, the program could be fixed with the objective to explore the possible execution traces by adding constraints such as input data, branching decisions. This could be useful for program understanding and debugging. Alternately, the objective could be the guided construction of a program, where excerpts of execution traces would provide feedback and guidance on the refinement of the program under construction. This would allow for coding and testing at the same time. The challenge of "exploring the immaterial" is that the computation of ACN extensions and indexes relies on reasoning rather than retrieval. This reasoning can easily get untractable or even undecidable. Two practical choices are to use decidable fragments (e.g., Pew is based on the decidable logic of OWL-DL), and to resort to approximate reasoning.

4.2 Applicative Perspectives

The scope of ACN applications is very large because ACN potentially applies to every situation where there is a need for user-centered information access and edition. In this section, we limit ourselves to applications that have already been discussed in the LIS team. For each application, the main actors (mostly LIS team members) are indicated in square brackets. Some of those applications are already well-advanced, while others are yet purely prospective. They are roughly ordered from most advanced to least advanced.

SPARQL endpoint explorer [Sébastien Ferré]. RDF datasets are generally made public both as RDF dumps and SPARQL endpoints. To explore those datasets, Sewelis could in principle be used by loading the RDF dumps. However, this loading process is tedious for users, and Sewelis does not scale to the largest RDF datasets. The limit is about a few million triples, compared to a few billion triples for the largest datasets such as DBpedia. If users want to frequently switch between datasets, those difficulties get even worse. The advantage of SPARQL endpoints is that the loading process has been done once for all on a server, and that they are ready to answer any question in the form of SPARQL queries. Moreover, SPARQL endpoints are based on SPARQL query engines, some of which have been highly optimized

(e.g., OpenLink Virtuoso[17]). Our objective is therefore to re-implement the logic and user-interface of SEWELIS on top of SPARQL endpoints. The main difficulty is that we do not control the API of SPARQL endpoints, which has not been designed for the incremental building of queries, and the return of rich feedback. Work on this application has been initiated by Joris Guyonvarc'h during his MSc thesis [**Guyonvarch et al., 2013**]. It demonstrated the feasability and scalability of the approach, which is based on the automatic generation of SPARQL queries at each navigation step. Note that this is transparent for users who do not ever see any SPARQL query. We continue work on this application with a new prototype, SPARKLIS[18], with a first objective to cover all search functionalities of SEWELIS. The advantage of this new prototype is to depend only on SPARQL endpoints, and to run entirely on the client browser as a Javascript application. Future objectives would be: (1) to improve the user interface with visualization (e.g., timeline, maps, charts), (2) to improve the NL verbalization of queries, and (3) to address federated search, i.e. exploration over several SPARQL endpoints at the same time.

Group Decision and Negotiation [Mireille Ducassé, Peggy Cellier]. Group Decision Support Systems (GDSS) are designed to help a group of decision makers to collectively reach an agreement on a decision problem. The general setting is that there are a number of alternative choices, which are described according to a number of criteria. The decision problem is to choose one of the alternatives, and one of the main difficulties is precisely that there are several criteria (multicriteria decision), which are not directly comparable. In a group, different persons may give different weights to the different criteria. Another problem is the potentially large set of alternatives and criteria, which leads to an information overload. The data associated to a decision problem, a binary relation between alternatives and criteria, has the shape of a logical context, and is therefore amenable to the application of ACN, and in particular CAMELIS. The use of CAMELIS has already been experimented on group decision use cases: an academic recruitment process [**Ducassé and Ferré, 2008**], and a committee to validate students' year at a technical university [Ducassé and Cellier, 2014]. CAMELIS was used for its capability:

- to select subsets of alternatives (ACN extension) according to logical combinations of criteria (ACN query),

[17] http://virtuoso.openlinksw.com/
[18] http://www.irisa.fr/LIS/ferre/sparklis/osparklis.html

- to update on the fly the description of alternatives without interrupting the decision process,
- to define rules that generate derived criteria or decisions.

The experiments have shown that using CAMELIS as a GDSS guarantees more fairness and speed because it enables to iterate on criteria rather than on each alternative in turn. Moreover, the queries and rules built in the decision process enable to keep track of all decisions and their motivations, and help decision makers to collectively take responsability of sensitive decisions. However, those experiments have a number of limits that require tool development and more experiments to be done. First, CAMELIS was not yet used in the actual meetings, but in a replay of them with some of the concerned actors. Second, those meetings were controversial only to a limited extent, and it remains to evaluate the approach on more sensitive and controversial multicriteria decisions. Finally, it will be necessary to design, implement, and evaluate a user interface dedicated to group decision, rather than the generic user interface of CAMELIS. Indeed, this interface constitutes the view shared by all decision makers, and therefore plays a crucial role in the decision process.

Workflow composition for biologists [Mouhamadou Ba, Mireille Ducassé, Sébastien Ferré]. Biologists face more and more data (e.g., sequences, annotations), and more and more services to process those data (e.g., Blast for sequence search, ClustalW for sequence alignment). However, those services perform elementary tasks, and need to be combined into workflows to address real biological problems [Romano, 2008, Barker and van Hemert, 2007]. Designing workflows is a form of programming where services play the role of high-level instructions. In fact, bioinformatic workflows are often written in script languages, where the script specifies the connections between services, and hence the dataflow. Note that bioinformatic services are essentially functional in that they have no inner state, and therefore return the same outputs for the same inputs. The connection with ACN is that there is the same trade-off about workflows as about queries. Programming languages and script languages are a very expressive and generic way to define workflows, but they are too low-level for widespread use by biologists. A number of graphical editors have been proposed to alleviate this difficulty. They are similar to syntactic editors for queries, and let users drag-and-drop services and connections between services. However, those graphical editors are limited to the simplest control structures (chaining, conditional) compared to programming languages (e.g.,

iterations, exceptions, higher-order, backtracking). Moreover, users are generally not guided in the choice of services and connections. Our objective in this application is to apply the ACN approach to the design of bioinformatic workflows. This objective can be split in two main tasks. The first task is to define a workflow language (ACN query language) that is at the same time expressive and high-level, similarly to LISQL in Sewelis or Squall. This language will be the starting point for, on one hand, the generation of an executable workflow in a target programming or script language (ACN extension), and on the other hand, the generation of a graphical representation when it is prefered to a textual representation. The second task is to define for each focus of a workflow under construction which services can be inserted (ACN index). The main constraint on service insertion is the type of service inputs and outputs. Because those constraints are based on intentional knowledge (type signature of services) rather than on extensional knowledge (e.g., workflow execution traces), this application is yet another example of the above theoretical perspective "Exploring the immaterial". Mouhamadou Ba has worked on this application since September 2012, as his PhD work [Ba, 2013], under supervision of Mireille Ducassé and myself, and with the support of the GenOuest bioinformatic platform led by Olivier Collin. He has so far worked on the second task, which is now well-advanced [**Ba et al., 2014**]. Given two services S_1 and S_2, we can decide whether an output of S_1 can be connected to an input of S_2 based on the respective types of the output and the input. To address data heterogeneity, the relation between types that we use is not equality or subtyping, as in existing work, but convertibility, i.e. whether the output can be converted to the input. Convertibility is decided automatically from the structural description of types, and actual converters can be generated along with the decision process. This will potentially save a lot of effort to workflow designers who often have to implement ad hoc converters for each workflow.

User-centered mining of geographical data [Soda M. Cissé, Olivier Ridoux, Peggy Cellier, Erwan Quesseveur]. Data mining techniques are used to extract knowledge from large datasets. The extracted knowledge generally takes the form of a set of *patterns*, where each pattern represents a statistically significant regularity in the data. A common problem is that, for combinatorial reasons, the set of patterns is too large for systematic manual inspection, and sometimes even larger than the initial dataset itself. Existing approaches tackling this problem generally propose a pruning of the set of patterns, e.g. by applying user-defined constraints, or by selecting the top-k patterns based on some ranking. There is again a trade-off between

expressivity, and usability. On one hand, a top-k approach based on some ranking is easy to use, and can return as few patterns as wanted. However, it provides no freedom in the selection of patterns as the ranking is the same for all users and datasets. On the other hand, expressive constraint languages provide freedom in the selection of small subsets of patterns but may be difficult to use. This is similar to the difference between search engines, based on ranking, and structured query languages, like SQL or SPARQL. The difference is that the searched objects are not items *in* data, but patterns *about* data. Our objective is here to put users *in the loop* by applying the ACN approach, thus allowing them to interactively explore a large set of patterns, and striving to reconcile expressiveness and usability. Inductive Databases (IDB) [de Raedt, 2002] could be a good starting point as their aim is to lift the database approach to data mining by designing powerful pattern query languages and their efficient evaluation by computing patterns on demand. Results about IDB could provide definitions for ACN queries and extensions, and the remaining challenge would be to define ACN indexes, i.e. relevant pattern contraints to filter out the current selection of patterns. The possibility to compute patterns on demand indicates that this application can be seen as an instance of the above perspective "Exploring the immaterial". We have done a first application of CAMELIS to the exploration of a set of sequential patterns, where the partial ordering over patterns was used to express constraints [**Cellier et al., 2011**]. The objective of the PhD of Soda Marème Cissé, supervised by Olivier Ridoux and Peggy Cellier, is to study this approach on the mining of sequential patterns over geographical data, in particular GPS tracks, in order to discover behaviour patterns. This work is led in collaboration with Erwan Quesseveur, a researcher in geography from University Rennes 2, who provides data from real social studies.

Memory prosthesis [Olivier Ridoux, Sébastien Ferré]. The capacity and persistency of digital memories have long suggested to use it to extend human memory, and palliate its deficiencies, hence the term of *memory prosthesis*. Many models of memory prostheses have been proposed, from the early Memex of Vannevar Bush [Bush, 1945] to the more recent MyLifeBits at MicroSoft Research [Gemmell et al., 2006]. Lamming et al. [Lamming et al., 1994] identify a number of guidelines for the design of a human memory prosthesis: (a) sensing of the environment, (b) automatic data capture, (c) manual data capture, note taking, (d) focusing on relevant information for retrieval, (e) easy to use, (f) available where needed, (g) integrated with other applications, (h) reliable and fail-safe, (i) respectful of privacy. Today, the availability of wearable computers (smartphones) provides

material support for guidelines (a), (b), and (f). Lifelogging applications offer automatic data capture (b) such as GPS tracks, health monitoring. We think the ACN approach can contribute to the guidelines about information retrieval (d), manual data capture (c), and ease of use (e). SEWELIS already fulfills the three aspects in an integrated fashion. The first challenge is to make it even simpler to use so that it can be used on a smartphone, i.e. on a small screen with no physical keyboard. The second challenge is to integrate it with automatic data capture, and sensing of the environment (e.g., GPS position). A difficulty here is that automatically captured data tends to be numeric (e.g., GPS coordinates, temperature), while manually captured data tends to be symbolic (e.g., place name, hot/warm/cold), and it is necessary to reconcile them for effective retrieval. A third challenge is to integrate the memory prosthesis with other applications (guideline (g)). First, other applications can be a source of information (e.g., calendar events, emails); and second, other applications are helpful to visualize some contents (e.g., calendar events, videos), or to act on them (e.g., to send by email). Lamming et al. also insist on the *prospective memory* in addition to the *retrospective memory*. The former is concerned with anticipation and the recall of events or things to do, and it should trigger recalls based on the user context, hence the importance of sensing the environment. Therefore, a fourth challenge is to extend SEWELIS so that it can initiate interaction with the user (e.g., trigger recalls) based on explictly stated recalls (todo items) or on previous behaviour (lifelogging).

Bibliography

[Aamodt and Plaza, 1994] Aamodt, A. and Plaza, E. (1994). Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59. Cited on page 54.

[Allard, 2011] Allard, P. (2011). *Logical modeling of multidimensional analysis of multivalued relations - Application to geographic data exploration.* PhD thesis, Thèse de l'Université de Rennes 1 - École doctorale MATISSE. supervised by S. Ferré and O. Ridoux. Cited on pages 34 and 37.

[Allard et al., 2010] Allard, P., Ferré, S., and Ridoux, O. (2010). Discovering functional dependencies and association rules by navigating in a lattice of OLAP views. In Kryszkiewicz, M. and Obiedkov, S., editors, *Concept Lattices and Their Applications*, pages 199–210. CEUR-WS. Cited on page 37.

[Amato and Meghini, 2008] Amato, G. and Meghini, C. (2008). Faceted content-based image retrieval. In Sacco, G., editor, *DEXA Work. Dynamic Taxonomies and Faceted Search (FIND)*, pages 402–406. IEEE Computer Society. Cited on page 20.

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). The expressive power of SPARQL. In *et al*, A. P. S., editor, *Int. Semantic Web Conf.*, LNCS 5318, pages 114–129. Springer. Cited on page 10.

[Ba, 2013] Ba, M. (2013). Guided composition of tasks with logical information systems - application to data analysis workflows in bioinformatics. In Cimiano, P., Corcho, O., Presutti, V., Hollink, L., and Rudolph, S., editors, *Extended Semantic Web Conf.*, LNCS 7882, pages 661–665. Springer. Cited on page 58.

[Ba et al., 2014] Ba, M., Ferré, S., and Ducassé, M. (2014). Generating data converters to help compose services in bioinformatics workflows. In *Int. Conf. Databases and Expert Systems Applications (DEXA)*. Springer. To appear. Cited on page 58.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press. Cited on page 44.

[Barker and van Hemert, 2007] Barker, A. and van Hemert, J. I. (2007). Scientific workflow: A survey and research directions. In Wyrzykowski, R., Dongarra, J., Karczewski, K., and Wasniewski, J., editors, *Int. Conf. Parallel Processing and Applied Mathematics*, LNCS 4967, pages 746–753. Springer. Cited on page 57.

[Bars et al., 2002] Bars, S., Ferré, S., and Ridoux, O. (2002). Logic functors for types as search keys. In *Int. Workshop on Isomorphisms of Types*. Cited on page 31.

[Bedel, 2009] Bedel, O. (2009). *GEOLIS : Un Système d'information logique pour l'organisation et la recherche de données géolocalisées*. PhD thesis, Thèse de l'université de Rennes 1. coencadrée par O. Ridoux et S. Ferré. Cited on page 33.

[Bedel et al., 2008a] Bedel, O., Ferré, S., and Ridoux, O. (2008a). Handling spatial relations in logical concept analysis to explore geographical data. In Medina, R. and Obiedkov, S., editors, *Int. Conf. Formal Concept Analysis*, LNAI 4933, pages 241–257. Springer. Cited on page 33.

[Bedel et al., 2012] Bedel, O., Ferré, S., and Ridoux, O. (2012). GEOLIS: a logical information system to organize and search geo-located data. In Bucher, B. and Ber, F. L., editors, *Innovative Software Development in GIS*, Geographical Information Systems Series, pages 151–188. Wiley. Cited on page 33.

[Bedel et al., 2008b] Bedel, O., Ferré, S., Ridoux, O., and Quesseveur, E. (2008b). GEOLIS: A logical information system for geographical data. *Revue Internationale de Géomatique*, 17(3-4):371–390. Cited on page 33.

[Bedel et al., 2006] Bedel, O., Ridoux, O., and Quesseveur, E. (2006). Combining logical information system and OpenGIS tools for geographical data exploration. In *Int. Conf. Free and OpenSource Sofware for Geoinformatics*. Cited on page 33.

[Belleannée et al., 1999] Belleannée, C., Brisset, P., and Ridoux, O. (1999). A pragmatic reconstruction of λprolog. *The Journal of Logic Programming*, 41:67–102. Cited on page 32.

[Berners-Lee et al., 1992] Berners-Lee, T., Cailliau, R., Groff, J.-F., and Pollermann, B. (1992). World-Wide Web: the information universe. *Internet Research*, 2(1):52–58. Cited on page 16.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*. Cited on page 9.

[Bernstein and Kaufmann, 2006] Bernstein, A. and Kaufmann, E. (2006). GINO - a guided input natural language ontology editor. In et al., I. F. C., editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 144–157. Springer. Cited on page 41.

[Bernstein et al., 2005] Bernstein, A., Kaufmann, E., and Kaiser, C. (2005). Querying the semantic web with Ginseng: A guided input natural language search engine. In *Work. Information Technology and Systems (WITS)*. Cited on pages 15 and 48.

[Bollacker et al., 2008] Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In *ACM SIGMOD Int. Conf. Management of data*, pages 1247–1250. ACM. Cited on page 42.

[Bosc and Pivert, 1995] Bosc, P. and Pivert, O. (1995). SQLf: a relational database language for fuzzy querying. *Transactions on Fuzzy Systems*, 3(1):1–17. Cited on page 15.

[Buffa et al., 2008] Buffa, M., Gandon, F., Ereteo, G., Sander, P., and Faron, C. (2008). Sweetwiki: A semantic wiki. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):84–97. Cited on page 42.

[Bush, 1945] Bush, V. (1945). As we may think. *The atlantic monthly*, 176(1):101–108. Cited on pages 16 and 59.

[Cabrio et al., 2012] Cabrio, E., Cojan, J., Aprosio, A. P., Magnini, B., Lavelli, A., and Gandon, F. (2012). QAKiS: an open domain QA system based on relational patterns. In Glimm, B. and Huynh, D., editors, *Int. Semantic Web Conf. (Posters & Demos)*, volume 914 of *CEUR Workshop Proceedings*. Cited on page 14.

[Carpineto and Romano, 1996] Carpineto, C. and Romano, G. (1996). A lattice conceptual clustering system and its application to browsing retrieval. *Machine Learning*, 24(2):95–122. Cited on page 18.

[Cellier et al., 2011] Cellier, P., Ferré, S., Ducassé, M., and Charnois, T. (2011). Partial orders and logical concept analysis to explore patterns extracted by data mining. In *Int. Conf. on Conceptual Structures for Discovering Knowledge*, pages 77–90. Springer. Cited on page 59.

[Chailloux et al., 2000] Chailloux, E., Manoury, P., and Pagano, B. (2000). *Developping Applications with Objective Caml*. O'Reilly. Cited on page 32.

[Chaudron and Maille, 2000] Chaudron, L. and Maille, N. (2000). Generalized formal concept analysis. In Mineau, G. and Ganter, B., editors, *Int. Conf. Conceptual Structures*, LNCS 1867. Springer. Cited on page 31.

[Cimiano et al., 2013] Cimiano, P., Lopez, V., Unger, C., Cabrio, E., Ngomo, A.-C. N., and Walter, S. (2013). Multilingual question answering over linked data (QALD-3): Lab overview. In Forner, P., Müller, H., Paredes, R., Rosso, P., and Stein, B., editors, *Information Access Evaluation. Multilinguality, Multimodality, and Visualization - Int. Conf. CLEF Initiative*, LNCS 8138, pages 321–332. Springer. Cited on pages 48 and 53.

[Codd et al., 1993] Codd, E., Codd, S., and Salley, C. (1993). *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate*. Codd & Date, Inc, San Jose. Cited on page 22.

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387. Cited on pages 9 and 22.

[Corby et al., 2006] Corby, O., Dieng-Kuntz, R., Gandon, F., and Faron-Zucker, C. (2006). Searching the semantic web: Approximate query processing based on ontologies. *Intelligent Systems*, 21(1):20–27. Cited on page 15.

[Damljanovic et al., 2010] Damljanovic, D., Agatonovic, M., and Cunningham, H. (2010). Identification of the question focus: Combining syntactic analysis and ontology-based lookup through the user interaction. In *Language Resources and Evaluation Conference (LREC)*. ELRA. Cited on page 14.

[Davies et al., 2010] Davies, S., Donaher, C., and Hatfield, J. (2010). Making the Semantic Web usable: interface principles to empower the layperson. *Journal of Digital Information*, 12(1). Cited on page 42.

[de Raedt, 2002] de Raedt, L. (2002). A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77. Cited on page 59.

[Dolques et al., 2010] Dolques, X., Huchard, M., Nebut, C., and Reitz, P. (2010). Learning transformation rules from transformation examples: An approach based on relational concept analysis. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW)*, pages 27–32. IEEE. Cited on page 38.

[Ducassé and Cellier, 2014] Ducassé, M. and Cellier, P. (2014). Fair and fast convergence on islands of agreement in multicriteria group decision making by logical navigation. *Group Decision and Negotiation*. To appear. Cited on page 56.

[Ducassé and Ferré, 2008] Ducassé, M. and Ferré, S. (2008). Fair(er) and (almost) serene committee meetings with logical and formal concept analysis. In Eklund, P. and Haemmerlé, O., editors, *Proceedings of the International Conference on Conceptual Structures*, LNAI 5113, pages 217–230. Springer. Cited on pages 34 and 56.

[Ducassé et al., 2011] Ducassé, M., Ferré, S., and Cellier, P. (2011). Building up shared knowledge with logical information systems. In Napoli, A. and Vychodil, V., editors, *Proceedings of the 8th International Conference on Concept Lattices and their Applications*, pages 31–42. INRIA. ISBN 978-2-905267-78-8. Cited on page 37.

[Ducrou and Eklund, 2008] Ducrou, J. and Eklund, P. (2008). An intelligent user interface for browsing and search MPEG-7 images using concept lattices. *Int. J. Foundations of Computer Science, World Scientific*, 19(2):359–381. Cited on page 18.

[Ferré, 2002] Ferré, S. (2002). *Systèmes d'information logiques : un paradigme logico-contextuel pour interroger, naviguer et apprendre*. Thèse d'université, Université de Rennes 1. Accessible en ligne à l'adresse http://www.irisa.fr/bibli/publi/theses/theses02.html. Cited on page 30.

[Ferré, 2006] Ferré, S. (2006). Negation, opposition, and possibility in logical concept analysis. In Missaoui, R. and Schmid, J., editors, *Int. Conf. Formal Concept Analysis*, LNCS 3874, pages 130–145. Springer. Cited on page 30.

[Ferré, 2008] Ferré, S. (2008). Agile browsing of a document collection with dynamic taxonomies. In Tjoa, A. M. and Wagner, R. R., editors, *DEXA*

Int. Work. Dynamic Taxonomies and Faceted Search (FIND), pages 377–381. IEEE Computer Society. Cited on page 31.

[Ferré, 2009a] Ferré, S. (2009a). Camelis: a logical information system to organize and browse a collection of documents. *Int. J. General Systems*, 38(4):379–403. Cited on pages 30, 32, and 127.

[Ferré, 2009b] Ferré, S. (2009b). Efficient browsing and update of complex data based on the decomposition of contexts. In Rudolph, S., Dau, F., and Kuznetsov, S. O., editors, *Int. Conf. Conceptual Structures*, LNCS 5662, pages 159–172. Springer. Cited on page 31.

[Ferré, 2010] Ferré, S. (2010). Conceptual navigation in RDF graphs with SPARQL-like queries. In Kwuida, L. and Sertkaya, B., editors, *Int. Conf. Formal Concept Analysis*, LNCS 5986, pages 193–208. Springer. Cited on page 38.

[Ferré, 2012] Ferré, S. (2012). SQUALL: a controlled natural language for querying and updating RDF graphs. In Kuhn, T. and Fuchs, N., editors, *Controlled Natural Languages*, LNCS 7427, pages 11–25. Springer. Cited on pages 47 and 173.

[Ferré, 2013] Ferré, S. (2013). Squall: A controlled natural language as expressive as SPARQL 1.1. In Métais, E., Meziane, F., Saraee, M., Sugumaran, V., and Vadera, S., editors, *Int. Conf. Applications of Natural Language to Information System (NLDB)*, LNCS 7934, pages 114–125. Springer. Cited on pages 47 and 173.

[Ferré, 2014] Ferré, S. (2014). SQUALL: The expressiveness of SPARQL 1.1 made available as a controlled natural language. *Data & Knowledge Engineering*. Cited on page 173.

[Ferré et al., 2012] Ferré, S., Allard, P., and Ridoux, O. (2012). Cubes of concepts: Multi-dimensional exploration of multi-valued contexts. In Domenach, F., Ignatov, D. I., and Poelmans, J., editors, *Int. Conf. Formal Concept Analysis*, LNCS 7278, pages 112–127. Springer. Cited on page 35.

[Ferré and Hermann, 2011] Ferré, S. and Hermann, A. (2011). Semantic search: Reconciling expressive querying and exploratory search. In Aroyo, L. and Welty, C., editors, *Int. Semantic Web Conf.*, LNCS 7031, pages 177–192. Springer. Cited on pages 38 and 40.

[Ferré and Hermann, 2012] Ferré, S. and Hermann, A. (2012). Reconciling faceted search and query languages for the Semantic Web. *Int. J. Metadata, Semantics and Ontologies*, 7(1):37–54. Cited on pages 22, 38, and 153.

[Ferré and King, 2004] Ferré, S. and King, R. D. (2004). BLID: an application of logical information systems to bioinformatics. In Eklund, P., editor, *Int. Conf. Formal Concept Analysis*, LNCS 2961, pages 47–54. Springer. Cited on page 31.

[Ferré and King, 2005] Ferré, S. and King, R. D. (2005). A dichotomic search algorithm for mining and learning in domain-specific logics. *Fundamenta Informaticae – Special Issue on Advances in Mining Graphs, Trees and Sequences*, 66(1-2):1–32. Cited on page 54.

[Ferré and King, 2006] Ferré, S. and King, R. D. (2006). Finding motifs in protein secondary structure for use in function prediction. *Journal of Computational Biology*, 13(3):719–731. Cited on page 54.

[Ferré and Ridoux, 2000a] Ferré, S. and Ridoux, O. (2000a). A file system based on concept analysis. In Sagiv, Y., editor, *International Conference on Rules and Objects in Databases*, number 1861 in Lecture Notes in Computer Science, pages 1033–1047. Springer. Cited on page 30.

[Ferré and Ridoux, 2000b] Ferré, S. and Ridoux, O. (2000b). A logical generalization of formal concept analysis. In Mineau, G. and Ganter, B., editors, *International Conference on Conceptual Structures*, number 1867 in Lecture Notes in Computer Science, pages 371–384. Springer. Cited on page 30.

[Ferré and Ridoux, 2001a] Ferré, S. and Ridoux, O. (2001a). A framework for developing embeddable customized logics. In Pettorossi, A., editor, *Int. Work. Logic-based Program Synthesis and Transformation*, LNCS 2372, pages 191–215. Springer. Cited on page 30.

[Ferré and Ridoux, 2001b] Ferré, S. and Ridoux, O. (2001b). Searching for objects and properties with logical concept analysis. In Delugach, H. S. and Stumme, G., editors, *International Conference on Conceptual Structures*, LNCS 2120, pages 187–201. Springer. Cited on page 30.

[Ferré and Ridoux, 2002] Ferré, S. and Ridoux, O. (2002). The use of associative concepts in the incremental building of a logical context. In U. Priss, D. Corbett, G. A., editor, *Int. Conf. Conceptual Structures*, LNCS 2393, pages 299–313. Springer. Cited on page 42.

[Ferré and Ridoux, 2004] Ferré, S. and Ridoux, O. (2004). An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419. Cited on pages 30, 32, and 81.

[Ferré and Ridoux, 2006] Ferré, S. and Ridoux, O. (2006). Logic functors: A toolbox of components for building customized and embeddable logics. Research Report RR-5871, Irisa. Cited on page 30.

[Ferré and Ridoux, 2007] Ferré, S. and Ridoux, O. (2007). Logical information systems: from taxonomies to logics. In *DEXA Work. Dynamic Taxonomies and Faceted Search (FIND)*, pages 212–216. IEEE Computer Society. Cited on page 31.

[Ferré et al., 2005] Ferré, S., Ridoux, O., and Sigonneau, B. (2005). Arbitrary relations in formal concept analysis and logical information systems. In *ICCS*, LNCS 3596, pages 166–180. Springer. Cited on page 38.

[Ferré and Rudolph, 2012] Ferré, S. and Rudolph, S. (2012). Advocatus diaboli - exploratory enrichment of ontologies with negative constraints. In ten Teije et al., A., editor, *Int. Conf. Knowledge Engineering and Knowledge Management (EKAW)*, LNAI 7603, pages 42–56. Springer. Cited on page 45.

[Foret and Ferré, 2010] Foret, A. and Ferré, S. (2010). On categorial grammars as logical information systems. In Kwuida, L. and Sertkaya, B., editors, *Int. Conf. Formal Concept Analysis*, LNCS 5986, pages 225–240. Springer. Cited on page 31.

[Fu et al., 2004] Fu, H., Fu, H., Njiwoua, P., and Mephu Nguifo, E. (2004). A comparative study of fca-based supervised classification algorithms. In *Int. Conf. Formal Concept Analysis*, LNCS 2961, pages 313–320. Springer. Cited on page 54.

[Fuchs et al., 1999] Fuchs, N., Schwertel, U., and Schwitter, R. (1999). Attempto controlled english not just another logic specification language. In *Logic-based program synthesis and transformation (LOPSTR)*, pages 1–20. Springer. Cited on page 14.

[Fuchs et al., 2006] Fuchs, N. E., Kaljurand, K., and Schneider, G. (2006). Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. In Sutcliffe, G. and Goebel, R., editors, *FLAIRS Conference*, pages 664–669. AAAI Press. Cited on pages 14, 41, and 48.

[Ganter and Kuznetsov, 2001] Ganter, B. and Kuznetsov, S. (2001). Pattern structures and their projections. In Delugach, H. S. and Stumme, G., editors, *Int. Conf. Conceptual Structures*, LNCS 2120, pages 129–142. Springer. Cited on page 31.

[Ganter and Wille, 1999] Ganter, B. and Wille, R. (1999). *Formal Concept Analysis — Mathematical Foundations*. Springer. Cited on page 17.

[Gemmell et al., 2006] Gemmell, J., Bell, G., and Lueder, R. (2006). Mylifebits: a personal database for everything. *Commun. ACM*, 49(1):88–95. Cited on page 59.

[Godin et al., 1993] Godin, R., Missaoui, R., and April, A. (1993). Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. *International Journal of Man-Machine Studies*, 38(5):747–767. Cited on page 17.

[Guérin et al., 2012] Guérin, C., Bertet, K., and Revel, A. (2012). An approach to semantic content based image retrieval using logical concept analysis. application to comicbooks. In *Int. Work. What can FCA do for Artificial Intelligence? (FCA4AI), co-located with ECAI*, pages 53–56. Cited on page 40.

[Guillas et al., 2008] Guillas, S., Bertet, K., Visani, M., Ogier, J.-M., and Girard, N. (2008). Some links between decision tree and dichotomic lattice. In Belohlavek, R. and Kuznetsov, S., editors, *Int. Conf. Concept Lattices and Their Applications (CLA)*, CEUR 433, pages 193–205. Cited on page 18.

[Guyonvarch et al., 2013] Guyonvarch, J., Ferré, S., and Ducassé, M. (2013). Scalable Query-based Faceted Search on top of SPARQL Endpoints for Guided and Expressive Semantic Search. Research report PI-2009, LIS - IRISA. Cited on pages 40 and 56.

[Hacene et al., 2007] Hacene, M. R., Huchard, M., Napoli, A., and Valtchev, P. (2007). A proposal for combining formal concept analysis and description logics for mining relational data. In Kuznetsov, S. O. and Schmidt, S., editors, *Int. Conf. Formal Concept Analysis*, LNCS 4390, pages 51–65. Springer. Cited on page 38.

[Haller, 2010] Haller, H. (2010). QuiKey – an efficient semantic command line. In *Knowledge Engineering and Management by the Masses (EKAW)*, pages 473–482. Springer. Cited on page 41.

[Harth, 2010] Harth, A. (2010). VisiNav: A system for visual search and navigation on web data. *J. Web Semantics*, 8(4):348–354. Cited on page 21.

[Hearst et al., 2002] Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K., and Yee, K.-P. (2002). Finding the flow in web site search. *Communications of the ACM*, 45(9):42–49. Cited on pages 20 and 31.

[Heim et al., 2010] Heim, P., Ertl, T., and Ziegler, J. (2010). Facet graphs: Complex semantic querying made easy. In et al., L. A., editor, *Extended Semantic Web Conference*, LNCS 6088, pages 288–302. Springer. Cited on page 21.

[Hermann, 2012] Hermann, A. (2012). *Création et mise à jour d'objets dans une base de connaissances*. PhD thesis, Thèse de l'INSA Rennes - École doctorale MATISSE. supervised by M. Ducassé and S. Ferré. Cited on page 41.

[Hermann et al., 2012] Hermann, A., Ferré, S., and Ducassé, M. (2012). An interactive guidance process supporting consistent updates of RDFS graphs. In ten Teije et al., A., editor, *Int. Conf. Knowledge Engineering and Knowledge Management (EKAW)*, LNAI 7603, pages 185–199. Springer. Cited on page 42.

[Hildebrand et al., 2006] Hildebrand, M., van Ossenbruggen, J., and Hardman, L. (2006). /facet: A browser for heterogeneous semantic web repositories. In *et al*, I. C., editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 272–285. Springer. Cited on page 21.

[Hitzler et al., 2009] Hitzler, P., Krötzsch, M., and Rudolph, S. (2009). *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC. Cited on pages 9 and 16.

[Hurtado et al., 2008] Hurtado, C., Poulovassilis, A., and Wood, P. (2008). Query relaxation in RDF. In Spaccapietra, S., editor, *Journal on Data Semantics X*, LNCS 4900, pages 31–61. Springer. Cited on page 15.

[Hyvönen and Mäkelä, 2006] Hyvönen, E. and Mäkelä, E. (2006). Semantic autocompletion. In *The Semantic Web (ASWC)*, pages 739–751. Springer. Cited on page 15.

[Hyvönen et al., 2005] Hyvönen, E., Mäkelä, E., Salminen, M., Valo, A., Viljanen, K., Saarela, S., Junnila, M., and Kettula, S. (2005). Museumfinland

- finnish museums on the semantic web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):224–241. Cited on page 21.

[Hyvönen et al., 2002] Hyvönen, E., Styrman, A., and Saarela, S. (2002). Ontology-based image retrieval. In Hyvönen, E. and Klemettinen, M., editors, *Towards the Semantic Web and Web Services*, pages 15–27. XML Finland Association. Cited on page 20.

[Kaljurand and Kuhn, 2013] Kaljurand, K. and Kuhn, T. (2013). A multilingual semantic wiki based on attempto controlled english and grammatical framework. In *The Semantic Web: Semantics and Big Data*, pages 427–441. Springer. Cited on page 42.

[Kämpgen and Harth, 2011] Kämpgen, B. and Harth, A. (2011). Transforming statistical linked data for use in OLAP systems. In *Int. Conf. Semantic systems*, pages 33–40. ACM. Cited on page 51.

[Kaufmann and Bernstein, 2010] Kaufmann, E. and Bernstein, A. (2010). Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases. *J. Web Semantics*, 8(4):377–393. Cited on pages 13 and 15.

[Kuhn, 2009] Kuhn, T. (2009). How controlled English can improve semantic wikis. In *Semantic Wiki Work. (SemWiki) at the Eu. Semantic Web Conf. (ESWC)*, volume 464. CEUR-WS. Cited on page 41.

[Kuhn, 2013] Kuhn, T. (2013). A survey and classification of controlled natural languages. *Computational Linguistics*. Cited on pages 14 and 48.

[Kyzirakos et al., 2012] Kyzirakos, K., Karpathiotakis, M., and Koubarakis, M. (2012). Strabon: A semantic geospatial DBMS. In *Int. Semantic Web Conf.*, pages 295–311. Springer. Cited on page 52.

[Lamming et al., 1994] Lamming, M., Brown, P., Carter, K., Eldridge, M., Flynn, M., Louie, G., Robinson, P., and Sellen, A. (1994). The design of a human memory prosthesis. *The Computer Journal*, 37(3):153–163. Cited on page 59.

[Lehmann et al., 2013] Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P. N., Hellmann, S., Morsey, M., van Kleef, P., Auer, S., and Bizer, C. (2013). DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*. Under review. Cited on page 10.

[Lindig, 1995] Lindig, C. (1995). Concept-based component retrieval. In *IJCAI Work. Formal Approaches to the Reuse of Plans, Proofs, and Programs*. Morgan Kaufmann. Cited on page 18.

[Lopez et al., 2012] Lopez, V., Fernández, M., Motta, E., and Stieler, N. (2012). PowerAqua: Supporting users in querying and exploring the semantic web. *Semantic Web*, 3(3):249–265. Cited on page 14.

[Lopez et al., 2011] Lopez, V., Uren, V. S., Sabou, M., and Motta, E. (2011). Is question answering fit for the semantic web?: A survey. *Semantic Web*, 2(2):125–155. Cited on page 13.

[Lu et al., 2007] Lu, J., Ma, L., Zhang, L., Brunner, J., Wang, C., Pan, Y., and Yu, Y. (2007). SOR: A practical system for ontology storage, reasoning and search (demo). In *Int. Conf. Very Large Databases (VLDB)*, VLDB Endowment, pages 1402–1405. ACM. Cited on page 21.

[Mäkelä et al., 2006] Mäkelä, E., Hyvönen, E., and Saarela, S. (2006). Ontogator - a semantic view-based search engine service for web applications. In *et al.*, I. F. C., editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 847–860. Springer. Cited on page 21.

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

[McCrae et al., 2011] McCrae, J., Spohr, D., and Cimiano, P. (2011). Linking lexical resources and ontologies on the semantic web with lemon. In *Extended Semantic Web Conference (ESWC)*, LNCS 6643, pages 245–259. Springer. Cited on page 53.

[Messai et al., 2008] Messai, N., Devignes, M.-D., Napoli, A., and Smaïl-Tabbone, M. (2008). Extending attribute dependencies for lattice-based querying and navigation. In Eklund, P. W. and Haemmerlé, O., editors, *Int. Conf. Conceptual Structures*, LNCS 5113, pages 189–202. Springer. Cited on page 18.

[Miller and Nadathur, 1986] Miller, D. A. and Nadathur, G. (1986). Higher-order logic programming. In Shapiro, E., editor, *In Third Int. Conf. Logic Programming*, LNCS, pages 448–462, London. Springer-Verlag. Cited on page 32.

[Montague, 1970] Montague, R. (1970). Universal grammar. *Theoria*, 36:373–398. Cited on page 47.

[Muggleton and Raedt, 1994] Muggleton, S. and Raedt, L. D. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679. Cited on page 54.

[Ngomo et al., 2013] Ngomo, A.-C. N., Bühmann, L., Unger, C., Lehmann, J., and Gerber, D. (2013). Sorry, I don't speak SPARQL: translating SPARQL queries into natural language. In *WWW*, pages 977–988. Cited on page 53.

[Noy et al., 2001] Noy, N., Sintek, M., Decker, S., Crubezy, M., Fergerson, R., and Musen, M. (2001). Creating semantic web contents with Protege-2000. *Intelligent Systems, IEEE*, 16(2):60–71. Cited on page 41.

[Oren et al., 2006] Oren, E., Delbru, R., and Decker, S. (2006). Extending faceted navigation to RDF data. In *et al*, I. C., editor, *Int. Semantic Web Conf.*, LNCS 4273, pages 559–572. Springer. Cited on page 21.

[Padioleau, 2005] Padioleau, Y. (2005). *Logic File System, un système de fichier basé sur la logique*. Thèse d'université, Université de Rennes 1. supervised by O. Ridoux. Cited on page 33.

[Padioleau and Ridoux, 2003] Padioleau, Y. and Ridoux, O. (2003). A logic file system. In *Usenix Annual Technical Conference*, pages 99–112. USENIX. Cited on page 33.

[Padioleau and Ridoux, 2005] Padioleau, Y. and Ridoux, O. (2005). A parts-of-file file system. In *USENIX Annual Technical Conference, General Track (Short Paper)*. Cited on page 33.

[Pedersen and Jensen, 2001] Pedersen, T. B. and Jensen, C. S. (2001). Multidimensional database technology. *Computer*, 34(12):40–46. Cited on page 22.

[Romano, 2008] Romano, P. (2008). Automation of in-silico data analysis processes through workflow management systems. *Briefings in Bioinformatics*, 9(1):57–68. Cited on page 57.

[Rouane-Hacene et al., 2013] Rouane-Hacene, M., Huchard, M., Napoli, A., and Valtchev, P. (2013). Relational concept analysis: mining concept lattices from multi-relational data. *Annals of Mathematics and Artificial Intelligence*, 67(1):81–108. Cited on page 38.

[Sacco, 2000] Sacco, G. M. (2000). Dynamic taxonomies: A model for large information bases. *IEEE Transactions Knowledge and Data Engineering*, 12(3):468–479. Cited on page 31.

[Sacco, 2008] Sacco, G. M. (2008). Rosso tiziano: A system for user-centered exploration and discovery in large image information bases. In Tjoa, A. M. and Wagner, R., editors, *DEXA Work. Dynamic Taxonomy and Faceted Search (FIND)*, pages 297–311. IEEE Computer Society. Cited on page 20.

[Sacco and Tzitzikas, 2009] Sacco, G. M. and Tzitzikas, Y., editors (2009). *Dynamic taxonomies and faceted search*. The information retrieval series. Springer. Cited on pages 20, 24, and 31.

[Schwitter et al., 2008] Schwitter, R., Kaljurand, K., Cregan, A., Dolbear, C., and Hart, G. (2008). A comparison of three controlled natural languages for OWL 1.1. In Clark, K. and Patel-Schneider, P. F., editors, *Workshop on OWL: Experiences and Directions (OWLED)*, volume 258. CEUR-WS. Cited on page 48.

[Shearer et al., 2008] Shearer, R., Motik, B., and Horrocks, I. (2008). Hermit: A highly-efficient OWL reasoner. In *OWLED*, volume 432. Cited on page 46.

[Smits et al., 2013a] Smits, G., Pivert, O., and Girault, T. (2013a). Towards reconciling expressivity, efficiency and user-friendliness in database flexible querying. In *Int. Conf. Fuzzy Systems (FUZZ)*, pages 1–8. IEEE. Cited on page 15.

[Smits et al., 2013b] Smits, G., Pivert, O., Jaudoin, H., and Paulus, F. (2013b). An autocompletion mechanism for enriched keyword queries to RDF data sources. In *Flexible Query Answering Systems*, LNCS 8132, pages 601–612. Springer. Cited on pages 15 and 48.

[Smits et al., 2014] Smits, G., Pivert, O., Jaudoin, H., and Paulus, F. (2014). AGGREGO SEARCH: Interactive keyword query construction (demo). In *Int. Conf. Extending Database Technology*, pages 636–639. Cited on page 48.

[SPARQL11, 2012] SPARQL11 (2012). SPARQL 1.1 query language. `http://www.w3.org/TR/sparql11-query/`, W3C Recommendation. Cited on pages 10, 13, and 14.

[Suominen et al., 2007] Suominen, O., Viljanen, K., and Hyvönen, E. (2007). User-centric faceted search for semantic portals. In Franconi, E., Kifer, M., and May, W., editors, *Eu. Semantic Web Conf.*, LNCS 4519, pages 356–370. Springer. Cited on page 21.

[Tarr et al., 1999] Tarr, P., Ossher, H., Harrison, W., and Sutton, S. (1999). N degrees of separation: Multi-dimentional separation of concerns. In *ICSE*, pages 107–119. IEEE Computer Society. Cited on page 17.

[Verbeke, 1973] Verbeke, C. (1973). Caterpillar fundamental English. *Training & Development Journal*, 27(2):36–40. Cited on page 14.

[Visani et al., 2011] Visani, M., Bertet, K., and Ogier, J.-M. (2011). Navigala: an original symbol classifier based on navigation through a galois lattice. *Int. J. Pattern Recognition and Artificial Intelligence*, 25(04):449–473. Cited on page 54.

[Völkel et al., 2006] Völkel, M., Krötzsch, M., Vrandecic, D., Haller, H., and Studer, R. (2006). Semantic wikipedia. In *Int. Conf. World Wide Web*, pages 585–594. ACM. Cited on page 42.

[Völker and Rudolph, 2008] Völker, J. and Rudolph, S. (2008). Lexicological acquisition of OWL-DL axioms. In Medina, R. and Obiedkov, S. A., editors, *Int. Conf. Formal Concept Analysis*, LNCS 4933, pages 62–77. Springer. Cited on page 47.

[Walter et al., 2013] Walter, S., Unger, C., and Cimiano, P. (2013). A corpus-based approach for the induction of ontology lexica. In *Int. Conf. Applications of Natural Languages to Information Systems (NLDB)*, LNCS 7934, pages 102–113. Springer. Cited on page 53.

[Wille, 1982] Wille, R. (1982). *Ordered Sets*, chapter Restructuring lattice theory: an approach based on hierarchies of concepts, pages 445–470. Reidel. Cited on page 17.

[XQuery3, 2013] XQuery3 (2013). XQuery 3.0: An XML query language. `http://www.w3.org/TR/xquery-30/`, W3C Proposed Recommendation. Cited on page 13.

Index

Acronyms

ACN Abstract Conceptual Navigation

BI Business Intelligence

CNL Controlled Natural Language

DL Description Logic

FCA Formal Concept Analysis

FS Faceted Search

GIS Geographical Information System

LCA Logical Concept Analysis

LIS Logical Information System

LOD Linked Open Data

NL Natural Language

NLI Natural Language Interface

NLP Natural Language Processing

OLAP On-Line Analytical Processing

OWL Web Ontology Language

QA Question Answering

QFS Query-based Faceted Search

RCA Relational Concept Analysis

RDB Relational Databases

RDF Resource Description Framework

SW Semantic Web

Appendix A

Introduction to Logical Information Systems (2004)

This journal article [**Ferré and Ridoux, 2004**] has been published in *Information Processing & Management* in 2004. It summarizes the different contributions of my PhD: Logical Concept Analysis (LCA), querying and guided navigation based on logical concept lattices, non-monotonic logical update of logical contexts, data mining and machine learning for the guided update of logical contexts, and implementation aspects including logic functors.

Introduction to Logical Information Systems

S. Ferré [a,1], O. Ridoux [b]

[a] *UWA, Penglais, Aberystwyth SY23 3DB, UK,*
Tel: +44 1970 621922, Fax: +44 1970 622455,
Email: sbf@aber.ac.uk

[b] *Irisa/IFSIC, Campus de Beaulieu, 35042 Rennes cedex, France,*
Tel: +33 2 99 84 73 30, Fax: +33 2 99 84 71 71,
Email: ridoux@irisa.fr

Abstract

Logical Information Systems (LIS) use logic in a uniform way to describe their contents, to query it, to navigate through it, to analyze it, and to maintain it. They can be given an abstract specification that does not depend on the choice of a particular logic, and concrete instances can be obtained by instantiating this specification with a particular logic. In fact, a logic plays in a LIS the role of a schema in data-bases. We present the principles of logical information systems, the constraints they impose on the expression of logics, and hints for their effective implementation.

Key words: information systems, information search and retrieval, query formulation, representation languages, deduction and theorem proving.

1 Introduction

Several researchers have recognized the *name problem* in information systems (Gifford, Jouvelot, Sheldon, and O'Toole, 1991; Gopal and Manber, 1999). In these systems, *things* are given *names* and very often a thing has only a few names (frequently only one). For instance, in the most rudimentary information systems, like hierarchical file systems, a thing is a file and its name is the only path that leads from the root to the file.

[1] This work was achieved at Irisa (Rennes, France), and funded by a scholarship from CNRS and Région Bretagne.

The problem with having only a few names is that they must be very carefully chosen to tackle for all future usages of the things. Experience shows that this is impossible to do for a wide range of future usages. For instance, there is no hierarchical organization of many pieces of software that fits all the needs of software development: programming, testing, documenting, debugging, etc. A non-IT example is a cook-book. Cook-books are often organized following the course of a meal, and it is thus very difficult to search for a recipe according to other criteria, like the (un)availability of a oven, or a diet constraint. Note that even electronic cook-books often follow this structure. Electronic or not, the classical solution to this problem is to search in the whole information system/document to find the desired thing. However, this is only a partial solution because query language used in search engines are often too restricted.

There have been several attempts for solving the name problem. We mention here only two of them that we have selected because they illustrate the difficulty of the enterprise, SFS (Gifford, Jouvelot, Sheldon, and O'Toole, 1991) and HAC (Gopal and Manber, 1999). They have in common to combine *hierarchical naming* and *boolean querying*.

Hierarchical naming is frequently found in computer tools: e.g., file systems, book-marks, or menus. In this model, searching is done by *navigating* in a classification structure that is often built and maintained manually. Navigating implies a notion of *place*; *being in* a place, and *going to* another place. A notion of *neighborhood* helps specifying the "other places" relatively to the place one is currently in. Many applications require that a place is a place to read from as well as a place to write on.

Boolean querying is often found in information servers such as search engines on the Web (e.g., Google). In this model, searching is done by using *queries*, generally expressed in a kind of propositional logic. A well-recognized difficulty of this model is the necessity of having a good knowledge of the terminology used in the information system, and of having a precise idea of what is searched for. However, it is easier to recognize an object than to describe it, and the necessity of expressing a query can repel casual users.

Then, which search model should be preferred: navigation or querying? In fact, it depends on situations, and it is sometimes needed to use both of them in the same search. For instance, someone could wish to begin his search by a query and then refine it with navigation. Hybrid systems combining hierarchical classification and boolean querying have been proposed in the domain of file systems:

— SFS (*Semantic File System*, Gifford, Jouvelot, Sheldon, and O'Toole, 1991) extends the hierarchical model of usual file systems with *virtual directories* that correspond to queries. These queries concern file properties that are automat-

ically extracted by *transducers*, and are expressed with valued attributes. So, two organization and storage methods coexist: the standard hierarchy that gives a name to files, and virtual directories that enable associative searches on intrinsic file properties. Unfortunately, these two methods cannot be combined freely in general. In particular, virtual directories are not places to write into.

— HAC (*Hierarchy And Content*, Gopal and Manber, 1999) also uses queries to build directories based on file contents, but these directories are integrated in the hierarchy. This enables to combine hierarchy and contents in searches. However, users are allowed to move a file in a directory even if it does not satisfy the query associated to the directory, which results in consistency problems.

The drawback of these hybrid systems is their lack of consistency. Indeed, they have two search models that are not tightly connected, which makes it difficult to switch from one model to the other, and to combine both in the same search.

We propose a scheme called Logical Information Systems (LIS) in which queries are really places to read from and to write into. The scheme is flexible in the sense that the neighborhood relation is very dense (i.e., things have many names). It incurs no inconsistency or dangling links problem, because the neighborhood relation is managed automatically. Finally, it supports both querying and navigation, and arbitrary combinations of both because names and queries belong to the same language (Godin, Missaoui, and April, 1993; Lindig, 1995). This scheme is based on a variant of Formal Concept Analysis (Barbut and Monjardet, 1970; Wille, 1982; Ganter and Wille, 1999) called Logical Concept Analysis (Ferré and Ridoux, 2000).

The article is organized as follows. Section 2 presents the principles of Logical Concept Analysis, and then Section 3 shows how it can be used for navigating and querying. How to create and update the content of a LIS is presented in Section 4. These sections refer to a logic passed as a parameter which is to be used for naming, querying and navigating. Section 5 presents other functions of a LIS like automated updating, data-mining and learning. Section 6 explains how all this can be done practically. Finally, conclusions and perspectives are given in Section 7.

2 Logical Concept Analysis

The origin of this work is the search for flexible organizations for managing, updating, querying, or navigating in data. In this context, several roles are

played by possibly different people: e.g., designer, administrator, and end-user. Hierarchical organizations are not flexible, and updating, querying and navigation are difficult to conciliate (see for instance the view update problem in data-bases (Keller, 1985)). The literature shows that *Formal Concept Analysis* (*FCA*, Ganter and Wille, 1999) is a good candidate for supporting querying and navigation.

The basis of FCA is a *formal context* that associates attributes to objects; objects are the *things* of the introduction of this article, and the collection of attributes associated to one thing/object is its *name*. FCA has received attention for its application in many domains such as in software engineering (Snelting, 1998; Lindig, 1995; Krone and Snelting, 1994). The interest of FCA as a navigation tool in general has also been recognized (Godin, Missaoui, and April, 1993; Lindig, 1995; Vogt and Wille, 1994). However, we feel it is not flexible enough as far as the naming of things is concerned, and the literature on FCA insists more on analyzing a given context than on managing evolving contexts. In this section, we present an extension to FCA that allows for a richer name language.

The variety of application domains brings the need for more sophisticated formal contexts than the mere presence/absence of attributes. For instance, many application domains use numerical values (e.g., lengths, prices, ages), and the need to express negation and disjunction is often felt. In a much more specialized scope, it is imaginable to use the type of software components as search keys (Di Cosmo, 1995). So, one cannot fix *a priori* limits to the sophistication of attributes. Several enrichments to the attribute structure have been proposed: e.g., many-valued attributes (Ganter and Wille, 1999), and first-order terms (Chaudron and Maille, 1998). However, not a single extended FCA framework covers all the concrete domains, and no one can pretend covering all the concrete domains to come.

For the same reasons, *logic* has already been proposed, in information retrieval, for expressing object descriptions and queries (van Rijsbergen, 1986; Meghini, Sebastiani, Straccia, and Thanos, 1993). In this setting, the relevance of an object to a query relies on logical inference. As with Description Logics (Brachman, 1979; Napoli, 1997), we use an exact relevance relation, called *subsumption*, as some applications may need precise retrieval mechanisms; but uncertain relevance has also been considered (Crestani and Lalmas, 2001). See also a discussion on the management of uncertainty in the conclusion of this article. So, we propose to construct a more general framework for concept analysis, Logical Concept Analysis (LCA), in which attributes are replaced by the formulas of a logic . Moreover, we make this logic a parameter of LCA. This allows for instantiating the general framework by merely filling in a dedicated logic.

Logical Concept Analysis (LCA) is a generalization of FCA, where sets of attributes are replaced by formulas of an (almost) arbitrary logic. More details about the relation with previous works in FCA can be found in (Ferré and Ridoux, 2000), and all proofs can be found in (Ferré and Ridoux, 1999).

We first define what we call a *logic* in this article.

Definition 1 (logic) *A* logic *is a 6-tuple* $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$*, where*

- $\mathcal{L}$ *is the language of* formulas*,*
- $\sqsubseteq$ *is the* subsumption *relation (pre-order over* $\mathcal{L}$*),*
- $\sqcap$ *and* $\sqcup$ *are respectively* conjunction *and* disjunction *(binary operations),*
- $\top$ *and* $\bot$ *are respectively* tautology *and* contradiction *(constant formulas).*

Such a logic must form a lattice (Davey and Priestley, 1990), whose order is derived in the usual way from the pre-order $\sqsubseteq$*, and such that* $\sqcap$ *and* $\sqcup$ *are respectively the* infimum *(greatest lower bound) and the* supremum *(least upper bound), and* $\top$ *and* $\bot$ *are respectively the* top *and the* bottom*. The notation* $\mathcal{L}$ *can be used as a name for the logic lattice.*

If $f \sqsubseteq g$ and $g \sqsubseteq f$, f and g are called logically *equivalent*, which is denoted as $f \equiv g$; we consider them as different representations of the same equivalence class, and in fact we will consider that elements of $\mathcal{L}$ are the equivalence classes. We just assume, for practical reasons, that operations $\sqsubseteq$, $\sqcap$, and $\sqcup$ are computable. Some *semantics* is usually used to define a logic, but we delay the discussion about it until Section 6.2 as we do not need it here. Just keep in mind that formulas are here interpreted by sets of individuals or objects (like in Description Logics), rather than by truth values (like in first-order logic). In order to clarify things, here is an example of a logic.

Example 2 (Propositional logic) *An example of logic that can be used in LCA is* propositional logic*. On the syntactic side, the set of propositions* $\mathcal{P}$ *contains atomic propositions (taken in a set* $\mathcal{A}$*), formulas* 0 *and* 1*, and is closed under binary connectives* $\wedge$ *and* $\vee$*, and unary connective* $\neg$*. We say that a proposition* p *is subsumed by another one* q *if* $\neg p \vee q$ *is a valid proposition* ($p \vDash q$)*. Then,* $(\mathcal{P}, \vDash, \wedge, \vee, 1, 0)$ *satisfies Definition 1, because it is the well-known boolean algebra.*

This example shows that though the interface of the logic is limited to the tuple $(\mathcal{L}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$*, an actual logic may have more connectives:* $\neg$ *in this example.*

We now define the main notions and results of LCA: *context*, *concept lattice*,

and *contextualized subsumption.*

2.2 Logical context and Galois connection

A *logical context* plays the role of tables in a database, as it gathers the knowledge one has about objects of interest (e.g., files, bibliographic references, recipes).

Definition 3 (logical context) *A* logical context *(context for short) is a triple* $K = (\mathcal{O}, \mathcal{L}, d)$ *where:*

- $\mathcal{O}$ *is a finite set of objects,*
- $\mathcal{L}$ *is a logic (as in Definition 1),*
- d *is a mapping from* $\mathcal{O}$ *to* $\mathcal{L}$ *that* describes *each object by a formula.*

Then, we define two mappings between sets of objects ($2^{\mathcal{O}}$) and formulas ($\mathcal{L}$) in a context K, that we prove to be a Galois connection (Davey and Priestley, 1990). A first mapping τ_K connects each formula f to its *instances*, i.e., objects whose description is subsumed by f; $\tau_K(f)$ is the *extent* of f. A second mapping σ_K connects each set of objects $O \subseteq \mathcal{O}$ to the most precise formula subsuming all descriptions of objects in O; $\sigma_K(O)$ is the *intent* of O.

Definition 4 (mappings τ_K and σ_K) *Let* $K = (\mathcal{O}, \mathcal{L}, d)$ *be a context,* $O \subseteq \mathcal{O}$*, and* $f \in \mathcal{L}$*,*

- σ_K *(σ for short)* $: 2^{\mathcal{O}} \to \mathcal{L},\ \sigma_K(O) := \bigsqcup_{o \in O} d(o)$
- τ_K *(τ for short)* $: \mathcal{L} \to 2^{\mathcal{O}},\ \tau_K(f) := \{o \in \mathcal{O} \mid d(o) \sqsubseteq f\}$

Lemma 5 (Galois connection) *Let* K *be a context. The pair* (σ_K, τ_K) *is a* Galois connection *because*

$$\forall O \subseteq \mathcal{O}, f \in \mathcal{L} : \sigma_K(O) \sqsubseteq f \iff O \subseteq \tau_K(f).$$

In the following, we will drop the subscript K when possible.

Example 6 ($Triv$) *An example context will illustrate the rest of our development on LCA. Context* K_{Triv} *is deliberately small and simple as it is aimed at illustrating theoretical notions, and not at showing a realistic application of LCA. The logic used in this context is propositional logic* $\mathcal{P}$ *(see Example 2) with a set of atomic propositions* $\mathcal{A} = \{a, b, c\}$*. We define context* K_{Triv} *by* $(\mathcal{O}_{Triv}, \mathcal{P}, d_{Triv})$*, where* $\mathcal{O}_{Triv} = \{x, y, z\}$*, and where* $d_{Triv} = \{x \mapsto a, y \mapsto b, z \mapsto c \wedge (a \vee b)\}$*.*

A Logical Information System (LIS) is essentially a logical context equipped with navigation and management tools. Formulas serve as queries, and extents

as answers via mapping τ_K. This is only a rough description. We will see in Section 3 that a LIS answer can also be a formula. For illustration purpose, we consider a bibliographical information system.

Example 7 (*Bib*) *Let $Bib = (\mathcal{O}, \mathcal{P}_v, d)$ be a logical context where objects are bibliographical references, whose description are composed of a type (e.g., article, in-proceedings), a list of authors, a title, and a year of publication. The logic $\mathcal{P}_v$, used for expressing descriptions, is similar to the propositional logic $\mathcal{P}$ (see Example 2) except that atoms are replaced by valued attributes in the form* attr value*:* attr *is the name of an attribute (e.g.,* `author`*,* `year`*), and* value *expresses a logical property about the value of the attribute. For instance, numerical attributes can be described in an interval logic (e.g.,* `year in 1990..2000`*,* `year in 1995`*), and string attributes can be described by a string (e.g.,* `title is "Logical Information Systems"`*) or a substring (e.g.,* `title contains "System"`*). We now show as an example the logical description of our article on LCA (Ferré and Ridoux, 2000).*

```
  type is "InProceedings"
∧ author is "Sébastien Ferré, Olivier Ridoux"
∧ title is "A Logical Generalization of Formal Concept Analysis"
∧ year in 2000
```

In the sequel of this article, the context Bib refers to all ICCS publications until the year 1999, which consists in 209 objects.

2.3 Concept lattice and labelling

A *formal concept*, central notion of LCA, is the association of a set of objects and of a formula, which is stable for the Galois connection (σ, τ).

Definition 8 (formal concept) *In a context $K = (\mathcal{O}, \mathcal{L}, d)$, a* formal concept *(concept for short) is a pair $c = (O, f)$ where $O \subseteq \mathcal{O}$, and $f \in \mathcal{L}$, such that $\sigma_K(O) \equiv f$ and $\tau_K(f) = O$.*

The set of objects O is the concept extent *(written $ext(c)$), whereas formula f is its* intent *(written $int(c)$).*

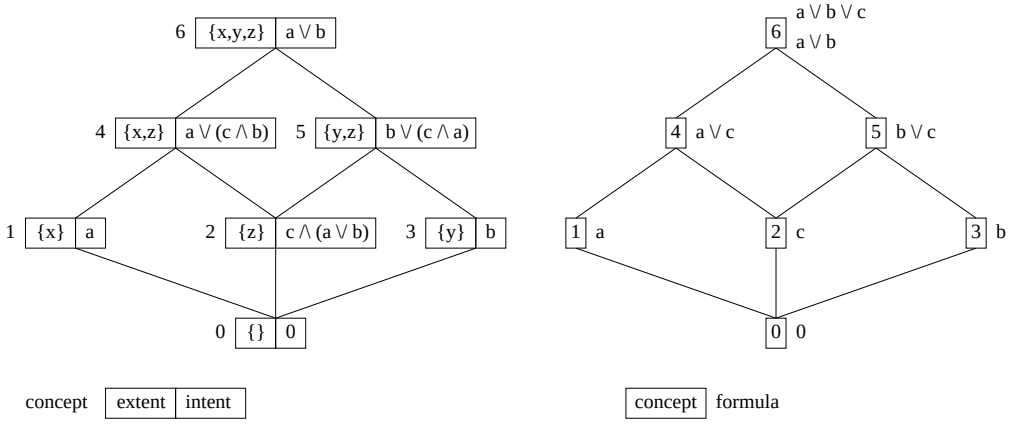
We write $=^c$ for concept equality. The set of all concepts that can be built in a context K is denoted by $\mathcal{C}_K$, and is partially ordered by $\leq^c$ defined below. The fundamental theorem of LCA is that $\langle \mathcal{C}_K; \leq^c \rangle$ forms a *lattice*, which is finite, hence complete.

Definition 9 (partial order $\leq^c$) *Let c_1 and c_2 be in $\mathcal{C}_K$,*

$c_1 \leq^c c_2 \iff ext(c_1) \subseteq ext(c_2)$
(could be defined equivalently by $int(c_1) \sqsubseteq int(c_2)$*).*

Theorem 10 (concept lattice) *Let* K *be a context. The partially ordered set* $\langle \mathcal{C}_K; \leq^c \rangle$ *is a finite lattice, whose supremum and infimum are as follows for every set of indices* J*:*

- $\bigvee^c_{j\in J}(O_j, f_j) =^c (\tau_K(\sigma_K(\bigcup_{j\in J} O_j)), \sqcup_{j\in J} f_j)$
- $\bigwedge^c_{j\in J}(O_j, f_j) =^c (\bigcap_{j\in J} O_j, \sigma_K(\tau_K(\sqcap_{j\in J} f_j)))$



(a) (b)

Figure 1. The concept lattice of context K_{Triv} (a) and its labelling (b).

Example 11 (*Triv***)** *Figure 1.(a) represents the Hasse diagram of the concept lattice of context* K_{Triv} *(introduced in Example 6). Concepts are represented by a number and a box containing their extent on the left, and their intent on the right. The higher concepts are placed in the diagram the greater they are for partial order* $\leq^c$*. It can be observed that the concept lattice is not isomorphic to the power-set lattice of objects* $\langle 2^{\mathcal{O}}; \subseteq \rangle$*. Indeed, set* $\{x, y\}$ *is not the extent of any concept, because* $\tau(\sigma(\{x, y\})) = \tau(a \vee b) = \{x, y, z\}$*.*

To make the concept lattice more readable, it is possible to label it with formulas and objects. Mapping μ labels with a formula f the concept whose extent is the extent of f. Mapping γ labels with an object o the concept whose intent is the intent of o, that is its description.

Definition 12 (labelling) *Let* $K = (\mathcal{O}, \mathcal{L}, d)$ *be a context,* $o \in \mathcal{O}$*, and* $f \in \mathcal{L}$*,*

- μ_K *(*μ *for short)* $: \mathcal{L} \to \mathcal{C}_K,\ ext(\mu_K(f)) = \tau_K(f)$
- γ_K *(*γ *for short)* $: \mathcal{O} \to \mathcal{C}_K,\ int(\gamma_K(o)) \equiv d(o)$.

The interesting thing with this labelling is that it enables to retrieve all data of the context: an object o satisfies ($\sqsubseteq$) a formula f in some context K if and

only if the concept labelled with o is below ($\leq^c$) the concept labelled with f in the concept lattice of K.

Lemma 13 (labelling) *Under the conditions of Definition 12,*

$$d(o) \sqsubseteq f \iff \gamma_K(o) \leq^c \mu_K(f) \ .$$

Example 14 (*Triv*) *Figure 1.(b) represents the same concept lattice as Figure 1.(a) (see also Example 11), but its concepts are decorated with the μ labelling instead of with the extents and intents. Formulas of the form $\bigvee A$ where $A \subseteq \mathcal{A}$ ($\bigvee \emptyset \equiv 0$) are placed on the right of the concept that they label. For instance, concept 1 is labelled by formula a (i.e., $\mu(a) =^c 1$). In Figure 1.(b) we have restricted labels to be formulas of the form $\bigvee A$, but it is only to have a finite number of labels that are not all in the formal context.*

2.4 Contextualized subsumption

In most contexts, it is possible to order some properties, although they are not comparable by $\sqsubseteq$. For instance, if in some context every bird flies, then we can say that property "bird" is *contextually subsumed* by the property "fly", although we have not necessarily $bird \sqsubseteq fly$ in $\mathcal{L}$. We introduce a *contextualized subsumption* as a generalization of implications between attributes that are used in standard CA for knowledge acquisition processes (Ganter and Wille, 1999; Snelting, 1998).

Definition 15 (contextualized subsumption) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context, and $f, g \in \mathcal{L}$. One says that f is* contextually subsumed *by g in context K, which is noted $f \sqsubseteq_K g$, if and only if $\tau_K(f) \subseteq \tau_K(g)$, i.e., if every object that satisfies f also satisfies g.*

Every arc of $\sqsubseteq_K$ is called a contextualized implication.

Contextualized subsumption has a close connection with the concept lattice.

Theorem 16 (contextualized subsumption vs. concept lattice) *Let K be a context. The partially ordered set of formulas $\langle \mathcal{L}; \sqsubseteq_K \rangle$, derived in the usual way from the pre-order $\sqsubseteq_K$, is isomorphic to the concept lattice $\langle \mathcal{C}_K; \leq^c \rangle$. The morphism from formulas to concepts is μ_K (Definition 12); and the morphism from concepts to formulas is int (Definition 8).*

A context plays the role of a theory extending the subsumption relation and enabling new entailments. Contextualized subsumption can also be seen as a means for extracting knowledge from contexts. Thus, two kinds of knowledge can be extracted: knowledge about context by deduction, and knowledge on

the domain from which the context is extracted by induction (e.g., generalizing "every bird flies" from *bird* $\sqsubseteq_K$ *fly*).

Example 17 (*Triv*) *As the contextualized subsumption is isomorphic to the order on concepts (Theorem 16), it is possible to use the labelled concept lattice (see Figure 1.(b)) to study contextualized subsumption in context* K_{Triv}. *For instance, as concept* 2 *is smaller than concept* 5 *relation* $c \sqsubseteq_{K_{Triv}} b \vee c$ *stands, which is already true in* $\mathcal{P}$. *More generally, it can be seen that all valid subsumptions in* $\mathcal{P}$ *are retained in contextualized subsumption. Examination of the labelled concept lattice shows that the context adds new valid entailments between formulas: e.g.,* $c \sqsubseteq_{K_{Triv}} a \vee b$, *because* $2 \leq^c 4$.

In the following sections, formal contexts will be used to formalize the content of an information system, and the concept lattice (or equivalently, the contextualized subsumption) will be used to organize things. It is the contextualized equivalence relation that gives so many names to things.

2.5 Feature Context

Logical languages contain usually infinitely many formulas, whose complexity is unbounded. This is a problem for algorithms that perform a search among formulas (e.g., for automated learning, Ganter and Kuznetsov, 2001). For efficiency and readability of results, we restrict the search space of formulas to a finite subset $F \subseteq \mathcal{L}$ whose elements are called *features*. Features differ from attributes of standard formal contexts in three ways:

(1) features belong to a fixed logical language and so, have a semantics,
(2) features are automatically ordered according to the subsumption $\sqsubseteq$, and
(3) a newly introduced feature can have a non-empty extent.

It is possible to extract a formal context, with F as the set of attributes, from the logical context: we call it the *feature context*. This context is not intended to be actually build from the logical context, but it is defined to allow reasoning about the logical context with a coarser grain than the full logic.

Definition 18 (feature context) *Let* $K = (\mathcal{O}, \langle \mathcal{L}; \sqsubseteq \rangle, d)$ *be a logical context, and* $F_K \subseteq \mathcal{L}$ *be a finite set of features, that may depend on* K. *The* feature context *of* K *is the formal context* $K_F = (\mathcal{O}, F_K, I_{K_F})$, *where* $I_{K_F} = \{(o, x) \in \mathcal{O} \times F_K \mid d(o) \sqsubseteq x\}$. *We also define* description features *for any object* o *by* $D_{K_F}(o) = \uparrow_F d(o)$, *where for any* $f \in \mathcal{L}$, $\uparrow_F f = \{x \in F \mid f \sqsubseteq x\}$.

The content of F_K is not strictly determined but depends on the context K. It should contain simple formulas subsuming logical descriptions (in K), frequently used formulas (in queries), and more generally, every formulas that users expect to see in answers. In fact, F_K acts as the vocabulary that a LIS uses in its answers.

Example 19 ($Triv$) *From boolean descriptions (like in context K_{Triv}, see Example 6) one can consider as features the clauses (i.e., disjunctions) of the conjunctive normal form of the descriptions. Thus, the features of context K_{Triv} would be $F_{K_{Triv}} = \{a, b, c, a \vee b\}$, and relation I_{K_F} would be $\{(x, a), (x, a \vee b), (y, b), (y, a \vee b), (z, c), (z, a \vee b)\}$.*

From the description given in Example 7, one can extract the following features (amongst others): `author contains "Ridoux"`, `year in 1950..2000`.

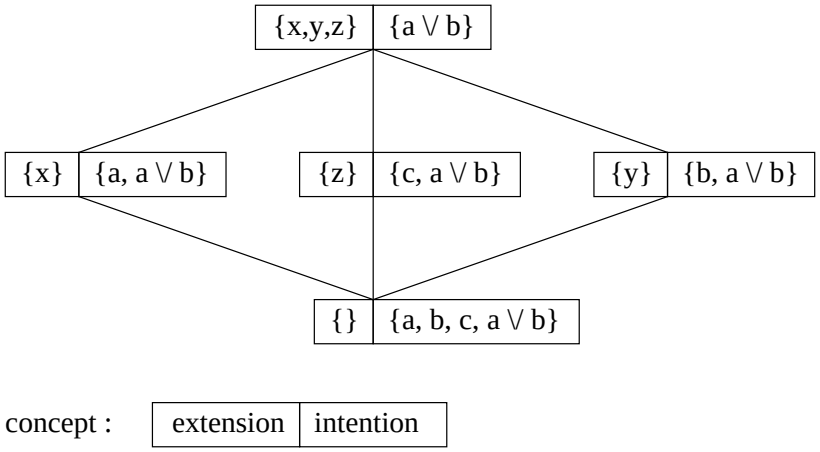


Figure 2. The feature concepts of context K_{Triv}

Feature concepts can be derived from a feature context as for ordinary logical contexts. See for instance Figure 2 for the lattice of feature concepts of context K_{Triv}.

Lemma 20 relates the Galois connections of logical and feature contexts.

Lemma 20 (logical vs. feature Galois connections)
Let $O \subseteq \mathcal{O}$, $X \subseteq F$,

- $\sigma_{K_F}(O) = \uparrow_F \sigma_K(O)$
- $\tau_{K_F}(X) = \tau_K(\sqcap X)$.

Theorem 21 shows the existence of a mapping that approximates a logical concept in a feature concept and then defines equivalence classes among logical concepts.

Theorem 21 (approximation of concept) *Let $(O, f) \in \mathcal{C}_K$ be a logical*

concept. The feature concept generated from O (intent: $\sigma_{K_F}(O)$) and the feature concept generated from f (extent: $\tau_{K_F}(\uparrow_F f)$) are in fact the same feature concept $(\tau_{K_F}(\uparrow_F f), \uparrow_F f)$, the smallest concept in $\mathcal{C}_{K_F}$ whose extent is larger than or equal to O.

2.6 Sub-context

It is often useful to reason on a *sub-context* by restricting the set of objects and the set of features. For instance, the need for *views* has been recognized in databases (Ullman, 1989) (see also Definition 31).

Definition 22 (sub-context) *Given a* domain $D \subseteq \mathcal{O}$*, restricting the set of objects, and a* view $V \subseteq F$*, restricting the set of features, we define the* sub-context *of a feature context K_F by the formal context $K_F(D, V) = (D, V, I_{K_F} \cap (D \times V))$.*

Example 23 ($Triv$) *We define 4 sub-context of K_{Triv} (see Example 6).*

(1) $D = \mathcal{O}$ *and* $V = F_{K_{Triv}} \backslash \{c\}$*:*
$I_{K_F} = \{(x, a), (x, a \vee b), (y, b), (y, a \vee b), (z, a \vee b)\}$.
(2) $D = \mathcal{O}$ *and* $V = \{c\}$*:*
$I_{K_F} = \{(z, c)\}$.
(3) $D = \mathcal{O} \backslash \{z\}$ *and* $V = F_{K_{Triv}}$*:*
$I_{K_F} = \{(x, a), (x, a \vee b), (y, b), (y, a \vee b)\}$.
(4) $D = \mathcal{O} \backslash \{z\}$ *and* $V = F_{K_{Triv}} \backslash \{c\}$*:*
$I_{K_F} = \{(x, a), (x, a \vee b), (y, b), (y, a \vee b)\}$.

Lemma 24 relates the Galois connections of feature contexts and sub-contexts.

Lemma 24 (feature vs. sub-context Galois connection) *Let* $O \subseteq D$*, and* $X \subseteq V$*,*

- $\sigma_{K_F(D,V)}(O) = \sigma_{K_F}(O) \cap V$,
- $\tau_{K_F(D,V)}(X) = \tau_{K_F}(X) \cap D$.

Example 25 ($Triv$) *The concept lattices of the four subcontexts of Example 23 are as in Figure 3.*

A domain can be specified by the extent $\tau_K(q)$ of a formula, i.e., the answers $\tau_K(q)$ to a query q. A view can be specified as the set of features subsumed by a query v, i.e., $\downarrow_F v = \{x \in F \mid x \sqsubseteq v\}$. E.g., in the logic presented in Example 7, the formula `(author contains "" ∨ title contains "concept" ∨ year in 1900..2000)` would select all features of attributes `author`, `title` and `year`, restricted to years in the last

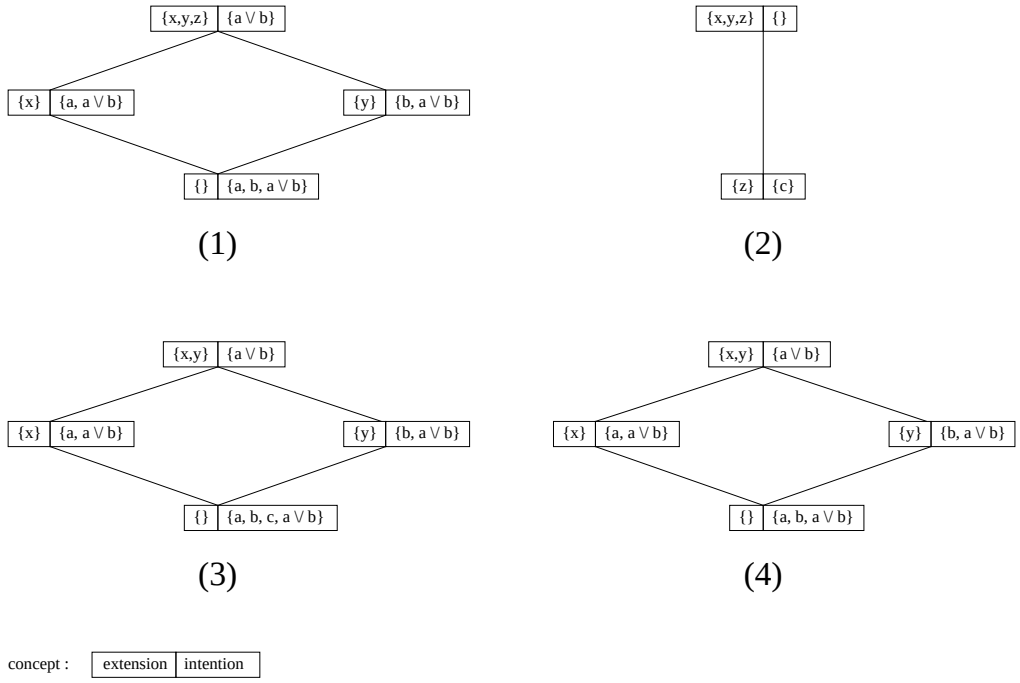


Figure 3. Four subconcept lattices of context K_{Triv}

century, and titles that contain the word "concept".

3 Navigating and Querying in a Logical Context

3.1 Navigating vs. Querying

Information systems offer means for organizing data, and for navigating and querying. Though navigation and querying are not always distinguished because both involve queries and answers, we believe they correspond to very different paradigms of human-machine communication. In fact, the difference can be clarified using the intent/extent duality.

Navigation implies a notion of place, and of a relation between places (e.g., file system directories, and links or subdirectory relations). Through navigation, a user may ask for the content of a place, or ask for related places. The ability to ask for related places implies that answers in the navigation-based paradigm belong to the same language as queries. In terms of the intent/extent duality, a query is an *intent*, and answers are *extents* for the content part, and *intents* for the related places.

In very casual terms, we consider navigation with possibly "no road-map", i.e., no *a priori* overview of the country. Related places form simply the land-

scape from a given place as shown by a "viewpoint indicator". However, our proposal is compatible with any kind of *a priori* knowledge from the user.

With querying, answers are extents only. A simulation of navigation is still possible, but forces the user to infer what could be a better query from the unsatisfactory answer to a previous query; i.e., infer an intent from an extent. This is difficult because there is no simple relation between a variation in the query, and the corresponding variation in the answer. The experience shows that facing a query whose extent is too vast, a user may try to refine it, but the resulting extent will often be either almost as vast as the former or much too small. In the first case, the query lacks of *precision* (i.e., number of relevant items in the answer divided by total number of items in the answer), whereas in the second case, the query *recall* (i.e., number of relevant items in the answer divided by number of relevant items in the system) is too low.

Godin et al. (1993) and Lindig (1995) have shown that Formal Concept Analysis is a good candidate for reconciliating navigation and querying. We follow this opinion, but we believe that care must be taken to make formal contexts as close to the description languages of the end-users, and we have proposed Logical Concept Analysis (LCA) where formal descriptions are logical formulas instead of being sets of attributes (Section 2.1).

Our goal in this section is to show how a form of navigation and querying can be defined, so that a user who knows neither the content of a Logical Information System, nor the logic of its descriptions, can navigate in it and discover the parts of the contents and the parts of the logic that are relevant to his quest. Note that a more expert user may know better and may navigate more directly to his goal, but since almost everybody has his shortcomings, the no-knowledge assumption is the safest one to do.

3.2 A Logical Information System

In this section, we will insist on navigation tools, and will delay the management of a logical context (e.g., creating, updating objects) until Section 4.

A LIS needs a user interface. We will formalize a shell-based interface, though this is not the most modern thing to do. This is because we believe that shell interfaces (like in UNIX or MS-DOS) are familiar to many of us, and because this abstraction level exposes properly the dialogue of queries and answers. A higher-level interface like a graphical one would hide it, whereas lower-level interfaces, like a file system, would expose irrelevant details. However, nothing prevents one to give a graphical interface to a LIS, or to implement it as a file system.

The shell commands are those of the UNIX shell, reinterpreted in the LIS framework. Main changes are the replacement of *paths* by formulas of $\mathcal{L}$ referring to concepts via mapping μ_K, and the use of contextualized subsumption $\sqsubseteq_K$. For the rest, commands have essentially the same effects. The correspondence between a UNIX file system and a logical information system is as follows:

$$\begin{aligned}
\text{UNIX shell} &\longrightarrow \text{LIS shell} \\
\text{file} &\longrightarrow \text{object} \\
\text{path} &\longrightarrow \text{logical formula} \\
\text{absolute name of a file} &\longrightarrow \text{object description} \\
\\
\text{directory} &\longrightarrow \text{formula/concept} \\
\text{root} &\longrightarrow \text{formula } \top\text{/concept } \top^c \\
\text{working directory} &\longrightarrow \text{working query/concept}
\end{aligned}$$

Navigation commands `cd`, `ls`, and `pwd` are defined in Section 3.3; and the querying command `ls -R` is defined in Section 3.4. Creation and update commands `touch`, `mkdir`, `rm`, `mv`, and `cp` will be defined in Sections 4.1 and 4.2.

3.3 Navigating in a Logical Context

Once objects have been logically described and recorded in a logical context $K = (\mathcal{O}, \mathcal{L}, d)$, one wants to retrieve them. One way to do this is *navigating* in the context. As already said, this way of searching is particularly useful in a context where the logic or the content are unknown. The aim of navigation is thus to guide the user from a *current place* to a *target place*, which contains the object(s) of interest. For this, a LIS offers to the user 3 basic operations (the corresponding UNIX-like command names are placed between parenthesis):

(1) to ask to LIS what is the current place (command `pwd`),
(2) to go in a certain "place" (command `cd` *place*),
(3) to ask to LIS effective ways towards other "places" (command `ls`).

3.3.1 Places as formal concepts

In a hierarchical file system, a "place" is a directory. In our case, a "place" is a formal concept, which can be seen as a coherent set of objects (extent) and

properties (intent) (see Definition 8). In large contexts, concepts cannot be referred to by enunciating either their extent or their intent, because both are generally too large. Formulas of the logic $\mathcal{L}$ can play this role because every formula refers to a concept through the labelling map μ (see Definition 12), and every concept is referred to by one or several formulas, which are often much concise than its intent. For instance in Figure 1, one can see that c is a concise name for the concept $(\{z\}, c \wedge (a \vee b))$.

We now describe the 3 navigation operations listed above. First of all, going from place to place implies to remember the current place, which corresponds to the working directory. In a LIS, we introduce the *working query*, wq, and the *working concept*, $wc := \mu_K(wq)$; we say that wq refers to wc. This working query is taken into account in the interpretation of most LIS commands, and it is initialized to the formula $\top$, which refers to the concept whose extent is the set of all objects. Command `pwd` displays the working query to the user.

The second navigation operation, command `cd`, takes as argument a query formula q saying in which place to go, and it changes the working query accordingly. We call l_{wq} (i.e., *elaboration* of wq) the mapping that associates to the query q a new working query according to the current working query wq. The query q can be seen as a *link* between the current and the new working query. Usually, `cd` is used to refine the working concept, i.e., to select a subset of its extent. In this case, the mapping l_{wq} is defined by $l_{wq}(q) := wq \sqcap q$, which is equivalently characterized by

$$\mu_K(l_{wq}(q)) =^c wc \wedge^c \mu_K(q) \text{ and } \tau_K(l_{wq}(q)) = \tau_K(wq) \cap \tau_K(q).$$

However, it is useful to allow for other interpretations of the query argument. For instance, we can allow for the distinction between *relative* and *absolute* queries, similarly to relative and absolute paths in file systems. The previous definition of the mapping l_{wq} concerns relative queries, but can be extended to handle absolute queries by $l_{wq}(/q) := q$, where '/' denotes the absolute interpretation of queries. This allows to forget the working query. We can also imagine less usual interpretations of queries like $l_{wq}(|q) := wq \sqcup q$. Finally, the special argument `..` for the command `cd` enables to go back in the history of visited queries/concepts. This works much like the "Back" button in *Web* browsers.

The last navigation operation, command `ls`, is intended to guide the user towards his goal. More precisely, it must suggest some relevant links that could act as queries for the command `cd` to refine the working query. These links are formulas of $\mathcal{L}$. A set of links given by `ls` should be finite, of course (whereas $\mathcal{L}$ is usually infinite), even small if possible, and complete for navigation (i.e., each object of the context must be accessible by navigating from $\top^c$).

3.3.2 Navigation Links

The following notion of *link* corresponds to the case where the elaboration mapping satisfies $l_{wq}(q) = wq \sqcap q$. To avoid to go in a concept whose extent is empty (a dead-end), we must impose the following condition on a link x: $\tau_K(wq \sqcap x) \neq \emptyset$. Furthermore, to avoid to go in a concept whose extent is equal to the extent of wq (a false start), we must impose this other condition: $\tau_K(wq \sqcap x) \neq \tau_K(wq)$. These conditions state that the extent of the new working query must be strictly between the empty set and the extent of the current working query. This characterizes relevant links (Lindig, 1995).

Now, as $\mathcal{L}$ is a too wide search space (it is often infinite), we will consider a finite set of features $F \subseteq \mathcal{L}$ in a context K in which links are selected.

Furthermore, we retain only greatest links (in the subsumption order) as they correspond to smallest navigation steps. Indeed, recall that each link is a suggestion to the user. Therefore, if the link `year in 1990..2010` is suggested, it is not worth suggesting the link `year = 2000`, because the former subsumes the latter. Following this principle, every conjunctive formula $x \sqcap y$ can be excluded from the search space for links, and so, of the set of features, because it is redundant with x and y. Thus, a way to build the set of features is to split object descriptions on outermost conjunctions (i.e., $a \wedge b$ is split into a and b, whereas $(c \wedge d) \vee e$ cannot be split this way).

However, to limit the number of links, it is useful to also extract abstracted form of these features, so as to allow a still more progressive navigation. For instance, the system would suggest the successive links `title`, `title contains "System"`, and then `title is "Logical Information Systems"`; instead of directly suggesting the latter. Only this latter feature appears explicitly in the description. The others are abstractions of it, and plays a role of "factorization" among links.

We now summarize this part by defining the set of links in a given context and working query.

Definition 26 (Navigation links) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* navigation links *for every working query $wq \in \mathcal{L}$ is defined by*

$$Link_K(wq) := Max_{\sqsubseteq}\{x \in F_K \mid \emptyset \neq \tau_K(wq \sqcap x) \neq \tau_K(wq)\}.$$

3.3.3 Local Objects and Navigation Completeness

As navigation aims at finding objects, command `ls` must not only suggest some links to other places, but also present the objects belonging to the current place, called the *objects of wq* or the *local objects*. We define a local object

as an object that is in the current place, but in no place reachable through a link.

Definition 27 (local object) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* local objects *is defined for every working query $wq \in \mathcal{L}$ by*

$$Local_K(wq) := \tau_K(wq) \setminus \bigcup_{x \in Link_K(wq)} \tau_K(x).$$

Given a place, there can be no local object, but there can also be several objects in some place. The less local objects there are, the better it is for navigation. More precisely, if two local objects have non-equivalent descriptions, it should be possible to make this difference appear in the links. Thus, the choice the user has to do is intentional (between one or several logical links) rather than extensional (between several objects). This idea is formalized as the *completeness* of navigation.

Definition 28 (navigation completeness) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. Navigation is* complete *in K if and only if for every working query $wq \in \mathcal{L}$, the following holds*

$$\forall o, o' \in Local_K(wq) : d(o) \equiv d(o').$$

This completeness can be guaranteed by ensuring that the set of features F_K is such that if two objects have non-equivalent descriptions there exists a feature that is satisfied by one object and not by the other.

Theorem 29 (navigation completeness) *A necessary and sufficient condition for navigation completeness in a context K is that for every objects o, o'*

$$d(o) \not\equiv d(o') \Rightarrow \exists x \in F_K : d(o) \sqsubseteq x \Leftrightarrow d(o') \not\sqsubseteq x.$$

In the case where all objects have different descriptions, there is never more than one local object. This must be compared to *Web* querying where the number of objects returned in response to a query is generally large. This is because with navigation, non-local objects are hidden behind the intentional properties that enable to distinguish these objects. It is the end-user who selects an intentional property to reveal its content.

Another interesting thing to notice is that the working query can be, and often is, much shorter than the whole description of the local object (which is also the intent of the working concept), as in the following example where the first formula is contextually equivalent (in context *Bib*, see Example 7) to the

second one for accessing the object.

`author contains "Mineau"` $\wedge$ `author contains "Missaoui"`

$\equiv$

`author is "Mineau, Missaoui"` $\wedge$ `title is "The Representation of Semantic Constraints in Conceptual Graph Systems"` $\wedge$ `type is "InProceedings"` $\wedge$ `year in 1997`

3.3.4 Links and Views

Even if the set of links is restricted to relevant and greatest ones among features, it appears in practice that it is too large and heterogeneous. For instance, in the context *Bib* (see Example 7), the set of links is a mix of author names, title words, etc. Our idea is to abstract the set of all author names (features `author contains ...`) by a single feature `author` (meaning "the attribute `author` is defined"). This feature is not useful for selecting objects, but it is useful to select links about the attribute `author`. We call this kind of features *views* as they present the navigation from a "point of view". We introduce a working view wv, similar to the working query, under which links must be searched for. For instance, if the working view is (`author`), links will be author names. We take it into account in the definition of navigation links, where the relation $x \sqsubset wv$ denotes a strict subsumption (i.e., $x \sqsubseteq wv$ and $wv \not\sqsubseteq x$).

Definition 30 (Navigation links with views) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. For every working query $wq \in \mathcal{L}$ and every working view $wv \in \mathcal{L}$, the set of* navigation links *is defined by*

$$Link_K(wq, wv) := Max_{\sqsubseteq}\{x \in F_K \mid x \sqsubset wv, \emptyset \neq \tau_K(wq \sqcap x) \neq \tau_K(wq)\}.$$

As a link is a variation of the working query that restricts the current extent, one defines a *view* as a variation of the working view that restricts the set of links. Moreover, if a view is not also a link, it must subsume at least two links. Indeed, if some view hides only one link, it is worth presenting the link directly. Finally, we define a set of links and views where views can be understood as summaries of sets of links, whose selection by the user allows him to see these underlying links in a narrower view.

Definition 31 (Navigation links and views) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. The set of* navigation links and views *for every working query $wq \in \mathcal{L}$ and every working view $wv \in \mathcal{L}$ is defined by ($\|E\|$ denotes the cardinality of*

a set E)

$$LV_K(wq, wv) := Max_{\sqsubseteq}\{x \in F_K \mid x \sqsubset wv,\ \ \emptyset \neq \tau_K(wq \sqcap x) \neq \tau_K(wq) \\ or\ \ \|Link_K(wq, wv \sqcap x)\| \geq 2\}.$$

To summarize, the view-based variant of command `ls` takes as argument a view v, sets the working view to $l_{wv}(v)$ (where l_{wv} works similarly to l_{wq}), shows the local objects if they exist, displays each link or view x of the set $LV_K(wq, wv)$ along with the size of its selected extent $\tau_K(wq \sqcap x)$, and finally displays the size of the working extent $\tau_K(wq)$. Links and views are distinguished according to their cardinality compared to the size of the working query; views simply have the same size as the working concept, whereas links have strictly smaller sizes.

3.3.5 User/LIS Dialogue

We now show how commands `cd` and `ls` compose a rather natural dialogue between the user and LIS. The user can refine the working concept with command `cd`, and asks for suggested links and views with the command `ls`. LIS displays to the user relevant links for forthcoming `cd`'s, and relevant views for forthcoming `ls`'s. With commands `cd` and `ls`, and links and views, both the user and a LIS can assert facts and ask questions;

- Command `cd` and links are *assertions* from the user and LIS.
  user: `cd kind` (i.e., "I want this kind of object!"),
  LIS: links to `kind` (i.e., "I have this kind of object!").
- Command `ls` and views are *questions* from the user and LIS.
  user: `ls kind` (i.e., "What kind of object do you have?"),
  LIS: view `kind` (i.e., "What kind of object do you want?").

It should also be noticed that both the user and LIS can answer to questions both by assertions and by questions.

Example 32 (*Bib*) *A complete example of a dialogue is given in Table 1. The left part of this table shows what is really displayed by our prototype, and the right part is an english translation of the dialogue. Notice that this translation is rather systematic and could be made automatic.* ***(n)*** *is the prompt for the n-th query from the user. On the 2nd query, the question of the user is so open, that LIS only answers by questions. On the 3rd query, the user replies to one of these questions (*`title`*) by an assertion; but on the 4th query, he sends back to LIS another of these questions (*`author`*) to get some relevant suggestions. On the 5th query, he just selects a suggested author,* `"Wille"`*, and then gets his co-authors on Concept Analysis with the 6th query. On the*

7th query, he selects a co-author and finally finds an object at the 8th query.

Table 1
Example of User/LIS Dialogue in context *Bib*.

(1) pwd	(1) What is currently selected?
1	All objects.
(2) ls	(2) What do you have?
209 type	What kind of type do you want?
209 author	What kind of author do you want?
209 year	What kind of year do you want?
209 title	What kind of title do you want?
209 object(s)	209 objects are currently selected.
(3) cd title contains "Concept A"	(3) I want objects whose title contains "Concept A"!
(4) ls author	(4) What kind of author do you have (for this)?
1 author contains "Mineau"	I have 1 object with author "Mineau"!
1 author contains "Lehmann"	I have 1 object with author "Lehmann"!
1 author contains "Stumme"	I have 1 object with author "Stumme"!
1 author contains "Prediger"	I have 1 object with author "Prediger"!
3 author contains "Wille"	I have 3 objects with author "Wille"!
4 object(s)	4 objects are currently selected.
(5) cd author contains "Wille"	(5) I want objects with author "Wille"!
(6) ls	(6) What kind of author do you have (now)?
1 author contains "Mineau"	I have 1 object with author "Mineau"!
1 author contains "Lehmann"	I have 1 object with author "Lehmann"!
1 author contains "Stumme"	I have 1 object with author "Stumme"!
3 author contains "Wille"	What kind of author "Wille" do you want?
3 object(s)	3 objects are currently selected.
(7) cd author contains "Mineau"	(7) I want objects with author "Mineau"!
(8) ls	(8) What do you have?
#200 Mineau, Stumme, Wille. Conceptual Structures Represented by Conceptual Graphs and Formal Concept Analysis. INPROC, 1999.	
1 object(s)	1 object is currently selected.
(9) pwd	(9) What is currently selected?
author contains "Wille" $\wedge$	Objects with authors "Wille"
author contains "Mineau" $\wedge$	and "Mineau",
title contains "Concept A"	and whose title contains "Concept A".

3.4 Querying a Logical Context

Extensional queries, as in data-bases or some *Web* browsers like Google, can be submitted to a logical information system using the `-R` option with command `ls`. The answer to query `ls -R q` is simply $\tau_K(l_{wq}(q))$, i.e., the extent

of the concept referred to by $l_{wq}(q)$ (see Section 3.3).

```
(1) ls -R /title contains "Logic" ∧ ¬ title contains "Concept" ∧
year in 1990..1995
#3 Gaines. Representation, discourse, logic and truth: situating
knowledge technology. INPROC, 1993.
#2 Sowa. Relating diagrams to logic. INPROC, 1993.
#72 Van den Berg. Existential Graphs and Dynamic Predicate Logic.
INPROC, 1995.
3 object(s)
```

4 Creating and Updating a Logical Context

Whereas much has been said on the construction of concept lattices (Kuznetsov and Objedkov, 2001), the construction of contexts is often left in the background. The construction process can fall into two categories: off-line and on-line. In the off-line case, the context is built once for all after the data have been gathered and the problem is to find an object description language appropriate to the intended analysis. The typical application of this category is the analysis of surveys. In the on-line case, the context is built progressively along the arrival of data and a problem is to properly describe new objects at the time they arrive. Information systems are the typical application of this category, but this is not the mainstream approach to using Concept Analysis.

Hypothesis 1 (on-line construction) *We consider here only the on-line case, as we focus on information systems. For each new piece of data that arrives, an object is created, and added to the context, with this piece of data as content.*

We propose that the description given to an object is two-parts. The first part, the *intrinsic description*, is automatically extracted from the object content, and depends on the kind of content and on the logic of the application. For instance, let us consider that objects are incoming e-mail messages. In this application, the building of the context is clearly on-line; and possible components of the intrinsic description are the `from`, `to`, and `subject` fields.

The second part, the *extrinsic description*, is manually assigned by users according to personal intentions and preferences. We must consider there are no known rules to infer extrinsic properties, as if the contrary holds they could be integrated in the intrinsic description. In a usual e-mail application, extrinsic properties are managed by storing e-mail messages in different folders accord-

ing to personal needs. However, extrinsic properties need not be organized in a hierarchical relation as folders often are.

4.1 Creating a Logical Context

The objects we want to represent in a context are often defined in the "world" by a *content*. We want to describe these objects both by an *intrinsic* properties (automatically extracted from their contents), and by *extrinsic* properties (manually assigned by users). All these elements are the *context data*, from which a logical context can be built.

Definition 33 (context data) Context data *are defined as a 6-tuple* $D = (\mathcal{O}, \mathcal{C}, \mathcal{L}, c, d_i, d_e)$*, where*

- $\mathcal{O}$ *is a set of* objects*,*
- $\mathcal{C}$ *is the domain of* contents*,*
- $\mathcal{L}$ *is a* logic*,*
- $c \in \mathcal{O} \to \mathcal{C}$ *maps every object to its content,*
- $d_i \in \mathcal{C} \to \mathcal{L}$ *extracts an* intrinsic description *from every content,*
- $d_e \in \mathcal{O} \to \mathcal{L}$ *maps every object to its* extrinsic description*.*

The building of a context from its data consists in keeping the set of objects and the logic, and composing a description that maps every object to its intrinsic description "plus" its extrinsic description. This "plus" denotes an *update* operation $\diamond$ that we present in more details in Section 4.2.

Definition 34 (context building) *Let* $D = (\mathcal{O}, \mathcal{C}, \mathcal{L}, c, d_i, d_e)$ *be context data. The* context built from data D *is defined as*

$$K(D) = (\mathcal{O}, \mathcal{L}, d), \text{where for every } o \in \mathcal{O}, d(o) = d_i(c(o)) \diamond d_e(o).$$

Example 35 ($E - Mail$) *In this section, we consider that objects are e-mail messages. An example of a simplified e-mail message content is:*

```
From: Alice@paris.fr
To: Bob@berlin.de, Chloe@madrid.es
Date: 2 May 2002, 14:52
Subject: Hello world!

When do you come in Paris?

See you,
Alice
```

From such a content, an intrinsic description that retains only fields `from`*,*

`to`, and `subject` can be extracted and formulated in logic $\mathcal{P}_v$ (see Example 7). This is only a matter of choice, as other fields could be taken into account. In particular, the message body would certainly be useful in a real application, but it is not necessary to our explanations. The intrinsic description we obtain from the above content is the following formula.

```
from is "Alice@paris.fr"
∧ to is "Bob@berlin.de, Chloe@madrid.es"
∧ subject is "Hello world!"
```

Then, the user (e.g., Bob) can add his personal comments by formulating an extrinsic description. For instance, the extrinsic description

```
personal ∧ ¬spam
```

means the message is "personal" (opposite of "professional"), and is not a "spam". The expected result of updating the intrinsic description with the extrinsic description is

```
from is "Alice@paris.fr"
∧ to is "Bob@berlin.de, Chloe@madrid.es"
∧ subject is "Hello world!"
∧ personal ∧ ¬spam
```

The LIS shell commands for creating new entries in a context are `mkdir` and `touch`. Command `mkdir` creates a new feature. For instance, `mkdir subject begins with "H"` introduces a finer feature than those that are obtained by simply splitting conjunctions. Command `touch` simply creates an object with empty content at some place designated by a formula. However, objects are normally created by applications that give objects a content (and so, an intrinsic description) and an extrinsic description, which is usually based on the working query.

4.2 Updating a Logical Context

From the definition of context data (see Definition 33), a logical context can be updated in 4 ways:

(1) addition of an object with its initial content and extrinsic description (commands `touch`, `cp`, and applications),
(2) update of the content of an object (applications),
(3) update of the extrinsic description of an object (command `mv`), and
(4) deletion of an object (command `rm`).

The only difficulty that arises from these operations is the *update* operation, which is used to change an extrinsic description in an incremental way (i.e., without completely redefining it), and to compose intrinsic and extrinsic descriptions. A naive solution would be to manually change extrinsic descriptions, and to directly integrate them in intrinsic descriptions. However, this has two important drawbacks. First, the intrinsic description changes with the contents and so, it is not a persistent support for extrinsic properties. Second, practice shows that one often wishes to express an update operation for a set of objects in a single command: for instance, add property "personal" (`personal`) to all e-mail messages sent from "Alice" (i.e., `from contains "Alice"`).

We give now a specification of such an operation, as it is given by Herzig and Rifi (Herzig and Rifi, 1999). This puts constraints on what a logical update operation should be.

Definition 36 (update) *Let $\mathcal{L} = (L, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ be a logic. The result of* updating *a description $d \in L$ by an entry $e \in L$ is the result of an operation $d \diamond e$, which must satisfy the following postulates:*

(1) (HR) $d \diamond e \sqsubseteq e$;
(2) (HR) $d \sqcap e \sqsubseteq d \diamond e$;
(3) (HR) $d \diamond \top \equiv d$;
(4) (HR) $d \not\sqsubseteq \bot$ and $e \not\sqsubseteq \bot$ implies $d \diamond e \not\sqsubseteq \bot$;
(5) (HR) for every $d' \equiv d$, $d' \diamond e \equiv d \diamond e$;
(6) (HR) for every $e' \equiv e$, $d \diamond e' \equiv d \diamond e$;
(7) (HR) for every $d_1, d_2 \in L$ such that $d \equiv d_1 \sqcup d_2$, $d \diamond e \equiv (d_1 \diamond e) \sqcup (d_2 \diamond e)$;
(8) (involution) $(d \diamond e) \diamond e \equiv d \diamond e$.

All postulates are from Herzig and Rifi, except postulate 8. Postulate 1 means that the entry must always be taken into account, and so, satisfied in the result. Postulate 2 adds that everything satisfied in the result comes from either the description d or the entry e. In the case where the entry is the tautology, i.e., brings no information, postulate 3 states that the description must be kept unchanged. Postulate 4 forces the result to be consistent, unless the description or the entry is already inconsistent. Postulates 5 and 6 mean that results must not depend on the syntax of formulas. Finally, postulate 7 says that operation $\diamond$ must be distributive for disjunction, and postulate 8 says that it must be an involution.

Herzig and Rifi (1999) present an update operation for propositional logic with dependencies between atoms, denoted by $WSS \downarrow^{dep}$, that satisfies all postulates in Definition 36. In logic, dependency between atoms is subsumption.

Example 37 (*Bib*) *Logic $\mathcal{P}_v$ is a propositional logic, whose usual atoms are replaced by valued attributes. This means that we must consider dependencies between such atoms. For instance, the atom*

`from is "Alice@paris.fr"` *implies the atom* `from contains "Alice"`*, contradicts the atom* `from is "Bob@berlin.de"`*, but is independent from the atom* `subject contains "Hello"`*.*

The LIS shell commands `rm`, `mv` and `cp` perform LIS updates. Command `rm q` suppresses the local object of $l_{wq}(q)$ (if it exists, see Section 3.3.3). Command `mv` moves a file from a place to another: `mv q e` moves the local object o of $l_{wq}(q)$ in concept $\mu(d(o) \diamond e)$. Similarly, `cp q e` creates a new copy of the local object in the same concept. With option `-r`, every object of the extent of $l_{wq}(q)$ is concerned, instead of only the local object.

```
(1) cd /from contains "Alice"
(2) mv . personal
(3) rm -r /spam
```

The move command (line 2) adds the feature `personal` to the local object of the working query `from contains "Alice"`. The remove command (line 3) deletes all objects described as spam.

Contents can also be changed by applications. This changes indirectly descriptions (their intrinsic parts), but the ensuing reorganization of the formal concept lattice is automatic and transparent. In fact, it costs not so much since the concept lattice is not actually represented (see Section 6).

5 Data-mining, automated updating and learning

Previous sections 3 and 4 present the core operations of a LIS. More operations can be defined. Some are derived from core operations, like a form of data-mining, and others are disjoint from the core but can be added to it.

5.1 Data-mining

The definition of links (see Definitions 3.3.2 and following) is a specific case of Knowledge Discovery in a formal context. It can be generalized to recover more classical KD operations like machine-learning through the computation of necessary or sufficient properties (modulo some confidence), or data-mining through association rules. Indeed, CA has been often applied in domains such as *data-analysis*, *data mining*, and *learning*.

Data-analysis consists in structuring data in order to help their understanding. These data are often received as tables or relations and structured by partitions, hierarchies, or lattices. With CA, formal contexts (binary relations

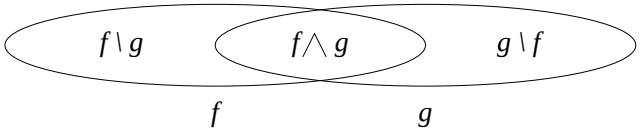
between objects and attributes) are structured in concept lattices (Ganter and Wille, 1999). This is applied for instance in software engineering for configuration analysis (Krone and Snelting, 1994). *Data-mining* is used to extract properties from large amounts of data. These properties are association rules satisfied (exactly or approximately) by the data. This is analogous to implications between attributes in FCA (see Ganter and Wille, 1999, p. 79), and to contextualized subsumption in LCA (see Section 2.4). *Unsupervised learning* is similar to data-analysis in the sense that one tries to discover some properties, and to understand some data, whereas *supervised learning* is similar to data-mining as some rules are searched to explain a target property using known properties. For instance, Kuznetsov applied CA to the learning of a positive/negative property from positive and negative instances (Kuznetsov, 1999).

The aim of this section is to show that these features of Knowledge Discovery (KD) can be incorporated in LIS, and how.

A context K plays the role of a theory by extending the subsumption relation and enabling new entailments (e.g., $bird \sqsubseteq_K fly$ when every bird flies in the context). All these contextual entailments are gathered with logical entailments to form the contextualized logic, which is thus a means for extracting some knowledge from the context. Two kinds of knowledge can be extracted: knowledge about the context by deduction ("Every bird of this context *do* fly"), and knowledge about the domain (which the context belongs to) by induction ("Every bird of the domain *may* fly").

Concept lattices produced by data-analysis are isomorphic to contextualized logics (see Theorem 16). Associations rules produced by data-mining or supervised learning match the contextualized subsumption relation, possibly qualified by a confidence defined by $conf(f \sqsubseteq_K g) = \frac{\|\tau_K(f) \cap \tau_K(g)\|}{\|\tau_K(f)\|}$ and a support defined by $supp(f \sqsubseteq_K g) = \|\tau_K(f) \cap \tau_K(g)\|$.

Considering two properties $f, g \in \mathcal{L}$, their contextual relation is determined by the cardinalities of 3 sets of objects: $\pi^l_K(f,g) := \|\tau_K(f) \setminus \tau_K(g)\|$, $\pi^c_K(f,g) := \|\tau_K(f) \cap \tau_K(g)\|$ and $\pi^r_K(f,g) := \|\tau_K(g) \setminus \tau_K(f)\|$. For instance, f contextually entails g if and only if $\pi^l_K(f,g) = 0$, f and g are contextually separated if and only if $\pi^c_K(f,g) = 0$, or x is a link of wq (see Section 3.3.2) if and only if $\pi^c_K(x, wq) \neq 0$ and $\pi^r_K(x, wq) \neq 0$. Note that the superscripts, l, c, and r refer to the left, center, and right part of the following Venn diagram:



So, the procedure that computes links for navigating in a LIS can be generalized to compute necessary or sufficient conditions, association rules, and links, only by specifying constraints on π^c_K, π^r_K, and π^l_K. Observe that $conf(f \sqsubseteq_K g) = \frac{\pi^c_K(f,g)}{\pi^l_K(f,g)+\pi^c_K(f,g)}$, and that $supp(f \sqsubseteq_K g) = \pi^c_K(f,g)$.

Definition 38 (Data-mining) *Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a context. For every working query $wq \in \mathcal{L}$ and working view $wv \in \mathcal{L}$ a set of facts is defined by*

$$Fact_K(wq, wv) := Max_{\sqsubseteq} \left\{ \begin{array}{r} (x, rank(x, wq)) \in F_K \times Rank \\ \mid x \sqsubset wv, property(x, wq) \end{array} \right\},$$

where Rank is a domain of ranking values, and rank and property are an application and a predicate defined using π^c_K, π^r_K, and π^l_K.

Example 39 (Generalized navigation)

Links: *Consider*
$rank(x, wq) = \pi^c_K(x, wq)$ *and* $property(x, wq) = \pi^c_K(x, wq) > 0$.
Then, $Fact_K(wq, wv)$ computes all links from working query wq under view wv.

Necessary conditions: *Consider*
$rank(x, wq) = conf(wq \sqsubseteq_K x)$
and $property(x, wq) = conf(wq \sqsubseteq_K x) \geq conf_{min}$,
where $conf_{min} \in [0, 1]$.
Then, $Fact_K(wq, wv)$ computes all necessary properties *to wq (i.e., properties entailed by wq) with a confidence greater than $conf_{min}$.*

Sufficient conditions: *Consider*
$rank(x, wq) = (supp(x \sqsubseteq_K wq), conf(x \sqsubseteq_K wq))$
and $property(x, wq) = supp(x \sqsubseteq_K wq) \geq supp_{min} \wedge conf(x \sqsubseteq_K wq) \geq conf_{min}$,
where $supp_{min}, conf_{min} \in [0, 1]$.
Then, $Fact_K(wq, wv)$ computes all sufficient conditions *to be in the extent of wq with a support and confidence greater than $supp_{min}$ and $conf_{min}$.*

5.2 Context Maintenance based on Learning

A context associates to objects a description that combines automatically extracted properties (*intrinsic*) and manually assigned ones (*extrinsic*) (cf. Section 4). The extrinsic properties are expressed by users according to intentions that are often subjective and changing, and that determine the classification and retrieval of objects. So, we believe it is important to assist users in this task through the automatic suggestion of extrinsic properties to be assigned

and even the discovery of rules to automate these assignments. The principle is to learn from the relationship between extrinsic and intrinsic descriptions of existing objects the extrinsic description of a new object whose intrinsic description is computed from its content. Because of the changing nature of users' intentions, the assistance given in the incremental building of a logical context must be interactive. We present formal principles, and an application to the classification of e-mail messages. Proofs can be found in (Ferré and Ridoux, 2002b).

5.2.1 Induction through Associative Concepts

Let us consider the situation where a new object o^* is added to a logical context $K = (\mathcal{O}, \mathcal{L}, d)$ along with an intrinsic description $d^*(o^*)$ to form a new context $K^* = (\mathcal{O} \uplus \{o^*\}, \mathcal{L}, d^*)$ with $d^*(o) = d(o)$ for all $o \in \mathcal{O}$. Our aim is to induce from the old context K a set of extrinsic properties $Ind_{K_F}(o^*) \subseteq F$ for the new object.

We first define *associative concepts* as the concepts of K when one considers only description features of o^*, $D_{K_F}(o^*)$.

Definition 40 (associative concept) *A non-empty concept of the subcontext $K_F(\mathcal{O}, D_{K_F}(o^*))$ is called an* associative concept *of o^* in K_F. The set of all such associative concepts is denoted by $AC_{K_F}(o^*)$.*

$AC_{K_F}(o^*)$ organizes the feature context K_F in a concept lattice (where the empty concept is missing) that is less finely detailed than $\mathcal{C}_{K_F}$. However, this coarser concept lattice is relevant to the features of o^*. Conversely, the finer details in $\mathcal{C}_{K_F}$ cannot be expressed with the features of o^*.

Then, an *induced feature* can be defined as a feature that contextually subsumes the intent of some associative concepts.

Definition 41 (induced property) *We say a feature x is an induced property for o^* if and only if there exists an associative concept $c \in AC_{K_F}(o^*)$, such that $\sqcap int(c) \sqsubseteq_K x$ or, equivalently, $ext(c) \subseteq \tau_K(x)$. $Ind_{K_F}(o^*)$ denotes the set of all induced properties of o^* in K_F.*

Intuitively, an associative concept c of a new object o^* is an already existing concept (for previous objects in K) that has some similarity with the description of o^*. When $x \in Ind_{K_F}(o^*)$ is induced from an associative concept c, $ext(c)$ is the *support* of the induction, and $int(c)$ is the *explanation*. A given associative concept can induce several features; and a given feature can be induced by several associative concepts, and so, it can have several explanations. Induced features that do not belong to description features are called *expected features*, which are the features suggested to users as extrinsic properties.

Example 42 (*Triv*) *Consider* $d(o^*) = c \wedge d$, *then* $D_{K_F}(o^*) = \{c, d\}$. $ac = (\{z\}, \{c\})$ *is an associative concept of* o^*, *and* $a \vee b$ *is a feature of* K_{Triv} *(see Figure 2) such that* $ext(ac) \subseteq \{x, y, z\} = \tau_{K_{Triv}}(a \vee b)$. *So,* $a \vee b$ *is an induced feature.*

5.2.2 Experimentation

The aim of this section is to present through experimentations the kind of interactions that help a user to assign extrinsic properties to incoming objects, and to gradually automate these assignments (e.g., for filtering spams).

5.2.2.1 Filtering spams We consider here the assisted filtering of spams. The following display shows the initial description of a new (non-solicited) e-mail message, with its expected features. The context on which the induction of expected features is based is made of 200 e-mail messages.

```
Current description:
   from is "hh2732774@dtcom.net" ∧
   to is "undisclosed-recipients" ∧
   subject is "earn money without a job!"

Expected features:
28 spam
    <- from contains "net" ∧ to is "undisclosed-recipients"
    <- to is "undisclosed-recipients"
    <- from contains "net"
    <- subject contains "earn" ∧ subject contains "money"
2 to contains "irisa"/ to contains "fr"/ from contains "com"
    <- subject contains "earn" ∧ subject contains "money"
...
```

We can see that, whereas the `from` and `subject` field are new, several features are induced. This is possible because message fields are split into words, which enables to find common features between the new object and existing ones: e.g., `from contains "net"`, `subject contains "money"`, `to is "undisclosed-recipients"`.

We also see that the above message is well-recognized as a spam, and this feature is even the most strongly induced one with several explanations and a weight of 28 supporting objects. These explanations suggest some rules such as "every e-mail message sent to `undisclosed-recipients` is a spam", or "every e-mail message whose subject contains words `earn` and `money` is a spam".

It is up to the user to validate these explanations, and to make them automatic rules, or to be cautious, and to consider them only as hints.

Figure 4 shows the filtering rate of spams during the incremental building of an e-mail context. The dashed line represents the rate of well-classified messages (as spam or non-spam) at the n-th insertion. The solid line represents the rate of classified messages (well or not). So, the part between the two lines represents badly classified messages (e.g., spam classified as non-spam), and the part above the solid line represents non-classified messages. This plot shows that after a transient phase of about 50 messages, the rate of well classified messages steadily reaches 85%, and there are nearly no bad classification.

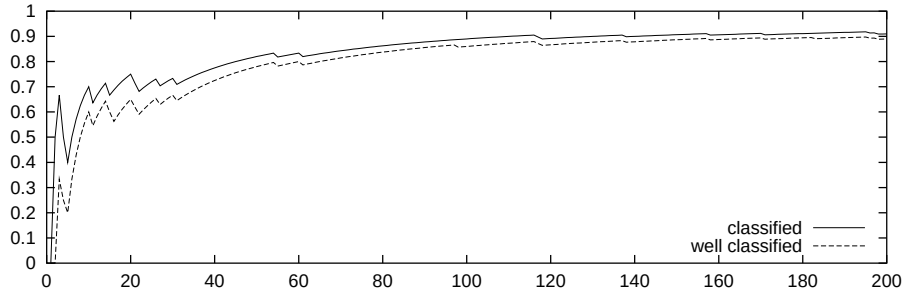


Figure 4. Filtering spams

The rates in Figure 4 are accumulated from the start of the experiment: $classified(n) = \frac{total\ classified(n)}{n}$. So they take into account the bad rates of the transient phase. The pseudo-instantaneous rates, $classified(n) = \frac{total\ classified(n) - total\ classified(n - \delta t)}{\delta t}$ are over 90% after the 80^{th} message for a time window (δt) of 50 messages (average number of messages per week during the experiment).

5.2.2.2 Classifying e-mail messages The second application is a variant of the first one in which keywords are not limited to two values. We classify e-mail messages in about 20 non-exclusive categories such as teaching, research, spam, call-for-paper, and so on. Note that these categories were not fixed a priori, but appeared only when required by the meaning of incoming messages. Thus, the vocabulary of categories remains open for ever.

Figure 5 shows the results of this experiment. The "automatic" line shows the rate of automatic classification using rules that are suggested by the system and accepted by the user (40 rules, including 15 for spams). The "suggested" line shows the rate of correct suggested classification. It tends to decrease simply because the sum of the two rates must be less than 1. Both rates are measured in number of features. The solid line shows the sum of the two rates.

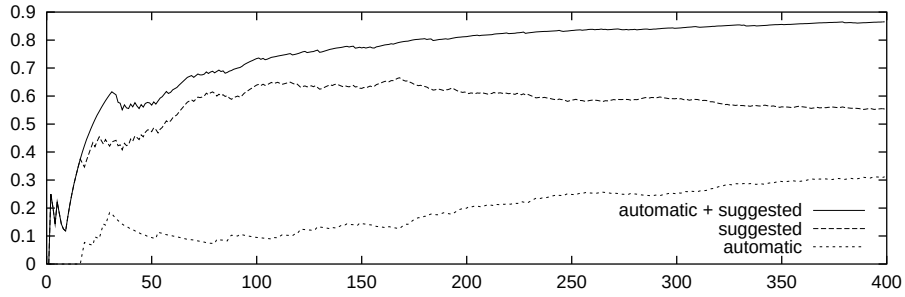


Figure 5. Classifying e-mail messages

Note again that these rates are cumulative; the instantaneous rates of "automatic+suggested" is constantly over 85% after the 100^{th} message for a time window of 50 messages.

6 Implementation of a LIS

Logic information systems are based on two choices: Concept Analysis and the use of an arbitrary logic. Both choices ask for performance questions as the complexity of concept lattices is exponential, and using logical descriptions implies the introduction of a theorem prover. However, theorem proving may be costly, e.g., exponential time for propositional logic.

6.1 Concept analysis without concept lattice

We first present the internal representation of a logical context we have chosen for our prototype; we also describe operations on this representation. Then, we give space complexity for the internal representation, and time complexity for the operations. These complexities are based on hypotheses, justified by experiments.

6.1.1 Internal Representation of a Logical Context

In a logical context $K = (\mathcal{O}, \mathcal{L}, d)$, we are mainly interested in navigation and querying. From their definition in Sections 3.3 and 3.4, it appears that all needed elementary operations are:

- $\sqsubseteq$: *subsumption* test between two formulas,
- τ_K: *extent* of a formula, which is based on subsumption $\sqsubseteq$ and on the set of objects $\mathcal{O}$,
- $\cap$: *set intersection* (between two extents),

- $Max_{\sqsubseteq}$: *selection* of maximal formulas w.r.t. $\sqsubseteq$, from a set of formulas.

Three of these operations use the subsumption test, i.e., they call the theorem prover of the logic. As its complexity is high for some logics, it is preferable to limit its use as much as possible. In particular, the computation of extents is intensively used in the search for links, whereas each computation of an extent means as many subsumption tests as the number of objects.

We propose to store subsumption relations between features, and between features and object descriptions, in what we call a *logical cache*. We also propose to cache extents of features, which are intensively used for navigation. Thus, a logical context K is stored as a directed acyclic graph whose nodes are logical formulas (features F_K and object descriptions $d(\mathcal{O})$), equipped with their extent, and whose arcs are subsumption relations between these formulas (arcs deducible by reflexivity and transitivity are not represented). This representation is the *Hasse diagram* of the partially ordered set $\langle F_K \cup d(\mathcal{O}); \sqsubseteq \rangle$.

Note that this representation is different from the concept lattice, which is usually used in systems based on concept analysis (e.g., Godin, Missaoui, and April, 1993; Cole and Stumme, 2000). We think the logical Hasse diagram is more appropriate to logical navigation for several reasons:

- the order used to compare a potential link to the working view and to select maximal links is the subsumption $\sqsubseteq$, and not the order on concepts $\leq^c$,
- it allows to search for navigation links among the features subsumed by the working view, without testing $x \sqsubset wv$; and similarly to select maximal features without actually checking subsumption,
- the number of concepts can be exponential with the number of objects, whereas the set of features is sufficient for navigation, and its size is linear with the number of objects (see Section 6.1.2),
- the order $\leq^c$ on concepts changes when objects are added, contrary to the order $\sqsubseteq$ on formulas.

The principal operations we consider on this logical cache are:

insertion of a formula: a new feature or object description is inserted in the graph and connected to other formulas according to the subsumption relation;

addition of an object: a new object is placed on the node representing its description, and extents of existing nodes are updated;

search for the maximal features satisfying some property: this is used to search for navigation links and views.

These operations are sufficient to implement the shell commands we presented for navigation and querying.

6.1.2 Theoretical and Practical Complexity

The two important parameters for determining complexities are the number of objects, n, and the number of features per object, f (there are other parameters, but they are bounded by f). The space complexity of the Hasse diagram is in $\mathcal{O}(f^2 n)$. Among the three above operation, only the insertion of a formula actually uses the subsumption test: the number of call to $\sqsubseteq$ is in $\mathcal{O}(fn)$. Other operations find their results in the logical cache. The time complexity of set operations is either in $\mathcal{O}(f^2)$ (insertion of a formula, and addition of an object), or in $\mathcal{O}(fn)$ (search for links and views).

Our experiments have shown that the number of features per object f is nearly constant for a given application (see Section 6.1.3). This is explained by the fact that features are extracted from object descriptions automatically and without considering other objects. The consequence for complexities is that every operation is either constant, or linear with the number of objects. This means that navigation and querying in a LIS are tractable.

6.1.3 Experiments

A prototype of a Logical Information System has been built for experimentation purpose. It has been implemented in λProlog (Miller and Nadathur, 1986; Belleannee, Brisset, and Ridoux, 1999) as a generic system in which a theorem-prover and a syntax analyzer can be plugged-in for every logic used in descriptions. It is not meant to be efficient, though it can handle several thousand entries. Contrary to other tools based on concept analysis, it does not create the concept lattice. It only manages a Hasse diagram of the features used so far.

For the ICCS *Bib* context (see Example 7), the Hasse diagram has 954 nodes and 2150 arcs. In the experiments reported in this article, all response times are shorter than 1 second. In other experiments with a full-sized *Bib* context, i.e., all BibTeX fields (Lamport, 1985) are represented and there are several thousand bibliographical references, the Hasse diagram has an average of 15 nodes per object, 3 arcs per node, and a height of about 5. This experiment and others support the idea that the number of features per object, f, is nearly constant for a given application; e.g., f is about 60 in *Bib* contexts. This has a positive implication on the complexity of LIS operations, because under this hypothesis their time complexity is either constant, or linear with the number of objects (see above).

We present in this section how the generic scheme that takes a logic as a parameter can be instantiated.

6.2.1 Principles

Using logics as schemas in data-bases implies that end-users or system administrators will have to define and implement logics. Clearly, this is out of reach for most of them. In order to make our principles practicable anyway, we have designed a framework for specifying and implementing logics. This framework is based on what we called *logic functors*. Using them, defining and implementing a logic consists merely in combining parameterized logics. Each logic functor consists in a logic component, e.g., propositional logic or interval comparison. Functors can be composed to form new logics, e.g., propositional logic on intervals.

Each functor is implemented as a parameterized theorem prover. Our goal is that the theory and theorem prover of a combination of functors result from a systematic combination of the theory and theorem prover of each functor.

All functors and their compositions implement a common interface which corresponds to the 6-tuple of Definition 1. This makes it possible to program generic applications that can be instantiated with a logic component. Conversely, customized logics built using the logic functors can be *embedded* in an application that respects this interface.

The whole framework development is geared towards manipulating logics as lattices. So, subsumption is considered as a relation between formulas, and we study the conditions under which this relation is a partial order.

Our idea is to consider that a logic interprets its formulas as functions of their atoms. By abstracting atomic formulas from the language of a logic we obtain what we call a *logic functor*. A logic functor can be applied to a logic to form a new logic. For instance, if propositional logic is abstracted over its atomic formulas, we obtain a logic functor called *prop*, which we can apply to, say, a logic on intervals *interv*, to form propositional logic on intervals, *prop*(*interv*).

6.2.2 Logics and logic functors

We present the logic functor structure. More details can be found in (Ferré and Ridoux, 2002a), particularly on the conditions for composability.

Definition 43 (logic) *A logic L is a triple (AS_L, S_L, P_L) where AS_L defines the abstract syntax of formulas of L, S_L defines their semantics, and P_L defines their interface.*

S_L is a pair $(I_L, \models_L)$ where

- *I_L is the interpretation domain of the formulas of L, and*
- *$\models_L \in \mathcal{P}(I_L \times AS_L)$ is the satisfaction relation between interpretations and formulas; $i \models_L f$ means that i is a model of f. We write $M_L(f) = \{i \in I_L \mid i \models_L f\}$ the set of all models of a formula f.*

P_L is a 5-uple $(\sqsubseteq_L, \sqcup_L, \sqcap_L, \top_L, \bot_L)$ where

- *$\sqsubseteq_L \in \mathcal{P}(AS_L \times AS_L)$ is the subsumption relation,*
- *$\sqcup_L, \sqcap_L \in AS_L \times AS_L \to (AS_L \cup \{undef\})$ are conjunction and disjunction, and*
- *$\top_L, \bot_L \in AS_L \cup \{undef\}$ are the tautology and contradiction for L.*

All logics built with logic functors present the same interface ($\sqsubseteq_L$,$\sqcup_L$,$\sqcap_L$,$\top_L$,$\bot_L$), plus other operations like updating (operation $\diamond$ in Section 4.1), parsing and printing. So, they implement the same abstract data-type. The 5 logic operations considered here correspond to a minimal requirement about logics; the ability to test subsumption, to build new formulas using conjunction and disjunction, and to determine if a formula is a tautology or a contradiction. A logic may have more connectives, but they will appear in its abstract syntax, not in its interface.

In order to simplify the presentation, we will only consider operations $\sqsubseteq_L$, $\sqcap_L$, and $\top_L$ in the sequel.

The definition of $\sqcap_L$ needs not be total. Similarly, $\top_L$ needs not be defined. Moreover, there is no *a priori* relation between S_L and P_L. So, we need a notion of completeness and consistency that takes into account partial definitions.

Definition 44 (completeness) *Let L be a logic, the operations of its interface P_L, i.e., $\sqsubseteq_L$, $\sqcap_L$, and $\top_L$, are complete w.r.t. a semantics S_L, if and only if for all formulas $f, g \in AS_L$ we have*

- *$M_L(f) \subseteq M_L(g) \Longrightarrow f \sqsubseteq_L g$,*
- *$\top_L \neq undef \Longrightarrow M_L(\top_L) = I_L$,*
- *$f \sqcap_L g \neq undef \Longrightarrow M_L(f \sqcap_L g) \supseteq M_L(f) \cap M_L(g))$,*

An interface P_L is complete w.r.t. a semantics S_L if and only if each of its operation is complete.

Definition 45 (consistency) *Let L be a logic, the operations of its interface P_L, i.e., $\sqsubseteq_L$, $\sqcap_L$, and $\top_L$, are consistent w.r.t. a semantics S_L, if and only if for all formulas $f, g \in AS_L$ we have*

- $f \sqsubseteq_L g \Longrightarrow M_L(f) \subseteq M_L(g)$,
- $\top_L$ *is always consistent,*
- $f \sqcap_L g \neq undef \Longrightarrow M_L(f \sqcap_L g) \subseteq M_L(f) \cap M_L(g)$,

An interface P_L is consistent w.r.t. a semantics S_L if and only if each of its operation is consistent.

By this definition, operations $\sqcap_L$ and $\top_L$ (and also $\sqcup_L$ and $\bot_L$) of a completely *undef*ined interface are trivially complete and consistent, but they makes a useless interface. So, the game of designing a new logic is to make it defined enough to be useful, but still complete and consistent.

Definition 46 (logic functor) *Assuming $\mathbb{L}$, $\mathbb{AS}$, $\mathbb{S}$, and $\mathbb{P}$ the collections of all logics, abstract syntax, semantics, and interface, a logic functor $F : \mathbb{L}^n \to \mathbb{L}$ is a triple (AS_F, S_F, P_F) defined as follows:*

- $AS_F : \mathbb{AS}^n \to \mathbb{AS}$ *such that* $AS_{F(L_1,\ldots,L_n)} = AS_F(AS_{L_1}, \ldots, AS_{L_n})$,
- $S_F : \mathbb{S}^n \to \mathbb{S}$ *such that* $S_{F(L_1,\ldots,L_n)} = S_F(S_{L_1}, \ldots, S_{L_n})$,
- $P_F : \mathbb{P}^n \to \mathbb{P}$ *such that* $P_{F(L_1,\ldots,L_n)} = P_F(P_{L_1}, \ldots, P_{L_n})$,

By convention, a logic will be considered as a logic functor of type $\mathbb{L}$.

Logic functors are used as follows. S_L describes the semantics; it acts as a specification. P_L implements an interface; its description must be constructive enough, so that it leads directly to a program. A part of the interface describes how the concrete syntax is parsed and printed (remember that AS_L is only the abstract syntax). The other part offers logic operations. The user composes a logic by applying logic functors to logics, say $F(L_1, \ldots, L_n)$, and a *logic composer* takes such an expression and produces automatically the concrete implementation of the logic by gluing together the concrete implementations of F, L_1, ..., and L_n. This results in a software component, with a formally specified interface, that can be plugged in any software system that assumes the same interface.

This methodology leads to designing a library of logic functors for describing objects of LIS: a unary propositional functor (*prop*), whose main quality is to make a total logic out of a partial one ($prop(partial) = complete$), several nullary logic functors for concrete domains like strings and intervals, several n-ary logic functors for combining descriptions, and an auto-epistemic logic

functor for representing (un)completeness of knowledge.

It is worth insisting on the auto-epistemic logic functor. Very often, users expect that a description `author is "Smith"` implies ¬ `author is "Jones"`. However, this is false in standard propositional logic. One needs a form of Closed World Assumption or a kind of Negation as Failure to prove that. However, to be used in a LIS a logic must be monotonic (because its subsumption relation must form a lattice). $\mathcal{ONL}$ (a.k.a. "All I Know" Levesque, 1990) is an epistemic modal logic that permits to express that a formula tells the whole truth. It is a variant of $\mathcal{ONL}$ that we are using in LIS (Ferré, 2001); and its functor is called *aik*. Since it is very frequent to have to express absolute knowledge, the idiom $prop(aik(prop(\ldots)))$ has become a mandatory prefix of the logics used in LIS.

The principle of composing logic functors has been implemented in a prototype. It reads logic specifications such as $prod(prop(atom), prop(interv))$ and produces automatically a printer, a parser, and a theorem-prover. This example means that logic formulas are products of a proposition on atoms and a proposition on intervals. The prototype reads such a constructed logic and builds a theorem-prover for it by instantiating the theorem-prover associated to each logic functor at every occurrence where it is used. The prototype, each functor implementation, and the resulting implementations are written in λProlog.

Coming back to the bibliography example of the introduction, we construct a dedicated logic with logic functors as follows:

$$prop(aik(prop(sum(atom, valattr(sum_n(interv, string, \ldots)))))).$$

which means propositional logic on epistemic modal formulas on propositional logic on a sum (i.e., a mix) of atomic formulas and valued attributes whose values are themselves made of interval formulas, string formulas, etc. According to the theory of logic functors, the following theorem holds.

Theorem 47 (*Bib* logic is ok) *The logic L used in Example 7,*

$L = prop(aik(prop(sum(atom, valattr(sum_2(interv, string))))))$,

is such that P_L is total, bounded, and consistent and complete in all five operations w.r.t. S_L.

7 Conclusion

7.1 Summary of LIS

We have presented the specifications of a Logical Information System based on (Logical) Concept Analysis. It is based on Concept Analysis and is generic w.r.t. a logic for describing objects. In this framework, navigation/querying and creation/updating can be seamlessly integrated.

In this way, standard commands of a file system shell can be mimicked in a logical context. However, a simple generalization of the definition of links forms a framework in which operations of data-analysis or data-mining can also be expressed. Using this framework, purely symbolic navigation as well as statistical exploration can be integrated smoothly as variants of the same generic operation.

As opposed to previous attempts of using Concept Analysis for organizing data, we do not propose to navigate directly in the concept lattice. Instead, we use the contextualized logic (i.e., the logical view of the concept lattice) to evaluate the relevance of navigation links. Those that do not narrow the focus of the search are called *views*. They only restrict the language of available navigation links. Other links, that do narrow the focus of the search, can be used to come closer to some place of interest. The definition of links can be generalized to encompass data-mining notions like necessary and sufficient conditions, and association rules.

The advantage of LIS is a great flexibility which comes from two factors:

(1) the integration of operations that were exclusive in most systems,
(2) the use of logic with Concept Analysis, which solves the name problem.

We have experimented it in various contexts: e.g., cook-books, bibliographical repository, software repository (search by keywords, and search by types), and simply a note-pad. Various logic components were used in these contexts: atoms, intervals, strings, propositional logic, type entailment, taxonomies (e.g., for ingredients). In all cases, a LIS goes beyond any a priori structure and permits many kinds of views on the same information. For instance, in the case of a cook-books, if every recipe is described by its ingredients, its process, the required kitchen utensils, its dietetic value, its place in a meal, and more cultural information, then a cook, a dietician, and a gourmet can have very different views on the same data, and acquire new information by data-mining and learning, simply by using a few LIS shell commands. Similarly, if software components have intrinsic descriptions like their types and languages, the modules they use, parts of specification, and requirements, and

extrinsic descriptions like their testing status, and who is using them, then several software engineering operations like developing and testing, versioning and configuring, and maintenance and evolution can be done using the same repository under different views, and also going smoothly from one view to another.

A thorough description of LIS can be found in the PhD thesis of the first author (Ferré, 2002b). It contains all proofs, gives more details about implementation, and in addition to this paper, considers the representation of complete vs. incomplete object descriptions.

The exclusive use of logic for describing objects, and for designing operations asks the question of whether *uncertainty* can be dealt with in LIS. In fact, LIS can handle uncertainty at two levels.

First level is the logical language. Because LIS is generic w.r.t. logic, it can accomodate uncertainty either in descriptions of objects or in queries by using logics that handle uncertainty. Even propositional logic can handle a kind of uncertainty with disjunctions, but other logics like interval logic or logic "All I Know" can handle other kinds of uncertainty. Intervals can deal with uncertainty of numerical values, and "All I Know" deals with the extent of knowledge. Both have been implemented as logic functors (see Section 6.2).

LIS is not opposed to using *fuzzy* modeling languages (logic, sets, etc), but it requires an entailment relation that forms a lattice. In particular, the entailment relation must be monotonic. Note that "All I Know" is a monotonic logic that handles a notoriously non-monotonic feature: the "Closed World Assumption". We take this as a hint that the restriction to monotonic logics is not a restriction in expressiveness.

Second level is the data-analysis level. As we have shown in Section 5.1, it is very natural to generalize navigation into data-analysis w.r.t. to indicators like confidence and support. This makes it possible to extract knowledge that is not necessarily 100% true.

7.2 Related Works

There have been several other proposal of navigation/querying based on Concept Analysis. Lindig (1995) designed a concept-based component retrieval based on sets of *significant keywords* which are equivalent to our links for the logic of attributes underlying FCA. Godin et al. (1993) propose a direct navigation in the lattice of concepts, which is in fact very similar to Lindig's approach except that only greatest significant keywords, according to the contextualized subsumption on attributes, are displayed to the user. They have

also notions common to our LIS such as working query, direct query specification, and history of selected queries.

Cole and Stumme (2000) developed a Conceptual E-mail Manager (CEM) where the navigation is based on Conceptual Scales (Prediger, 1997; Prediger and Stumme, 1999). These scales are similar to our views in the sense that they select some attributes acting as links and displayed, as for us, with the size of the concept they select. A difference with our LIS is that these links are ordered according to concept lattices of scales, but it can also be done in LIS by a post-treatment on answers of command `ls`.

However, the main difference with all of these approaches is that we use an (almost) arbitrary logic to express properties. This enables us to have automatic subsumption relations (e.g., (`author is "Wille, Mineau"`) $\sqsubseteq$ (`author contains "Wille"`) $\sqsubseteq$ (`author`)), and thus some implicit views (e.g., `author`, `year`). Another difference is that we propose to handle in a uniform way, based on CA, navigation and querying as above, but also, updating, data-mining, learning, etc.

van Rijsbergen (1986) also combined information retrieval (IR) and logic by introducing a logical model of IR, in which both object descriptions and queries are logical. As for us, many logics could be used in this model, but a first difference is that the relevance relation is defined as a measure combining the exhaustivity and specificity of the object description to the query, rather than an exact one (the subsumption $\sqsubseteq$). This measure is used to rank the answers of a query. A second difference is that navigation is not here combined with querying. Later, Chiaramella (1997) added some navigation to this logical model of IR. However, this navigation is structural rather than conceptual as in a LIS, which means that a navigation step leads from a set of objects to another, rather than from a query to a set of sub-queries. The fact that our answers are intentional rather than extensional justifies we can rely on an exact relevance relation because answers to queries are not flat sets of objects. However, it is also possible to take into account uncertainties in a LIS (see the end of Section 7.1).

7.3 Future Work

Our most practical perspective is to design a *logical file system*, which would implement the ideas we have presented in this article, and serve as a Logic Concept Repository for a LIS. The expected advantage is to offer the services described here at a standard system level that is accessible for every application. So doing, even applications that do not know about logical information systems (like e.g., all existing compilers) would benefit from it. For this logi-

cal file system, it will also be important to make our ideas work with a large volume of data in an efficient way.

A graphical user-interface to logical file systems would allow to display in an integrated fashion the working query, the working view, and the corresponding extent and set of links. For instance, a graphical interface for keeping trace of navigation, like what is becoming standard for file browsers, has been already experimented for a simple logic (attributes with values) but should be developed further. This amounts to keep a trace of the path from the start of the navigation to the current place. Moreover, the set of links could be presented graphically as a diagram of ordered formulas. A further refinement is to take into account the contextualized subsumption, to get something similar to concept lattices derived from scales (Cole and Stumme, 2000). This amounts to represent an overview of possible future navigations.

The *World Wide Web* can also be explored using our techniques if one considers answers to web-queries as a formal context into which to navigate. More ambitious is to think of a Web-based LIS. In this case, the main issues will be distribution of data and computation.

An application of the learning schema exposed in Section 5.2.1 is to use the learning schema for navigating. A description would play the role of a query, and its associative concepts could be proposed to the user as alternative queries. The advantage is that though the initial query could have an empty answer, the alternative ones always correspond to non-empty concepts. So, it makes it possible to start a search with only an example of what the user is looking for, and then see actual representatives of the queried concepts.

Another interesting perspective follows the observation that the notion of *associative concepts* is closely related to *modified* and *new concepts* in the incremental concept formation (Godin, Missaoui, and Alaoui, 1995). We developed further this correspondence, which led to an improved algorithm for incrementally computing concepts of sparse contexts (Ferré, 2002a).

In all this article, names or descriptions are essentially unary predicates, whichever is the actual logic used for this purpose. However, several applications require to express relations between objects, i.e., n-ary predicates. For instance, a LIS for a software environment should permit to express such relations as `calls` f or `is connected to` x, where f and x are objects. These relations form concrete links between objects, which we plan to consider for navigation in a future work. The main difficulty is to manage the concrete links in a way that remains compatible with the other navigation links. This will also permit to represent topological informations, e.g., `West of` x or 10 `miles from` y, that are used in Geographical Information Systems.

References

Barbut, M., Monjardet, B., 1970. Ordre et classification — Algèbre et combinatoire (2 tomes). Hachette, Paris.

Belleannee, C., Brisset, P., Ridoux, O., 1999. A pragmatic reconstruction of λprolog. The Journal of Logic Programming 41, 67–102.

Brachman, R. J., 1979. On the epistemological status of semantic nets. In: Findler, N. V. (Ed.), Associative Networks: Representation of Knowledge and Use of Knowledge by Examples. Academic Press, New York.

Chaudron, L., Maille, N., 1998. 1st order logic formal concept analysis: from logic programming to theory. Computer and Information Science 13 (3).

Chiaramella, Y., 1997. Browsing and querying: two complementary approaches for multimedia information retrieval. In: N. Fuhr, G. Dittrich, K. T. (Ed.), Hypermedia - Information Retrieval - Multimedia. Universitätsverlag Konstanz, pp. 9–26.

Cole, R., Stumme, G., 2000. CEM - a conceptual email manager. In: Mineau, G., Ganter, B. (Eds.), Int. Conf. Conceptual Structures. LNCS 1867. Springer, pp. 438–452.

Crestani, F., Lalmas, M., 2001. Logic and uncertainty in information retrieval. In: Lectures in Information Retrieval. LNCS 1980. pp. 179–207.

Davey, B. A., Priestley, H. A., 1990. Introduction to Lattices and Order. Cambridge University Press.

Di Cosmo, R., 1995. Isomorphisms of Types: from λ-calculus to information retrieval and language design. Progress in theoretical computer science. Birkhäuser.

Ferré, S., 2001. Complete and incomplete knowledge in logical information systems. In: Benferhat, S., Besnard, P. (Eds.), Symbolic and Quantitative Approaches to Reasoning with Uncertainty. LNCS 2143. Springer, pp. 782–791.

Ferré, S., Oct. 2002a. Incremental concept formation made more efficient by the use of associative concepts. Research Report RR-4569, Inria, Institut National de Recherche en Informatique et en Automatique.
URL `http://www.inria.fr/RRRT/RR-4569.html`

Ferré, S., Oct. 2002b. Systèmes d'information logiques : un paradigme logico-contextuel pour interroger, naviguer et apprendre. Thèse d'université, Université de Rennes 1, accessible en ligne à l'adresse http://www.irisa.fr/bibli/publi/theses/theses02.html.
URL `http://www.irisa.fr/bibli/publi/theses/theses02.html`

Ferré, S., Ridoux, O., Dec. 1999. Une généralisation logique de l'analyse de concepts formels. Technical Report RR-3820, Inria, Institut National de Recherche en Informatique et en Automatique, an english version is available at http://www.irisa.fr/lande/ferre.

Ferré, S., Ridoux, O., 2000. A logical generalization of formal concept analysis. In: Mineau, G., Ganter, B. (Eds.), Int. Conf. Conceptual Structures. LNCS 1867. Springer, pp. 371–384.

Ferré, S., Ridoux, O., 2002a. A framework for developing embeddable customized logics. In: Pettorossi, A. (Ed.), Int. Work. Logic-based Program Synthesis and Transformation. LNCS 2372. Springer, pp. 191–215.

Ferré, S., Ridoux, O., 2002b. The use of associative concepts in the incremental building of a logical context. In: U. Priss, D. Corbett, G. A. (Ed.), Int. Conf. Conceptual Structures. LNCS 2393. Springer, pp. 299–313.

Ganter, B., Kuznetsov, S., 2001. Pattern structures and their projections. In: Delugach, H. S., Stumme, G. (Eds.), Int. Conf. Conceptual Structures. LNCS 2120. Springer, pp. 129–142.

Ganter, B., Wille, R., 1999. Formal Concept Analysis — Mathematical Foundations. Springer.

Gifford, D. K., Jouvelot, P., Sheldon, M. A., O'Toole, J. W. J., 1991. Semantic file systems. In: 13th ACM Symposium on Operating Systems Principles. ACM SIGOPS, pp. 16–25.

Godin, R., Missaoui, R., Alaoui, H., 1995. Incremental concept formation algorithms based on Galois (concept) lattices. Computational Intelligence 11 (2), 246–267.

Godin, R., Missaoui, R., April, A., 1993. Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. International Journal of Man-Machine Studies 38 (5), 747–767.

Gopal, B., Manber, U., 1999. Integrating content-based access mechanisms with hierarchical file systems. In: third symposium on Operating Systems Design and Implementation. USENIX Association, pp. 265–278.

Herzig, A., Rifi, O., 1999. Propositional belief update and minimal change. Artificial Intelligence 115 (1), 107–138.

Keller, A. M., Mar. 1985. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In: 4th ACM Symp. Principles of Database Systems. pp. 154–163.

Krone, M., Snelting, G., May 1994. On the inference of configuration structures from source code. In: Int. Conf. Software Engineering. IEEE Computer Society Press, pp. 49–58.

Kuznetsov, S., 1999. Learning of simple conceptual graphs from positive and negative examples. In: Żytkow, J. M., Rauch, J. (Eds.), Principles of Data Mining and Knowledge Discovery. LNAI 1704. Springer, pp. 384–391.

Kuznetsov, S., Objedkov, S., 2001. Comparing performance of algorithms for generating concept lattice. In: et al., E. M. N. (Ed.), ICCS-2001 Int. Workshop on Concept Lattices-based Theory, Methods and Tools for Knowledge Discovery in Databases. Stanford University, CRIL – IUT de Lens, France.

Lamport, L., 1985. LaTeX— A Document Preparation System. Addison-Wesley, 2nd edition.

Levesque, H., Mar. 1990. All I know: a study in autoepistemic logic. Artificial Intelligence 42 (2).

Lindig, C., 1995. Concept-based component retrieval. In: IJCAI95 Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs.

Meghini, M., Sebastiani, F., Straccia, U., Thanos, C., 1993. A Model of infor-

mation Retrieval based on Terminological Logic. In: 16th Annual Int. ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 298–307.

Miller, D. A., Nadathur, G., 1986. Higher-order logic programming. In: Shapiro, E. (Ed.), In Third Int. Conf. Logic Programming. LNCS. Springer-Verlag, London, pp. 448–462.

Napoli, A., Dec. 1997. Une introduction aux logiques de descriptions. Rapport de recherche RR-3314, Inria, Institut National de Recherche en Informatique et en Automatique.

Prediger, S., 1997. Logical scaling in formal concept analysis. LNCS 1257 , 332–341.

Prediger, S., Stumme, G., 1999. Theory-driven logical scaling. In: International Workshop on Description Logics. Vol. 22. Sweden.

Snelting, G., Jul. 1998. Concept analysis — A new framework for program understanding. ACM SIGPLAN Notices 33 (7), 1–10.

Ullman, J. D., 1989. Principles of Database and Knowledge-Base Bystems. Computer Science Press, Rockville, Maryland.

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

Vogt, F., Wille, R., 1994. TOSCANA — a graphical tool for analyzing and exploring data. In: Symposium on Graph Drawing. LNCS 894. pp. 226–233.

Wille, R., 1982. Ordered Sets. Reidel, Dordrecht Boston, Ch. Restructuring lattice theory: an approach based on hierarchies of concepts, pp. 445–470.

Appendix B

Camelis: a Logical Information System to Organize and Browse a Collection of Documents (2009)

This journal article [**Ferré, 2009a**] has been published in a special edition of the *International Journal of General Systems* in 2009. It contains a mature and comprehensive description of the first generation of logical information systems, and its main implementation Camelis. It contains only the necessary theoretical elements, and emphasizes the practical aspects and the user point-of-view. It illustrates the LIS principles on a real personal photo collection. It also develops a richer user experience with various querying and navigation modes: e.g., querying by formula, querying by example, navigating downward/upward/sideward. It also provides algorithmic results with a data structure, the *logic cache*, basic operations and their complexity. A modular architecture facilitates the application to various application domains, using different logics (for describing objects) and different *transducers* (for importing external data).

International Journal of General Systems
Vol. 00, No. 00, Month 200x, 1–25

RESEARCH ARTICLE

CAMELIS: a logical information system to organize and browse a collection of documents

Sébastien Ferré*

IRISA, Université de Rennes 1, 35042 Rennes cedex, France
(*Received 00 Month 200x; final version received 00 Month 200x*)

Since the arrival of digital cameras, many people are faced to the challenge of organizing and browsing the overwhelming flood of photos their life produces. The same is true for all sorts of documents, e.g. emails, audio files. Existing systems either let users fill query boxes without any assistance, or drive them through rigid navigation structures (e.g., hierarchies); or they do not let users put annotations on their documents, even when this would support the organization and retrieval of any documents on customized criteria. We present a tool, CAMELIS, that offers users with an organization that is dynamically computed from documents and their annotations. CAMELIS is designed along the lines of Logical Information Systems (LIS), which are founded on logical concept analysis. Hence, (1) an expressive language can be used to describe photos and query the collection, (2) manual and automatic annotations can be smoothly integrated, and (3) expressive querying and flexible navigation can be mixed in a same search and in any order. This presentation is illustrated on a real collection of more than 5,000 photos.

Keywords: document collection; annotation; browsing; information retrieval; formal concept analysis; logical information systems

1. Introduction

Formal Concept Analysis (FCA) has been recognized as a good paradigm for information retrieval (Godin *et al.* 1993, Carpineto and Romano 1996, Martinez and Loisant 2002, Cole *et al.* 2003) because it makes possible the combination of querying and navigation in a same search. Querying alone is not satisfying because it requires users to know the query language, and to have a precise idea of what they search for. Navigation, by leading users to the result step by step, is more interactive, but the navigation structure is most often very rigid so that only one or a few paths exist to each object (e.g., file hierarchy, hyperlink graph). In FCA, the concept lattice plays the role of the navigation structure. Each concept combines a query as a set of attributes (the intent), and a navigation place as a set of objects (the extent). Attributes can be added to the query in any order, so that a concept can be reached through several paths.

Logical Information Systems (LIS) (Ferré and Ridoux 2004) have been introduced (1) to combine querying and navigation, (2) to make use of an expressive language for object descriptions and queries, (3) to be generic w.r.t. the kind of objects and the language, and (4) to be reasonably efficient on large collections of

*Corresponding author. Email: ferre@irisa.fr

ISSN: 0308-1079 print/ISSN 1563-5104 online

DOI: 10.1080/0308107YYxxxxxxxx
http://www.informaworld.com

documents. Because of (2) it becomes necessary to do complex reasoning to decide whether an object description matches a query (e.g., the object description is a string, and the query is a regular expression). Logics are the right tools to encapsulate representation and reasoning facilities. Because of (3), we cannot fix the logic *a priori*. So we defined a generalization of FCA, Logical Concept Analysis (LCA) (Ferré and Ridoux 2000), where logical formulas of an almost arbitrary logic can be used instead of sets of attributes. This makes FCA an instance of LCA, where object descriptions and queries are sets of attributes. Because of (4) we propose, like other authors (Carpineto and Romano 1996, Ducrou and Eklund 2008), a local view on one concept at a time, instead of a global view on the whole concept lattice. Indeed, the concept lattice has a size that can grow exponentially with the size of the context. However, unlike other approaches, our local view is defined only in terms of the document collection (objects and their properties), and neither refers explicitly to the concept lattice, nor requires its effective computation. We show that such a *concept view* supports an intuitive, flexible, informative and efficient exploration of the concept lattice.

Camelis[1] is a complete implementation of a logical information system. It is generic in that a logic module can be plugged in so as to cover different application needs. It uses specific data structures and algorithms so that it scales up to 100,000 objects. It has a graphical interface that displays at all times the concept view: i.e., the query that led to the current concept, its extent, and an index of properties that provides a summary of the extent, and navigation links to other concepts. Both browsing and updating the context are done through this interface.

Among the various existing applications of Camelis, the most convincing is the management of a photo collection. Indeed, photos can be described along many facets like date, location, event, visible persons, visible objects. A file hierarchy enforces a strict order between these facets, making some searches hardly possible. Tag-based systems like Flickr™ are limited because a photo tagged with 'Sydney' as a location will not be an answer to a query containing 'Australia'; and a photo tagged with "formal concept analysis" will not be an answer to "concept analysis". These limitations are easily solved by dedicated logics: here, a taxonomy of locations, and a logic of string patterns. In this paper, we illustrate the capabilities of Camelis and LIS on a real context, the personal photo collection of the author. The collection contains more than 5,000 photos, and has been incrementally defined since 2003, coinciding with the arrival of new photos.

There already exist other implementations of logical information systems. LisFS (Padioleau and Ridoux 2003) instantiates the Virtual File System (VFS) of Linux to provide LIS navigation at the system level, hence making it accessible to existing applications. Geolis builds on top of LisFS (Bedel *et al.* 2008), and is a Geographical Information System (GIS) with a Web interface. It displays the extent as a map instead of a list, and this map is also the support of graphical navigation links. Odalisque (Allard and Ferré 2008) applies to the Web semantic by enabling the browsing of OWL-DL ontologies.

Section 2 recalls the main definitions of LCA, and introduces the *concept view*, both formally and in the graphical interface. Section 3 assumes an existing context, and presents all the facilities provided by Camelis to browse and retrieve photos, from querying by formula and querying by example, to different kinds of navigation: downward and upward, sideward, pivot, backward and forward. Section 4 illustrates the incremental definition of the context with the arrival of a new pack of photos. Section 5 sketches the different components of Camelis, and describes the main

[1]The version used in this paper is 1.4, down-loadable at `http://www.irisa.fr/LIS/ferre/camelis/`.

data structures and algorithms along with their complexities. Section 6 compares LIS with state-of-the-art approaches.

2. Logical concept analysis

We recall the basics of Logical Concept Analysis (LCA) (Ferré and Ridoux 2000, 2004), whose principles are the same as in FCA, except that *logical properties* partially ordered by a *subsumption* relation are used instead of attributes. This subsumption relation is founded on a *semantics* that can be seen as a formal context.

2.1 Logic

For genericity we do not want to fix the logic *a priori*, both in the theory and in the implementation. So we define what we expect from a logic in the framework of Logical Information Systems (LIS). A classical view is to define a logic as the combination of a *syntax* that defines the language of formulas, and a *semantics* that gives a meaning to those formulas.

Definition 2.1: A *logic* is a triple $\mathcal{L} = (I, L, \models)$, where:

- I is a set of *interpretations*,
- L is a set of well-formed *formulas*,
- and $\models$ is a relation between interpretations and formulas, and means "is a model of".

L makes up the syntax, while I and $\models$ make up the semantics of the logic.

In FCA terms, a logic can be seen as the formal context of the universe, where I is the set of all possible objects, L is the set of attributes, and $\models$ is the incidence relation determining which object has which attribute.

If it happens that every model of some formula f is also a model of another formula g, then we say that f is subsumed by g, and we write $f \sqsubseteq g$.

Definition 2.2: Let $\mathcal{L} = (I, L, \models)$ be a logic, and $f, g \in L$ be two formulas. The *subsumption* $\sqsubseteq$ is a binary relation between formulas, defined by:

$$f \sqsubseteq g \iff \forall i \in I : i \models f \Rightarrow i \models g.$$

This subsumption relation plays a crucial role in LIS because it helps to organize logical properties into taxonomies, and decides whether an object is an answer to some query. Other logical operations could be defined for logics (e.g., least common subsumer, conjunction, disjunction), but they are not necessary for LIS. From the point of view of LIS, logics are black boxes that are invoked through an API that contains a parser (from strings to formulas), a pretty-printer (from formulas to strings), and a subsumption operation.

Our specification of logics are akin to Description Logics (DL) (Donini *et al.* 1997), where formulas are called *concepts*, and interpretations are called *individuals*. A correspondence with first-order logic (FOL) exists, and states that a DL concept is equivalent to a FOL formula with one free variable, and that subsumption is then equivalent to implication. This implies that a concept is not interpreted by a truth-value, but by a set of individuals. In our logics, a formula denotes a set of interpretations, its models.

In the photo application, the formulas used to represent photo properties, and to denote sets of photos, are valued attributes and taxonomic terms. Value domains and taxonomies are defined as logics. In order to fix these ideas, we give the definition of the value domain of dates, and the taxonomy of locations. Other value domains exist for intervals of number (e.g., image size, exposure time), string patterns (e.g., event, comment), time; and other taxonomies exist for representing types of photos, visible persons, and visible objects.

Example 2.3 The logic of dates $\mathcal{L}_{date} = (I_{date}, L_{date}, \models_{date})$ is defined as follows:

- I_{date} is the set of all possible dates: e.g., 21/11/2007;
- L_{date} is the set of all possible dates at different resolution (year, month, day): e.g., `2007`, `nov 2007`, `21 nov 2007`;
- $\models_{date}$ is the obvious matching relation between interpretations and formulas: e.g., 21/11/2007 $\models$ `nov 2007`, 21/11/2007 $\not\models$ `nov 2006`;
- $\sqsubseteq_{date}$ is a similar matching relation extended to non-exact dates: e.g., `21 nov 2007` $\sqsubseteq$ `nov 2007` $\sqsubseteq$ `2007`.

It is easy to prove that this definition of subsumption is correct w.r.t. the semantics.

Example 2.4 Let $(Loc, \leq)$ be a partial ordering representing the taxonomy of locations. The logic of locations $\mathcal{L}_{loc} = (I_{loc}, L_{loc}, \models_{loc})$ is defined as follows:

- $I_{loc} = Loc$ is the set of all possible locations (seen as atomic locations);
- $L_{loc} = Loc$ is the set of all possible locations (seen as sets of locations);
- $\models$ is equal to the partial ordering on locations: e.g., *Paris* $\models$ `France` ("Paris is in France");
- $\sqsubseteq$ is also equal to this partial ordering: e.g., `Paris` $\sqsubseteq$ `France` ("Paris is included in France").

Note that the same location is distinguished as an atomic location (interpretation), and as a set of locations (formula). This models the fact that the same location (e.g., Paris) can be represented on a map as a point or as a region, depending on the scale of the map.

Using one of the above logics would allow us to describe photos with only one property, and querying them only with a pattern. We define the *union logic* of a set of elementary logics so as to allow for several properties on objects, and Boolean connectors in queries (*and, or, not*). We here define only the syntax and subsumption, as the presentation of the semantics would take us out of the scope of this paper (details are available in a research report (Ferré and Ridoux 2006)).

Example 2.5 Let $\{\mathcal{L}_k\}_k$ be a finite set of elementary logics $\mathcal{L}_k = (I_k, L_k, \models_k)$, which are pair-wise disjoint (i.e., every interpretation and formula belongs to a unique elementary logic). The *union logic* of the logics $\{\mathcal{L}_k\}_k$ is a logic $\mathcal{L} = (I, L, \models)$, where:

- $L = L_d \cup L_q$: the language is made of descriptions (L_d), and queries (L_q); descriptions are finite sets of elementary formulas, and queries are Boolean combinations of elementary formulas or the most general formula (*all*);
- and for every $d \in L_d$, $q_1, q_2 \in L_q$, $x_k, y_k \in L_k$, the subsumption is defined by

Figure 1. Property trees from logics of date, string, and location, as displayed in the graphical interface of Camelis.

the following inference rules:

$$
\begin{array}{lcl}
d \sqsubseteq y_k & \iff & \exists x_k \in d : x_k \sqsubseteq_k y_k \\
d \sqsubseteq \mathit{not}\ q_1 & \iff & d \not\sqsubseteq q_1 \\
d \sqsubseteq q_1\ \mathit{and}\ q_2 & \iff & d \sqsubseteq q_1 \wedge d \sqsubseteq q_2 \\
d \sqsubseteq q_1\ \mathit{or}\ q_2 & \iff & d \sqsubseteq q_1 \vee d \sqsubseteq q_2 \\
d \sqsubseteq \mathit{all} & \iff & \mathit{true} \\
x_k \sqsubseteq y_k & \iff & x_k \sqsubseteq_k y_k
\end{array}
$$

This definition of subsumption applies the closed world assumption as a description d is said to be subsumed by $\mathit{not}\ q_1$ iff it cannot be proved that d is subsumed by q_1. This could be extended to compare two Boolean queries, but such comparisons are never used in Camelis.

A logic dedicated to the photo application is defined as the union of various taxonomies (e.g., locations, types, persons), and valued attributes over various domains (e.g., strings, dates, numbers). This logic has to be plugged in Camelis, because the system has no fixed logic *a priori*. Figure 1 shows a screenshot of Camelis showing trees of formulas derived from this logic. Each node is labelled by a formula, and the child-parent relation corresponds to subsumption, like in classical representations of taxonomies. Now, if taxonomies are embedded in logics, and logics are presented as taxonomies, we may think that the use of logic is overkill. In fact, the justification for logic is that most value domains are infinite, and cannot be defined as a taxonomy, i.e. by enumerating formulas and subsumption links. Moreover, in complicated logics (e.g., description logics), the semantics plays a crucial role in ascertaining the correctness of the subsumption definition. In summary, logic can be seen as a methodology to design infinite taxonomies in a concise way.

2.2 Logical context

While a logic describes the universe, a logical context describes a finite set of objects. In the following examples, we assume that the logic $\mathcal{L}$ is the union logic of some elementary logics. So, each object description is assumed to be a set of elementary formulas, and Boolean connectors are allowed in queries.

Definition 2.6: A *logical context* is a tuple $K = (\mathcal{O}, \mathcal{L}, d)$, where $\mathcal{O}$ is a finite set of object identifiers, $\mathcal{L}$ is a *logic*, and $d \in \mathcal{O} \to L$ is a mapping from objects to their logical description.

In the photo application, object identifiers are file paths or URLs to photos, and their description is a set of logical formulas such as location, date, event. An example of description, for some object o, is $d(o) = \{$`Montpellier, date = 24 oct 2007, event is "CLA 2007"`$\}$. An object can match a formula, even if it does not belong to its description. For instance, the subsumption relation $d(o) \sqsubseteq$ `date = 2007` means that every model of the object description is a model of `date = 2007`. So, `date = 2007` is a *property* of the object o, and reciprocally, o is an *instance* of `date = 2007`.

Definition 2.7: Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a logical context, $o \in \mathcal{O}$ be an object, and $f \in \mathcal{L}$ be a formula. The formula f is a *property* of o, or equivalently the object o is an *instance* of f, iff $d(o) \sqsubseteq f$, which is denoted by $o : f$.

This implies that an object can have an infinite set of properties, while all descriptions are finite. Hence, object descriptions can be kept small and precise at the same time. For instance, an object described with location `Montpellier` will automatically be an instance of any location that contains it, i.e., `France` and `Europe`. Here lies the main benefit of logics w.r.t. attributes. A similar benefit can be obtained with conceptual scaling (Ganter and Wille 1999), provided a set of attributes $\mathcal{A} \subseteq L$ is selected. A formal context can be derived from a logical context K and a set of attributes $\mathcal{A}$, as $K(\mathcal{A}) = (\mathcal{O}, \mathcal{A}, :)$, where the incidence relation is the "instance of" relation. However, this projection entails a loss of information: (1) the subsumption relations between formulas, (2) which properties belong to the description of objects, and which properties are inferred, and (3) the ability to extend the set of attributes or to use arbitrary formulas in queries (without referring to the logical context).

2.3 Concept view

Given a logical context K, LCA defines the extent of a formula, the intent of a set of objects, shows that this forms a Galois connection, and hence that a concept lattice is defined (Ferré and Ridoux 2004). This concept lattice plays the role of a navigation structure in LIS as it does in other concept-based information systems (Godin *et al.* 1993, Carpineto and Romano 1996, Martinez and Loisant 2002, Cole *et al.* 2003). However, the concept lattice derived from a logical context is much too large and dense to be displayed to users. Even when considering the formal context $K(\mathcal{A})$ restricted to a finite set of formulas, the concept lattice is still too large. We emphasize that the main problem is not computing concept lattices (there exist efficient algorithms to this purpose (Kuznetsov and Objedkov 2001)), but making them readable to users: a lattice with *only* 100 concepts is already a challenge to interpret, even for an expert. There exist solutions in FCA to reduce the size of concept lattices, by removing concepts (e.g., iceberg concept lattices (Stumme *et al.* 2002)), or merging nodes using similarity measures. However, reducing the concept lattice implies reducing the navigation space, whereas

we aim to make navigation the most flexible and expressive, which implies (in our opinion) a large navigation space.

Our claim is that showing a rich view on a single concept at a time is enough in most cases, and does not require to effectively compute the concept lattice. Every formal concept (i.e., conjunctive query) can be reached just by following navigation links, and this is also true for many logical concepts (use of disjunction and negation). All other concepts can be made reachable by extending the sets of navigation links, or by entering an arbitrary formula as a query. Section 3 presents all the means that are available to browse the concept lattice downward, upward, sideward, and to reach a concept from a query or from a set of examples.

The *concept view* is made of three parts: (1) the query, (2) the answers of that query as a set of objects, and (3) a set of properties occurring in those answers. The query is a logical formula, e.g., a Boolean combination of elementary properties such as taxonomic terms or valued attributes (see Definition 2.5):

```
date in [aug 2007,may 2008] and (France or Germany) and not event
contains "secret",
```

which denotes photos taken between August 2007 and May 2008, in France or Germany, and not related to a secret event.

The set of answers of a query is defined as the *extent* of that query, i.e., the set of instances of that query.

Definition 2.8: Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a logical context, and $q \in \mathcal{L}$ be a query formula. The *extent* of the query q is defined as the set of instances of q:

$$extent(q) = \{o \in \mathcal{O} \mid o : q\}.$$

Different queries may have the same answers, and hence the same extent. In such a case, we consider that those queries are different ways to identify and reach the same navigation place. Every pair $(extent(q), q)$ is an *inf-semi-concept* of the logical context, i.e., a pair (O, f) where O is the set of all instances of the formula f, but f is not necessarily the most specific formula that is a property of every object in O (Ganter and Wille 1999). This is one of many views of the concept $(extent(q), i)$, where i is the *intent* of the concept, i.e., the most specific formula that is a property of every instance of q.

The intent of the current concept is not computed as, in the frequent case where the union logic is used, it would be trivially the disjunction of the descriptions of the objects in the extension. An approximate intent can be computed given a finite vocabulary X of formulas.

Definition 2.9: Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a logical context, X be a vocabulary over $\mathcal{L}$, and $q \in \mathcal{L}$ be a query. The *intent* of query q is the set of vocabulary formulas shared by all answers of q:

$$intent(q) = \{x \in X \mid extent(q) \subseteq extent(x)\}.$$

This intent provides a description of the current concept, but provides no navigation links to other concepts. To this purpose, we propose a richer description of the current concept, that includes the above intent, and provides many navigation links to move downward, upward and sideward in the concept lattice (see Section 3.1). This richer description relies on a vocabulary X to show a kind of *index* over the current answers. Every formula in the vocabulary that is a property of some answer is displayed along with its count, i.e. the number of answers that are instances of that formula.

Definition 2.10: Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a logical context, X be a vocabulary over $\mathcal{L}$, and $q \in \mathcal{L}$ be a query. The *index* of query q is the set of vocabulary formulas occurring in the extent of q:

$$index(q) = \{(n, x) \mid x \in X, n = |extent(q) \cap extent(x)|, n > 0\}.$$

Note that the finite set X does not affect the logical context, and does not replace the use of logics. It plays the role of a vocabulary for summarizing the instances of the current concept, and representing navigation links to other concepts. Changing the vocabulary only affects the concept view, not the logical context, nor the concept lattice. Section 2.4 explains how a relevant *navigation vocabulary* can be automatically extracted from the logical context.

A formula is not shown in the index if its count is 0, because it tells us nothing about the current extent. In order to get fewer formulas in the index it is possible to set a minimum count (the default is 1). If this minimum count is set to the size of the extent, the set of formulas in the index is equal to the intent. An index formula f that is not in the intent splits the extent in two non-empty subsets, and thus determines two more specific inf-semi-concepts: $(extent(q) \cap extent(f), q \ and \ f)$ and $(extent(q) \setminus extent(f), q \ and \ not \ f)$.

The index of a query provides sets of properties of the query answers, such as countries, date intervals, or events. Because the count of formulas depends on the context and the current extent, indexes need to be recomputed each time the context is updated or the query changes. This makes the indexes dynamic and informative. Specific data structures are used to make their computation efficient, i.e. linear in the size of the context (see Section 5).

Figure 2 shows the graphical interface of Camelis, which reflects the concept view presented above. At every time, the *query* is displayed as a text field at the top, the *extent* is displayed as a preview list at the right, and the *index* is displayed as a set of property trees at the left. The query is editable, and the extent can be scrolled page by page (the page size can be customized). The trees reflect the subsumption ordering on index properties, and help to structure and control the display of the index. As a logic can be any partial ordering, not necessarily a tree, the same property can appear several times under different nodes. This is not a problem as tree nodes are expanded on demand. Initially, all nodes are collapsed. When expanding a node, options are available to change the minimum count or the sorting of properties (e.g., logical ordering or decreasing count).

2.4 *Navigation vocabulary*

An important issue is which subset X of formulas should be used as a vocabulary in the computation of concept views ? The idea is to make it depend on the actual logical context, rather than to define it *a priori*, which would not be very different from using a formal context and conceptual scaling. Given a logical context K, we want to specify how the set of formulas X should be derived. Whenever the logical context K changes, this set X should be updated accordingly. Users should also have the ability to add and remove formulas in the vocabulary in order to customize it.

A first fact is that all formulas appearing in object descriptions should be part of the vocabulary X, because they are obviously relevant to it. However, this is not enough because this would make the vocabulary an almost flat list of formulas. We can safely exclude from X any formula that has no instance. Of course, when the classification changes, a formula can become relevant by gaining instances, hence

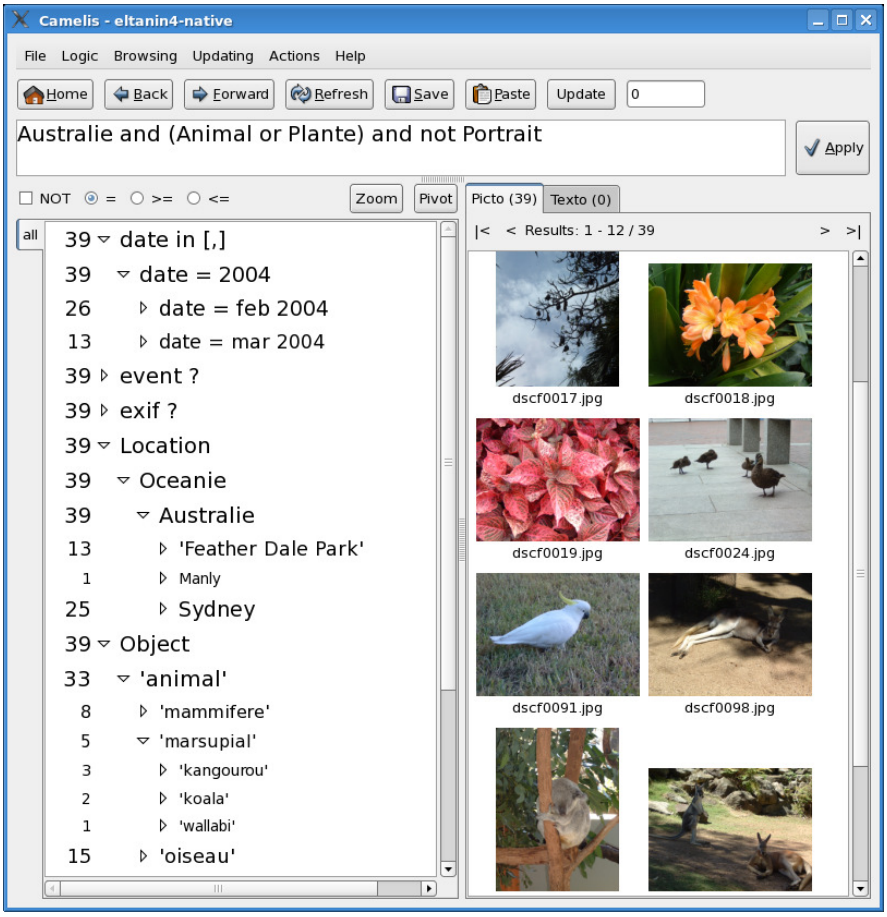


Figure 2. A screenshot of CAMELIS after a few navigation steps.

the need to synchronize the set X w.r.t. the logical context. Therefore, we may define the vocabulary as the set of all formulas having at least an instance,

$$X = \{f \in L \mid \exists o \in \mathcal{O} : (o : f)\},$$

but this is in general too large or even infinite. Indeed, we would get all date intervals containing any date occurring in some object description, and all substrings of strings occurring in the logical context. An alternative is to equip the logic with an additional operation that returns a set of abstract formulas from a concrete formula.

Definition 2.11: Let $\mathcal{L}$ be a logic. An *abstraction operation* over the logic $\mathcal{L}$ is a function from any formula $f \in L$ to a finite set of formulas $abstr(f) \subseteq L$ such that, for every $g \in abstr(f)$, g is an *abstraction* of f, i.e., we have $f \sqsubseteq g$.

In the logic of the photo application, we can define the abstraction operation such that every precise date generates the month, the year, and the decade; every location generates encompassing locations; and, every string generates a substring for each word. For instance, the formula `Montpellier` generates the formulas `France` and `Europe`; the formula `date = 24 oct 2007` generates the formulas `date = oct 2007`, `date = 2007`, and `date in [2000,2010]`; and the formula `event is "CLA 2007"` generates `event contains "CLA"`, and `event contains "2007"`. From this abstraction operation, we can now define the default definition of X, as proposed by the system given a logical context.

Definition 2.12: Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a logical context. The *derived vocabulary* is the set of formulas

$$X = \{f \in L \mid \exists o \in \mathcal{O} : f \in abstr(d(o))\}.$$

As said above, the derived vocabulary is incrementally synchronized with the logical context. If a formula f is added to the description of an object, then f and all its abstractions are automatically added into the derived vocabulary. If, after removing a formula from the description of an object, some formulas in X have no more instances, they are removed from the derived vocabulary. In this way, the derived vocabulary is always maintained consistent with the actual data.

In the event the logic is the union of some elementary logics, the vocabulary is made only of elementary formulas because Boolean combinations can be composed through the navigation process by selecting elementary formulas (see Section 3.1). Because the derived vocabulary is automatically produced, it may not fit exactly user needs. It may contain spurious formulas, and lack useful formulas. This is why we allow users to add and remove formulas in the vocabulary at any time, in the course of navigation for instance.

The following sections show how the concept view can be used to navigate among photos along various directions, and to get feedback about selected extents.

3. Browsing a collection of documents

Browsing a logical context takes place in the full logical concept lattice, where every set of objects that can be characterized by a logical query forms a concept. Navigation in this concept lattice consists of following *navigation links* that lead from one concept view to another. A navigation link is not defined by the destination concept itself, but by a transformation of the current query, which leads to another concept, and which results in the update of the extent and the index, thus producing a new concept view. Different kinds of query transformations, i.e. navigation links, are available by acting on different parts of the concept view. Navigating downward, upward and sideward in the concept lattice, as well as pivoting, are available through the selection of index properties (Section 3.1). Querying by formula is available by directly editing the query (Section 3.2). Querying by example is available through the selection of objects in the extent (Section 3.3). A history of concept views is also maintained so as to support navigation backward and forward in the history. The initial query is `all`, such that the initial concept is the one whose extent contains all objects.

Those browsing modes are illustrated on a collection of personal photos taken over the period 1999-2007. The logical context contains 5,820 objects with an average of 40 descriptors. The derived formal context has a vocabulary of 35,712 formulas, and is filled with as much as 763,614 crosses, i.e. an average of 131 properties per object (including the 40 descriptors). Indeed, whenever a photo is described by a formula (e.g., a date), it acquires abstractions of this formula as additional properties (e.g., month, year). The process of building such a context is developed in Section 4, and on average, only 10 descriptors were provided by hand for each photo.

3.1 Navigation

The principle of navigation is to interpret index properties as navigation links from the current concept to other concepts. Intent properties link to more general concepts (upward navigation), and non-intent properties link to more specific concepts (downward navigation). Upward and downward navigation can be combined in such a way as to provide sidewards navigation. Pivot navigation makes use of index properties as new starting points for browsing. Because navigation links are defined as query transformations rather than edges in the Hasse diagram representation of the concept lattice (Carpineto and Romano 1996, Cole *et al.* 2003), concepts that are accessible in one navigation step are not necessarily lower and upper neighbors of the current concept. For instance, long downward jumps in the concept lattice are possible with the most specific properties of the index. The advantage of our approach is that the set of navigation links is defined in terms of the vocabulary and the current selection of objects (the extent), instead of the concept lattice. We think the concept lattice is relevant to researchers and developers of information systems, but not to end users, who only want to focus on their data (objects and their properties).

A navigation link is decomposed into a *selection* and a *navigation mode*. This means that a same selection can be used in different ways to reach different concepts. In the simple case, a selection is a property from the index. In the general case, a selection is the disjunction of the properties selected in the index (e.g., `France or Germany or Italy`). Buttons in the interface can be activated to apply modifiers on such selections: adding negation (e.g., `not (Animal or Plant)` from the selection of `Animal` and `Plant`), replacing equalities by inequalities (e.g., `date >= 2002` from the selection of `date = 2002`). We define and illustrate the following four navigation modes: downward for specializing the query, upward for generalizing the query, sideward as a combination of downward and upward navigation, and pivot for restarting a new search from the results of a previous search. The definitions of navigation modes rely on the fact that queries can be put in conjunctive normal form, i.e. conjunctive sets of simpler queries. For instance, `France and not date <= 2000` is equivalent to {`France`, `not date <= 2000`}.

3.1.1 Downward navigation

Firstly, suppose some user, say Lisa, wants to find some photos from Australia. She first expands the property `Location`, and finds photos from Europe (5346), Africa (162), and Australia (148). After selecting the property `Australia`[1]:

- the query becomes `Australia`,
- the extent displays the first page of the 148 selected photos.
- the property `Australia` becomes part of the intent (count = 148), and it is automatically expanded to show sub-locations of Australia,
- the properties `Europe` and `Africa` are no more visible, because no more relevant (count = 0),

Then Lisa expands the property `Type` and see there are different types of photos: e.g., buildings (29), animals (34), plants (6). She gets interested in Australian living things, so she selects both `Animal` and `Plant`, which leads to the refined query `Australia and (Animal or Plant)`, whose extent contains 40 photos. One of those photos is a portrait, which Lisa does not want, so she selects the negation of `Portrait`, which leads her to the new query `Australia and (Animal or Plant)`

[1]French words can be seen in screenshots because it is the real photo collection of the author, but English translations are used in the text for better understanding.

`and not Portrait` (39 photos). By expanding more properties, she discovers that those photos were taken in February and March 2004, mainly in Sydney and the Feather Dale Park, and that 5 photos of three different species of marsupials are present: e.g., kangaroo, koala, wallaby.

Figure 2 shows the interface obtained after the previous navigation operations. At this stage, Lisa can either browse the 39 photos in CAMELIS, or launch a slideshow in an external application.

Those three navigation steps led to concept views with more and more precise queries, and hence smaller and smaller extents. This is called *downward* navigation, or *zoom-in*, because it corresponds to moving downward in the concept lattice, towards smaller extents. Its principle is to specialize the current query q by the selection x. A simple definition of the resulting query would be q `and` x, but this would entails redundancy in queries: e.g., `Australia and Sydney` which is equivalent to `Sydney` because `Sydney` is subsumed by `Australia` in the logic. A better definition is to replace by x every part of the query that is subsumed by x:

$$q \rightsquigarrow (q \setminus \{y \in q \mid x \sqsubseteq y\}) \cup \{x\}.$$

When the selection is a non-intent property, the new concept view is always strictly smaller but not empty. This is a big advantage compared to pure querying systems, where it is common to get empty results. Another advantage is that the index gives an informative feedback on the current extent. For instance, after selecting photos having some location and type, the index displays the dates and visible objects of those photos, and for each property, how many of them have it.

3.1.2 Upward navigation

During downward navigation, Lisa sometimes wants to remove or generalize some properties in the query so as to obtain larger extents: this is *upward* navigation, or *zoom-out*, because it corresponds to moving upward in the concept lattice, towards larger extents. For instance, she realizes there are not enough photos of animals and plants. If she wants to remove the last selection, moving backward in the history is a simple way to achieve this. But if she wants to remove the first selection `Australia`, she would need to move three steps backward, and re-select the last two refinements. She could also edit the query by hand, but users usually prefer to navigate rather than to edit queries. Further, intent properties are shared by all extent objects, and so cannot be used for downward navigation. This makes them available for upward navigation. When an intent property occurring in the query is selected, it is removed from the query.

$$q \rightsquigarrow q \setminus \{x\}.$$

For instance, if Lisa selects `Australia`, the new query is `(Animal or Plant) and not Portrait` (282 photos from multiple locations). When an intent property subsuming parts of the query is selected, it replaces those subsumed parts in the query.

$$q \rightsquigarrow (q \setminus \{y \in q \mid y \sqsubseteq x\}) \cup \{x\}.$$

For instance, if she selects `Pacific`, the new query is `(Animal or Plant) and not Portrait and Pacific`.

3.1.3 Sideward navigation

We show in this section that downward and upward navigation can be combined in two forms of *sideward navigation*. From the previous query `Australia`

`and (Animal or Plant) and not Portrait`, we first select the property `Plant` to navigate downward to the query `Australia and not Portrait and Plant` (6 photos). This is our starting point for sideward navigation.

At this point, Lisa sees that 1 photo has also the type `Landscape`, which interests her. So she selects this property (downward navigation), and as the result has only 1 photo, she generalizes it by removing the property `Plant` from the query (upward navigation). We have done a sidestep from Australian plants (6 photos) to Australian landscapes (80 photos), replacing in the query the property `Plant` by the property `Landscape`. From there, she performs a new sidestep from the property `Landscape` to the property `Building`, now watching 28 photos of Australian buildings. These steps are suggested and supported by photos sharing two properties. This illustrates the relevance of assigning several types to photos, which is common in this photo context. The same would apply to persons visible on photos, as a photo often shows several people.

The same does not apply to locations, as a photo cannot be taken in two incomparable locations (e.g., in Australia and in Europe). However it is still possible to navigate sideward among locations, through the taxonomy of locations. Suppose Lisa wants to find building photos from other locations. She first generalizes `Australia` by `Location` in the query (upward navigation), and then browses suggested locations before selecting `Spain` (downward navigation). She thus has done a sidestep from Australian buildings to Spanish buildings, and finds 18 photos (which are, according to the index, mainly churches taken in the north-west of Spain in 2003).

The former form of sideward navigation is a downward-upward combination, and can be qualified as *contextual* because it relies on objects sharing some properties. The latter form of sideward navigation is an upward-downward combination, and can be qualified as *logical* because it relies on subsumption relations between properties.

3.1.4 Pivot navigation

A user may not remember a property she wants to use to refine the query, but she can find it through another query. For instance, suppose Lisa wants to retrieve photos of the buildings of some town. She does not remember which town it is, but she remembers that the ICFCA conference took place there in 2004. Therefore, she can first reach the query `event contains "ICFCA" and date = 2004` by navigating downward. The resulting extension shows photos of ICFCA'04, and the index shows relevant information about these photos, such as precise dates, locations, and so on. By expanding locations in the index, she discovers that Sydney, in Australia, is the location of ICFCA'04. Then, she can make the query become `Sydney`, and further refine it to the desired query `Sydney and Building` (downward navigation). The property `Sydney` plays the role of a *pivot* between the two queries.

Pivot navigation relies on the ability of the concept view to answer queries not only by a set of objects (the extent), but also by a set of properties (the index). In previous navigation modes, these properties where added or removed from the query, whereas here they are used as new queries. Given a query q and a selection x, the query transformation is defined by

$$q \rightsquigarrow x.$$

Therefore, pivot navigation is a way to restart a search from the results of a first search. This kind of navigation has also been applied in collaborative websites (Millen *et al.* 2006, Zhou *et al.* 2008).

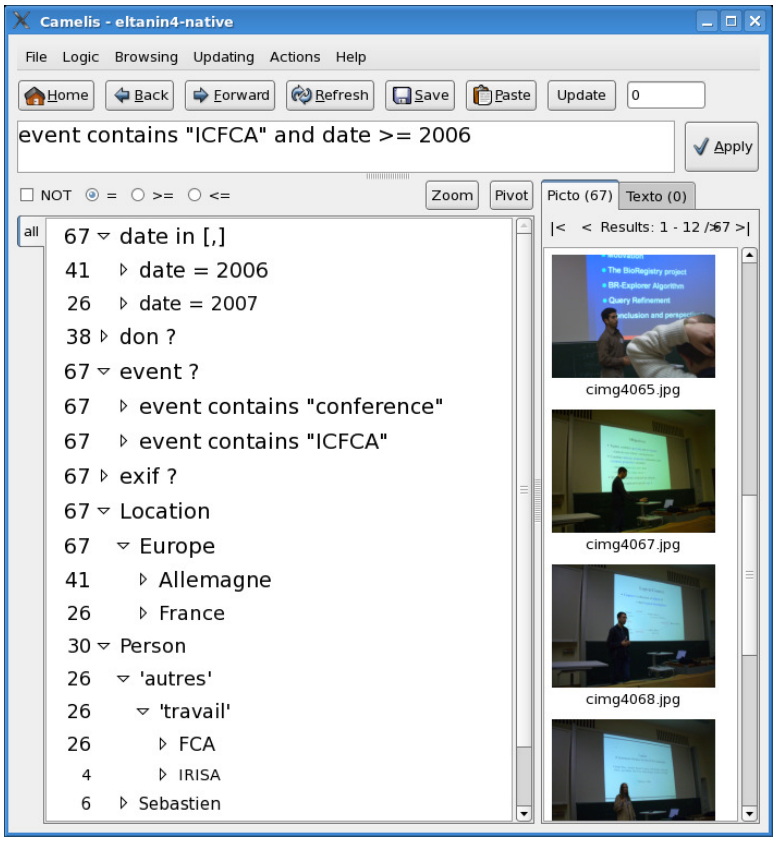


Figure 3. A screenshot of CAMELIS after entering a query.

There is an interesting analogy with natural language. Indeed, the query above can be rephrased as "photos of buildings in the town, *where* the ICFCA conference took place in 2004". The idea of pivot is reflected by the fact that Sydney occurs in the main sentence as "town", and in the relative sentence as the relative pronoun "where". The relative pronoun indicates which facet to browse for a pivot: e.g., "where" indicates a location, "when" indicates a date, and "who" indicates a person. Iterated pivot navigation then corresponds to nested relative sentences, such as "photos of buildings in the town, *where* the ICFCA conference took place in the year, *when* I also visited Hinterzarten". The first pivot is the year 2004, and the second pivot is the town Sydney.

3.2 Querying by formula

Most useful queries can be reached by a succession of navigation steps, but not all. Indeed the logic allows the expression of string patterns (e.g., on events) and arbitrary intervals (e.g., on dates), and the index cannot display them all. However it is always possible to use these patterns and intervals by directly editing the query. For instance, suppose Lisa wants to find ICFCA-related photos, she enters `event contains "ICFCA"` in the query field, and find herself in the same situation as if she had selected the corresponding property in the index. She finds that the 68 photos in the extent are scattered in three different years (2004, 2006, 2007) and in three different locations (Dresden, Clermont-Ferrand, Sydney), and they show people from the FCA community. She can further refine her search to photos taken since 2006 by modifying the query into `event contains "ICFCA" and date >= 2006`. She now finds that both year 2004, and location Sydney have disappeared

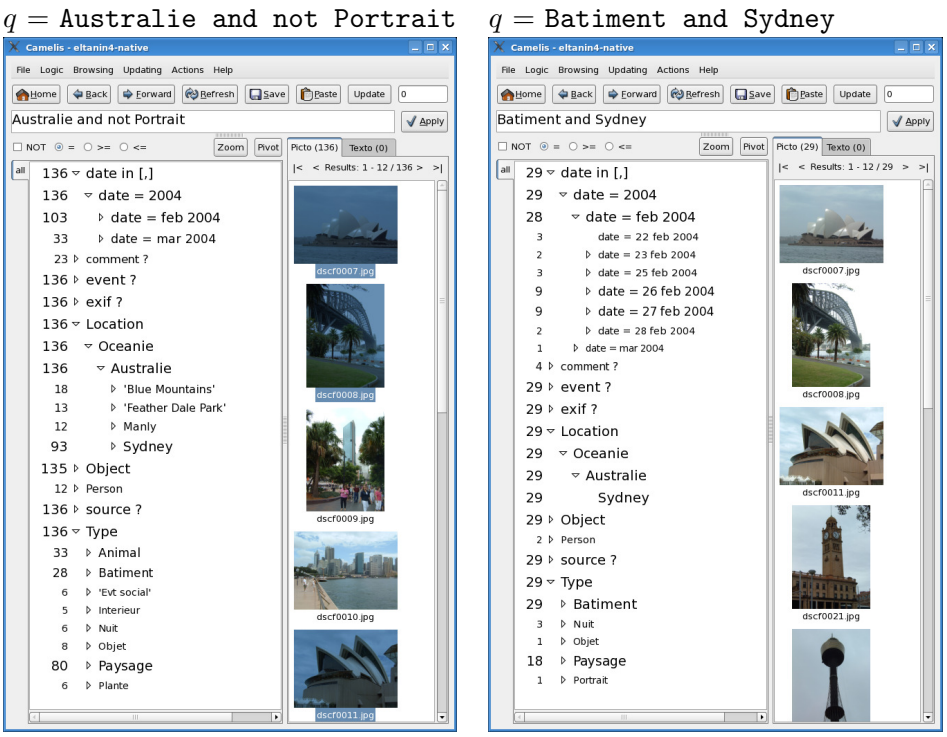


Figure 4. A screenshot of CAMELIS before and after querying by example.

from the index as they are no more relevant to the new query. The result can be seen in Figure 3.

3.3 Querying by example

In all previous sections the query is modified either by the selection of properties, or by direct edition. In this section we present how a query can be determined by the selection of a subset of photos, thus supporting querying by example. The principle is to make the query be the intent of the subset of photos.

Suppose Lisa starts with the query `Australia and not Portrait`. While browsing photos in the result, she sees interesting photos of buildings (e.g., 2 photos of the Opera, and 1 photo of the Harbor Bridge), and she would like to find more. By selecting them she moves to a new query that is the conjunction of the properties of the intent of those three photos (see Definition 2.9). She gets no additional photo, because the intent query is often very specific. At this stage, upward navigation can be used to generalize the query. Unlike approaches based on metrics, the user can choose which properties of the intent should be generalized or removed from the current query (Amato and Meghini 2008). By removing in the query properties related to date and event, the query becomes `Sydney and Building`, and she finds 29 photos. Figure 4 shows the three selected photos in the initial query (left side), and the resulting concept view of the final query (right side).

A special case of querying by example is when selecting only one photo. Then there is only one object in the extent, because there are enough properties to uniquely characterize each photo, and the query contains all the object properties, which are more easily read as intent properties in the index. So this is an easy way to access the full description of any object.

3.4 Browsing history

As in Web browsers, it is possible to navigate *backward* and *forward* in the browsing history, i.e. a stack of concept views. This is useful for opening sub-explorations in the course of an exploration. Imagine that, while browsing photos of Australian animals, Lisa finds a photo of a koala, and wants to look at all other photos of koalas. She first selects the property `'koala'`, which leads her to a new concept view with 2 photos, and then she can move back to the previous concept view. When moving back, scrolling positions are remembered, so that Lisa can go on easily browsing Australian animals.

4. Organizing a collection of documents

This section shows how the context that is used in Section 3 can be built in a reasonably efficient way for the user. An important practical need is that this process can be incremental with the arrival of new photos, and that everything that is done can be undone. The parts of a logical context that can be updated are (1) the set of objects (i.e., adding and removing photos), (2) the description of objects (i.e., adding and removing properties to objects), and (3) the taxonomic parts of the logic (i.e., moving a property downward and upward in a taxonomy).

4.1 Importing documents with intrinsic properties

After Lisa took part to CLA'07, she gets a new folder of photos taken during this event. In order to add these new photos to the photo context, she applies the command `Import files` to this folder so that each photo it contains becomes a new object in the context. These new objects come with an initial description that is automatically computed from the file location (e.g., path on the disk, URL on the Web) and file contents (e.g., JPEG picture, Java class). The navigation vocabulary is also extended with properties that are extracted from the new descriptions (e.g., the year and the month from an exact date). The properties making this initial description are called *intrinsic*. The intrinsic properties of photos are the file properties (e.g., path, size, last access date), and the EXIF metadata that is embedded in JPEG files (e.g., date, time, orientation, exposure). From there it is easy to select the newly imported photos by setting the query to the appropriate file path property (e.g., `file path contains "My Photos/CLA2007/"`).

4.2 Adding extrinsic properties to objects

The state of the art in image analysis (Datta *et al.* 2005) makes it possible to make intrinsic low-level properties of photos, such as orientation, intensity, dominant colors or textures (Martinez and Loisant 2002). However, most high-level properties such as event or visible persons, which are the most useful, cannot be determined automatically from their contents, and have to be tagged manually by users (Hyvönen *et al.* 2002). Those properties are called *extrinsic*. In fact there is no strict border between intrinsic and extrinsic properties. For instance, the location could be made intrinsic with the help of a GPS-equipped camera and a geographical information system, but these features are rarely available. Some properties (e.g., sunset) could be made intrinsic, but certainly not all. The borderline is fixed as a trade-off between the cost of manually supplying properties, and the cost of developing reliable property extraction algorithms. The advantage of extrinsic properties is that they can be customized at will to the needs of users, and the

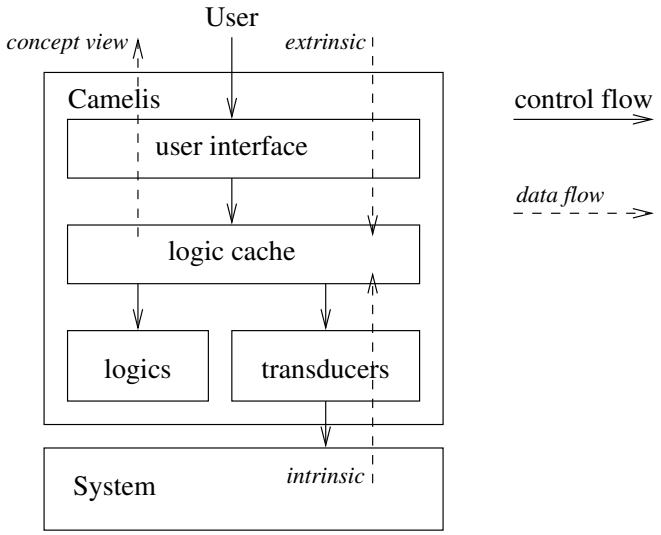


Figure 5. The architecture of CAMELIS.

design of the interface makes it efficient enough as we have experienced with the rich description of more than 5,000 photos.

The principle for efficiently giving new properties to photos is based on copy and paste. A set of photos is first copied, and then pasted on a set of properties, which can be either selected in the index, or directly entered in a text field. The effect is that every selected property is added to the description of every selected object. Removing properties is done by pasting on the negation of properties. All the new photos of CLA'07 have the same event and location, so Lisa pastes all of them into `event is "conference CLA'07" and Montpellier`. Both properties are new, and are then inserted in the property trees: `event is "conference CLA'07"` is placed under `event contains "conference"` thanks to the logic on strings, but `Montpellier` is placed at the highest level because it is a new location. The taxonomy of locations is updated with Montpellier by the drag-and-drop of `Montpellier` under `France`, which enforces a subsumption relation between the two locations.

Other properties, e.g. types, persons, objects, are added in the same way. When a property already exists, it suffices to find and select it in the property trees. Otherwise it suffices to name it. In the latter case, either it is a valued attribute and it is automatically inserted, or it is a taxonomic term and it can be moved once and for all in a taxonomy. We have observed that after some number of photos there is less and less often the need to name new properties, and that the user can rely on the property trees to maintain consistency in the use of properties. Of course the fact that there is only one author helps to construct a consistent vocabulary, but we could imagine a collaborative system under the principles of WIKIPEDIA^{TM} or FLICKR^{TM} to incrementally develop shared taxonomies.

5. Implementation issues

A key issue in implementing logical information systems is to retain the genericity of the approach. This implies separating the generic code from the code that is specific to applications. The aspects that can change from one application to another are the logic and the kind of files that can be imported. This is why CAMELIS comes with two open toolboxes of components: *logic functors* for composing *ad-hoc* logics

(Section 5.2), and *transducers* for extracting objects and their descriptions from files (Section 5.3). Those toolboxes are *open* in the sense that new components may always be needed to satisfy new application needs.

Figure 5 sketches the architecture of CAMELIS and its relation to the system and users. In addition to the two open toolboxes of logics and transducers, there are two generic parts: the logic cache and the user interface. The *logic cache* (Section 5.1) implements the generic definitions of LCA (Section 2) for rendering concept views to the user through the user interface, and handling updates. Extrinsic updates come from the user through the user interface, while intrinsic updates come from the system through the transducers. The user interface is a separate module so that it can easily be replaced by another such as a Web interface to build on-line applications.

5.1 Logic cache

In this section, we present the data structure and algorithms, along with their complexities, that are used in LIS for computing concept views, and updating the logical context. Those algorithms were designed to be generic, and so can make no assumption on the logic. They just rely on the existence of a procedure to test whether there is a subsumption relation between two formulas, and an operation to generate the set of abstractions of a formula. Because the subsumption test may be costly for some expressive logics (e.g., NExpTime-complete for SHOIN, the OWL-DL description logic (Tobies 2000)), the choice was made to minimize its use in information retrieval operations, which are more frequent than update operations. Therefore the cost of computing the subsumption test is moved to update operations, in the form of an incremental pre-processing, whose result is called a *logic cache* (Ferré and Ridoux 2004, Padioleau and Ridoux 2003).

The *logic cache* is the Hasse diagram of the navigation vocabulary of a logical context, ordered by subsumption, where each formula is labelled by its extent in the context. In this way, all subsumption relations between formulas are cached, and only the insertion of a new formula requires the use of the subsumption test $\sqsubseteq$.

Definition 5.1: Let $K = (\mathcal{O}, \mathcal{L}, d)$ be a logical context, and X be a vocabulary over $\mathcal{L}$. The *logic cache* representing the context K is the structure $C = (X, \prec, ext)$, where $(X, \prec)$ represents the Hasse diagram of the subsumption-ordered vocabulary $(X, \sqsubseteq)$, and ext is a mapping from formulas to their extent in K (Definition 2.8).

The vocabulary is mainly produced on a per-object basis by the abstraction operation (Definition 2.11), which means that every new object produces a set of formulas that already exist or that extend the existing vocabulary. We assume in the following that the size of object descriptions is bounded, which means that elementary formulas have a bounded size, and that the number of formulas produced by each object is bounded, say by k. We now briefly describe the implementation of the main LIS operations. For the evaluation of complexities, we name $n = |\mathcal{O}|$ the number of objects, $x = |X|$ the size of the vocabulary, and a the maximum number of lower neighbors of formulas in the cache.

The addition of a formula into the vocabulary consists of inserting the formula f in the logic cache as a new element, and computing its extent. The insertion consists of locating the upper and lower neighbors of f, by traversing the logic cache and performing subsumption tests. The extent of f is computed as the union of the extents of the lower neighbors. The complexity is in $O(x)$ subsumption tests for the insertion, and $O(an)$ basic instructions for computing the extent. The addition

of a new object o with description $d(o)$ consists of adding into the vocabulary every formula produced by the new object (its properties), and inserting the object o in the extent of those formulas. So, its time complexity is in $O(kx)$ subsumption tests, and $O(kan)$ basic instructions.

When the query q belongs to the vocabulary X, the computation of its extent simply consists of one access in the logic cache. Its complexity is in $O(1)$. More generally, if the query is a Boolean combination of formulas, it suffices to combine the extent of each formula by intersection, union, and complement with respect to conjunction, disjunction, and negation. The complexity is then in $O(n)$, if we consider the size of queries is bounded. For the computation of the children properties of some node in the property trees, it suffices to perform a selection among its lower neighbor in the logic cache to keep only those having a count > 0 in the current extent. This requires the intersection of two extents for each lower neighbor, hence a complexity in $O(an)$.

To summarize, the incremental computation of a logic cache (addition of n objects) relies heavily on the subsumption test, and its complexity is in $O(kxn)$ subsumption tests, and in $O(kan^2)$ basic instructions. The information retrieval operations only rely on set operations on extents, and are both linear in the number of objects, i.e. in $O(n)$ for computing the extent, and in $O(an)$ for computing the children property of a node in the property trees. The biggest context that has been built is a complete home directory, which has 95,000 objects described by an average of 50 properties each. The computation of the logic cache, including the calls to transducers, takes about 20min (1.2GHz CPU, 1Gb memory, Linux Fedora Core 6), and produces a data structure consuming 50Mb.

5.2 Logic functors

The advantage of having a system that is generic w.r.t. logic is counter-balanced by the fact that defining and implementing a logic requires an expertise that most application developers have not. For example, this requires to prove the correctness of the subsumption algorithm w.r.t. the semantics of the logic. However, it can be observed in practice that many parts of logics can be reused from one application to another. For instance, the need to represent valued attributes, taxonomies, or Boolean connectors is quite common.

We have introduced reusable logic components, called *logic functors*, that can be composed to form more complex logics (Ferré and Ridoux 2002). Logic functors are either atomic logics, or parametrized logics that expect one or several logics as arguments. The formulas of a parametrized logic are composed of formulas of its parameter logics. The same can be said for other aspects of logics: e.g., interpretations, subsumption, the abstraction operation. Section 2.1 gives three examples of logic functors. The logic of dates and the taxonomy of locations are atomic logics, while the union of logics is a parametrized logic whose parameters are the elementary logics (see Example 2.5). LogFun[1] is a toolbox of logic functors that can be used in Camelis. It contains a few dozens of logic functors, which can be grouped in the following categories:

(1) *Concrete domains*: **Int** (integers), **Float** (floating-point numbers), **Time** (hours, minutes, seconds), **Date** (day, month, year), **Char** (characters, and classes such as letters, vowels, digits), **Nt** (ADN nucleotides and chemical classes), **AA** (amino-acids and chemical classes), **String** (strings, and string

[1] `http://www.irisa.fr/LIS/ferre/logfun/`.

patterns such "contains", "begins with"), **Atom** (the classical atoms), **Taxo** (custom taxonomies), **Permission** (UNIX file right access);

(2) *Algebraic structures*:

- **Unit**: logic with a single formula,
- **Top**(L_1): adds a most general formula to the logic L_1,
- **Prod**(L_1,L_2): product of logics, formulas are pairs of formulas from L_1, L_2,
- **Sum**($\{L_k\}_k$): sum of logics, the language is the union of the languages of the logics $\{L_k\}_k$,
- **Interval**(L_1): intervals over an ordered concrete domain L_1,
- **Enum**(L_1): formulas are enumerations, i.e. disjunctive sets of formulas of L_1,
- **Segment**(L_1): segments over an ordered concrete domain L_1 (e.g., a period of time),
- **Multiset**(L_1): formulas are conjunctive multisets of formulas in L_1,
- **Motif**(L_1): complex motifs over sequences of L_1-formulas (as used in bioinformatics),
- **Union**($\{L_k\}_k$): the union logic (Definition 2.5);
- **Option**(L_1) = Sum(Unit,L_1): optional values from L_1,
- **List**(L_1) = Top(Option(Prod(L_1,List(L_1)))): lists of formulas of L_1, and patterns such as "begins with",
- **Vector**(L_1) = Option(Prod(L_1,Vector(L_1))): vectors of formulas of L_1,
- **Tree**(L_1) = Top(Prod(L_1,List(Tree(L_1)))): trees whose nodes are labelled by formulas in L_1, and patterns over trees,
- **Prop**(L_1): Boolean combinations of formulas in L_1,

(3) *Dedicated logics*: complex combinations of logic functors have been defined to represent the types of various programming languages (**Java**, **Caml**, **Mercury**), as well as of a natural language formalism (**pregroups** (Lambek 2006)).

Note how some logic functors are defined from other functors. Sometimes, those definitions are recursive like in **List**, **Vector**, and **Tree**, thus allowing the representation of recursive data structures.

An important feature of this framework is that all correctness proofs are made at the level of functors. When composing functors, the correctness of the resulting logic is automatically checked by the composer (Ferré and Ridoux 2006). This allows for an engineering methodology where logic functors are developed by a few logic experts, and certified *ad-hoc* logics are composed by application developers.

5.3 Transducers

A *transducer* is a component that is specific to a file format or file hierarchy, and extracts from them a logical context, i.e. a set of objects and their description. They save users from a lot of manual annotation. For instance, an MP3 transducer extracts information such as the artist, or the song title. Some transducers can extract several objects from a single file. This is the case of the BibTEX transducer, which produces an object for each bib-item, and a property for each field. The transducers available in the current version of Camelis belong to different categories:

(1) objects represent entire files, whose properties are:

- **File**: basename, directories, extension, last modification date, size, etc.,
- **URL**: hostname, directories, basename, extension,
- **MP3**: MP3 tags (e.g., artist, album, title, year), in addition to file properties,
- **JPEG**: EXIF tags (e.g., date, size, exposure), in addition to file properties;

(2) objects represent parts of files:

- **Comma-separated files** (.csv): lines of files, described by a valued attribute for each column,
- **Bibtex** (.bib): bibliographical references (bib-items), described by authors, title, type, journal/conference, year, etc.,
- **Mozilla email folders**: emails, described by date, sender, subject, and email folders,
- **Mozilla bookmarks**: bookmarks, described by folders and URL properties,
- **Java class** (.java): methods, described by their class, argument types, return type, etc.,
- **Caml module interface** (.mli): functions, described by their module, argument types, return type, etc.,
- **DBLP result page**: bibliographical references, described like in **Bibtex**;

(3) objects are collected through the traversal of a file hierarchy:

- **Dir**: the contents of a directory is recursively traversed, calling other transducers depending on the type of each file.

Transducers also help to synchronize the logical context with source files in an incremental fashion.

6. Related systems

The domain of information management and retrieval is very large, and it is not possible to compare LIS with all existing approaches. However, most of them fall in three common paradigms: hierarchical systems, search engines, and tag-based systems. We discuss those three paradigms as well as FCA approaches. We show that each approach can be mapped into a subset of our concept view, such that our approach encompasses most of their functionalities. The main limitation of our approach is the maximum size of collections that can be handled efficiently, which is about 100,000 at that time. This is a lot compared to other FCA approaches, but little compared to Web search engines.

6.1 *Hierarchical systems*

They are found in file systems, email tools, bookmarks, etc. The navigation space is a hierarchy of folders. In terms of concept view, the query is a folder path, the extent contains only objects right in that folder, and the index is a flat list of subfolders. Because of the hierarchy, there is only one path leading to each navigation place. This implies a strict order on the search criteria. For instance, if photos are classified first by date intervals, and then by events, then an event cannot be reached without specifying a date interval. Moreover, specifying only a date interval results in no answer but only a list of events. Searching with other criteria (e.g., visible persons), disjunction or negation is simply impossible. Of course, for every search there exists an appropriate hierarchy, but no hierarchy is appropriate to all searches. In fact, hierarchical systems would be suitable only if objects were described by a single property, and all properties belong to a single taxonomy (e.g., locations), which is obviously rarely the case.

Some file systems have been extended with *virtual folders*, whose contents is defined as the results of a query (e.g., Semantic File System (Gifford *et al.* 1991), Smart Folder in Mac OS). Virtual folders belong to the paradigm of search engines, and are integrated in the paradigm of hierarchical systems. However, virtual folders are limited in that it is not possible to navigate from them, they are not logically organized (e.g., in a taxonomy), and it is not possible to write in them, i.e., assigning

properties to files by moving them into a virtual folder.

6.2 *Search engines*

They are found on the Web, but also as utilities in systems and various software (e.g., the command `find` in Unix, virtual folders). The navigation space is a more or less expressive query language, from simple keywords to complex Boolean queries. In terms of concept view, the extent is the set of answers to the query, and there is no index. To compensate for the lack of an index, answers are usually ranked w.r.t. relevance. The only possibility to move from place to place is to manually edit the query. This means the user has no overview on the current answers, and no suggestion on how to proceed. This makes it difficult to control the number of results, and makes query results often too large or too small. Hence the need for ranking results. However, ranking is a subjective task as it depends on the intention the user has in mind. Another difficulty in the use of search engines is that updating is performed with a different interface, and only intrinsic properties are usually considered.

6.3 *Tag-based systems*

They are found in Web 2.0 applications, such as FLICKR™, YOUTUBE™ or GMAIL™. The navigation space is the same as in FCA, because queries are conjunctive sets of tags, where tags are independent binary properties. So, there is no subsumption relations between those properties, and they cannot be combined in queries by Boolean connectors. In terms of concept view, the index is called a *tag cloud*, and is the flat list of most frequent tags. A crucial difference with LIS is that this frequency is always w.r.t. the whole context, not w.r.t. the current extent. This means the tag cloud is always the same, whatever the navigation place. Moreover, when selecting a tag, the query is usually replaced by that tag, instead of being extended by it.

6.4 *FCA tools*

There also exist tools in the FCA community, which share the same navigation space as tag-based systems. The difference is that the concept lattice adds a navigation structure to suggest moves from place to place. Traditionally, a global view on the concept lattice is presented to users. However, this hardly scales up above a few dozen objects at most. This is why many concept-based information systems also rely on a concept view (Lindig 1995, Carpineto and Romano 1996, Ducrou and Eklund 2008). IMAGESLEUTH (Ducrou *et al.* 2006, Ducrou and Eklund 2008) is certainly the tool the most similar to CAMELIS w.r.t. presentation and navigation. In terms of concept view, the index is split in two lists: the intent, and other properties leading to direct sub-concepts. It provides downward and upward navigation, querying by attributes, and querying by example. It also uses *perspectives* (sets of attributes), which are in fact simple cases of 2-level taxonomies: the 1st level is made of perspectives like `Location` or `Person` for photos, and the 2nd level is made of attributes such as concrete locations or persons. It also provides a way to reach similar concepts according to some distance. According to the definition of this distance, we can say that our sideward navigation are a way to reach such similar concepts.

We think that the main advantages CAMELIS has is brought by the use of logic. Logic enables to express different kinds of properties (e.g., dates, string patterns),

and to organize them according to a well-defined subsumption relation. Logic enables users to create and customize several taxonomies. Logic enables to express complex queries where all kinds of properties can be freely combined with Boolean operators. This expressiveness is nevertheless accessible through navigation as illustrated in Section 3.1. Another major advantage of Camelis is to provide informative indexes from any query, and to support all forms of navigation (downward, upward, sideward, pivot) and querying (by formula, by example). Moreover, our user interface is expressed only in terms that can be directly understood by users: objects, their properties, the generalization ordering over properties. It provides the same advantages as concept lattices in terms of flexibility, and it is more intuitive, and also more efficient. The concept views supporting navigation and reflecting the concept lattice are computed on the fly when moving to a new concept view, or when expanding nodes in the property tree.

7. Conclusion

Logical Information Systems (LIS) reconcile querying and navigation by defining a concept view that combines a logical query and a concept extent. The concept views are connected together by the underlying concept lattice. The index part of the concept view is at the same time a feedback or summary of the extent, and a set of navigation links to other concepts. Those navigation links support the exploration of the concept lattice in many directions. Querying by formula and querying by example are also provided, but they are viewed as special kinds of navigation links as they are just another way to reach a concept view. Properties can be added or removed from the description of objects at any time, and this incrementally entails the update of the concept views.

Camelis has greatly benefited from several years of application on our collection of photos. This makes it a mature implementation of LIS, and solves the problem of organizing and retrieving photos in a rich and flexible way. Camelis has also deeply changed the way we take and share photos. We can quickly build customized slideshows. For instance, to present our region Brittany to a group of visitors, we selected all buildings and landscapes of this region, except those showing relatives. We are not reluctant any more to take photos that are irrelevant to the current event because we know we can easily find them afterwards: e.g., the photo of an animal during a conference event. This allows us to progressively gather collections of photos on various themes. For instance, we have photos for 51 different species of animals, 17 different music instruments, and 255 named persons. Beside photos, Camelis is also applied to music files (whose tags are automatically extracted as intrinsic properties), and to sets of bibliographical references (imported from BibTEX files and DBLP search results). It is possible to combine those collections of documents in a same logical context so as to browse them together, e.g., to retrieve photos and references about a same conference.

Note on contributor

Sébastien Ferré is an assistant professor at the University of Rennes 1, France, and a member of the research team LIS. He holds a PhD in Computer Science from the University of Rennes 1, and has also been an assistant researcher at the University of Wales, Aberystwyth. He works on Logical Information Systems (LIS), and has published papers about formal concept analysis, logics for knowledge representation and reasoning, information retrieval and exploration, and data-mining. His application domains are personal information management, geographical information systems, and software engineering.

References

Allard, P. and Ferré, S., 2008. Dynamic Taxonomies for the Semantic Web. *In*: G. Sacco, ed. *Int. Work. Dynamic Taxonomies and Faceted Search (FIND)* IEEE Computer Society, 382–386.

Amato, G. and Meghini, C., 2008. Faceted Content-Based Image Retrieval. *In*: G. Sacco, ed. *DEXA Work. Dynamic Taxonomies and Faceted Search (FIND)* IEEE Computer Society, 402–406.

Bedel, O., Ferré, S., Ridoux, O. and Quesseveur, E., 2008. GEOLIS: A Logical Information System for Geographical Data. *Revue Internationale de Géomatique*, 17 (3-4), 371–390.

Carpineto, C. and Romano, G., 1996. A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. *Machine Learning*, 24 (2), 95–122.

Cole, R., Eklund, P. and Stumme, G., 2003. Document Retrieval for Email Search and Discovery using Formal Concept Analysis. *Journal of Applied Artificial Intelligence*, 17 (3), 257–280.

Datta, R., Li, J. and Wang, J.Z., 2005. Content-based image retrieval: approaches and trends of the new age. *In*: H. Zhang, J. Smith and Q. Tian, eds. *Int. Work. Multimedia Information Retrieval* ACM, 253–262.

Donini, F.M., Lenzerini, M., Nardi, D. and Nutt, W., 1997. The Complexity of Concept Languages. *Information and Computation*, 134 (1), 1–58.

Ducrou, J. and Eklund, P., 2008. An Intelligent User Interface for Browsing and Search MPEG-7 Images using Concept Lattices. *Int. J. Foundations of Computer Science, World Scientific*, 19 (2), 359–381.

Ducrou, J., Vormbrock, B. and Eklund, P.W., 2006. FCA-Based Browsing and Searching of a Collection of Images. *In*: H. Schärfe, P. Hitzler and P. Øhrstrøm, eds. *Int. Conf. Conceptual Structures*, LNCS 4068 Springer, 203–214.

Ferré, S. and Ridoux, O., 2000. A Logical Generalization of Formal Concept Analysis. *In*: G. Mineau and B. Ganter, eds. *Int. Conf. Conceptual Structures*, LNCS 1867 Springer, 371–384.

Ferré, S. and Ridoux, O., 2002. A Framework for Developing Embeddable Customized Logics. *In*: A. Pettorossi, ed. *Int. Work. Logic-based Program Synthesis and Transformation*, LNCS 2372 Springer, 191–215.

Ferré, S. and Ridoux, O., 2004. An Introduction to Logical Information Systems. *Information Processing & Management*, 40 (3), 383–419.

Ferré, S. and Ridoux, O., Logic Functors: A Toolbox of Components for Building Customized and Embeddable Logics. , 2006. , Research Report RR-5871, INRIA (103 p.).

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

Gifford, D.K., Jouvelot, P., Sheldon, M.A. and O'Toole, J.W.J., 1991. Semantic file systems. ACM SIGOPS, 16–25.

Godin, R., Missaoui, R. and April, A., 1993. Experimental Comparison of Navigation in a Galois Lattice with Conventional Information Retrieval Methods. *International Journal of Man-Machine Studies*, 38 (5), 747–767.

Hyvönen, E., Styrman, A. and Saarela, S., 2002. Ontology-Based Image Retrieval. *In*: E. Hyvönen and M. Klemettinen, eds. *Towards the Semantic Web and Web Services* XML Finland Association, 15–27.

Kuznetsov, S. and Objedkov, S., 2001. Comparing Performance of Algorithms for Generating Concept Lattice. *In*: E. Mephu Nguifo, ed. *ICCS-2001 Int. Workshop on Concept Lattices-based Theory, Methods and Tools for Knowledge Discovery in Databases* CRIL – IUT de Lens, France: Stanford University.

Lambek, J., 2006. Pregroups and natural language processing. *Science+Business media*, 28 (2), 41–48.

Lindig, C., 1995. Concept-Based Component Retrieval. Morgan Kaufmann.

Martinez, J. and Loisant, E., 2002. Browsing image databases with Galois' lattices. ACM, 791–795.

Millen, D.R., Feinberg, J. and Kerr, B., 2006. Dogear: Social bookmarking in the enterprise. *In*: R.E.G. et al, ed. *Conf. Human Factors in Computing Systems (CHI)* ACM, 111–120.

Padioleau, Y. and Ridoux, O., 2003. A Logic File System. USENIX, 99–112.

Stumme, G., Taouil, R., Bastide, Y., Pasquier, N. and Lakhal, L., 2002. Computing iceberg concept lattices with Titanic. *Data Knowl. Eng.*, 42 (2), 189–222.

Tobies, S., 2000. The complexity of reasoning with cardinality restrictions and nominals in expressive Description Logics. *J. Artificial Intelligence Research (JAIR)*, 12, 199–217.

Zhou, D.X., Oostendorp, N., Hess, M. and Resnick, P., 2008. Conversation Pivots and Double Pivots. ACM, 959–968.

Appendix C

Reconciling Faceted Search and Query Languages for the Semantic Web (2012)

This journal article [**Ferré and Hermann, 2012**] has been published in the *International Journal of Metadata, Semantics and Ontologies* in 2012. It motivates and formalizes Query-based Faceted Search (QFS), and it also proves the safeness and completeness of its navigation. It introduces Sewelis as an implementation of QFS, and illustrates user interaction on a genealogical data. It reports on a user study on CS students that demonstrates the usability and effectiveness of QFS and Sewelis.

Reconciling Faceted Search and Query Languages for the Semantic Web

S. Ferré

IRISA,
University Rennes 1,
35042 Rennes, France
Fax: +33 2 99 84 71 71
E-mail: ferre@irisa.fr
*Corresponding author

A. Hermann

IRISA,
INSA Rennes,
35043 Rennes, France
Fax: +33 2 99 84 71 71
E-mail: Alice.Hermann@irisa.fr

Abstract Faceted search and querying are two well-known paradigms to search the Semantic Web. Querying languages, such as SPARQL, offer expressive means for searching RDF datasets, 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 fall in dead-ends (empty results). However, faceted search systems do not offer the same expressiveness as query languages. We introduce *Query-based Faceted Search* (QFS), the combination of an expressive query language and faceted search, to reconcile the two paradigms. We formalize the navigation of faceted search as a navigation graph, where navigation places 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). In this paper, the LISQL query language generalizes existing semantic faceted search systems, and covers most features of SPARQL. A prototype, Sewelis, has been implemented, and a usability evaluation demonstrated that QFS retains the ease-of-use of faceted search, and enables users to build complex queries with little training.

Keywords: semantic web; faceted search; query language; exploratory search; navigation; expressiveness.

Reference to this paper should be made as follows: Ferré, S. and Hermann, A. (2012) 'Combining Faceted Search and Query Languages for the Semantic Web', *Int. J. Metadata, Semantics, and Ontologies*, Vol. ?, Nos. ?/?, pp.???–???.

Biographical notes: S. Ferré is an assistant professor at the University of Rennes 1, France, and a member of the LIS research team in the IRISA laboratory. He holds a PhD in Computer Science from the University of Rennes 1, and has also been an assistant researcher at the University of Wales, Aberystwyth. He works on Logical Information Systems (LIS), and has published papers about formal concept analysis, logics for knowledge representation and reasoning, information retrieval and exploration, and data-mining. His application domains are personal information management, geographical information systems, and software engineering.

A. Hermann is a PhD student at INSA Rennes, and a member of the LIS research team in the IRISA laboratory. Her PhD subject is to adapt and extend Logical Information Systems (LIS) to the Semantic Web, both for data exploration and data creation.

1 Introduction

A key issue of the Semantic Web is 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 knowledge base (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 the second mode *exploratory search*, following Marchionini (2006). Exploratory search is often associated to *faceted search* (Hearst et al. 2002, Sacco & Tzitzikas 2009), but it is also at the core of Logical Information Systems (LIS) (Ferré & Ridoux 2000, Ferré 2009), and Dynamic Taxonomies (Sacco 2000). Exploratory search allows users to find information without *a priori* knowledge about either the data or its schema. Faceted search works by suggesting restrictions, i.e., selectors for subsets of the current selection of items. Restrictions are organized into facets, and only those that share items with the current selection are suggested. This has the advantage to provide guided navigation, and to prevent dead-ends, i.e., empty selections. Therefore, faceted search is *easy-to-use* and *safe*: *easy-to-use* because users only have to choose among the suggested restrictions, and *safe* because, whatever the choice made by users, the resulting selection is not empty. The selections that can be reached by navigation correspond to queries that are generally limited to conjunctions of restrictions, possibly with negation and disjunction on values. This is far from the expressiveness of query languages for the semantic web, such as SPARQL[1]. There are *semantic faceted search* systems that extend the expressiveness of reachable queries, but still to a small fragment of SPARQL (e.g., SlashFacet (Hildebrand et al. 2006), BrowseRDF (Oren et al. 2006), SOR (Lu et al. 2007), gFacet (Heim et al. 2010), VisiNav (Harth 2010)). For instance, none of them allows for cycles in graph patterns, unions of complex graph patterns, or negations of complex graph patterns.

Languages for querying the semantic web, such as SPARQL (Angles & Gutierrez 2008), OWL-QL (Fikes et al. 2004), or SPARQL-DL (Sirin & Parsia 2007), are quite expressive but are difficult to use, even for specialists. Users are asked to fill an empty field (problem of the writer's block), and nothing prevents them to write a query that has no answer (dead-end). 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 graphical query editor (e.g., SemanticCrystal (Kaufmann & Bernstein 2010), the SCRIBO Graphical Editor[2]) or an auto-completion system (e.g., Ginseng (Kaufmann & Bernstein 2010)) or keyword query translation (e.g., Hermes (Tran et al. 2009)), the query will be syntactically correct and semantically consistent w.r.t. the ontology but it can still produce no answer.

The contribution of this paper, *Query-based Faceted Search* (QFS), is to define a semantic 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, crossing). 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, named LISQL, generalizes existing semantic faceted search systems, and covers most features of SPARQL. 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 (list of items). It informs users in a precise and concise way about their exact position in the navigation space. 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 discusses the limits of set-based faceted search by formalizing the navigation from selection to selection. Section 4 introduces LISQL queries and their transformations. In Section 5, navigation with QFS is formalized and proved to be *safe* and *complete* w.r.t. LISQL, as well as *efficient*. Section 6 provides details about our prototype implementation Sewelis. Section 7 reports about a user study that demonstrates the usability of our approach. Our approach is also compared in Section 8 to other works in faceted search and query languages for the Semantic Web. Finally, Section 9 concludes.

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 (Hitzler et al. 2009). 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/1999/02/22rdf-syntaxns#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 2.1: 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) (Baader et al. 2003), 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*. Properties *father* and *mother* are declared as subproperties of property *parent*. The transitive closure of property *parent* is obtained by defining property *ancestor* as transitive, and a super-property of *parent*. For illustration purposes, we use genealogical datasets converted from GED files[3]. In particular, we use a small 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.

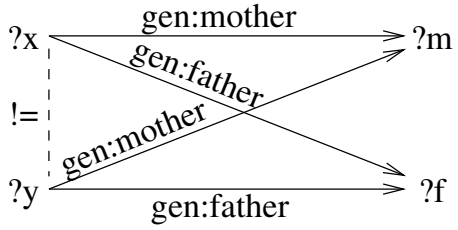


Figure 1 A graphical representation of a graph pattern.

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` syntax of SQL queries, using graph patterns in the WHERE clause. For instance, pairs of siblings can be retrieved by the following query:

```
SELECT DISTINCT ?x ?y FROM <mygen.rdf>
WHERE { ?x gen:mother ?m. ?x gen:father ?f.
        ?y gen:mother ?m. ?y gen:father ?f.
        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 siblings if they share a same mother and a same father, and are different". The FILTER condition is necessary because nothing prevents two variables to bind to a same resource. SPARQL provides a number of relational algebra operators to combine basic graph patterns: join, set union (`UNION`), left join (`OPTIONAL`), named graphs (`GRAPH`), and filtering by various constraints (`FILTER`). SPARQL 1.1 extends its expressiveness with set difference (`MINUS`), negation (`NOT EXISTS`), subqueries, aggregations, and expressions.

2.2 Faceted Search

Faceted Search (Hearst et al. 2002, Sacco & Tzitzikas 2009) 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 (Sacco & Tzitzikas 2009)). 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* (Marchionini 2006), 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*. Conversely, each feature f has a set of items, noted $items(f)$. A facet is not necessarily defined on all items. At any navigation step, the current *selection* is defined as a set of items S. The initial selection S_0 is generally the whole item collection. From the current selection S, a set of *restrictions* are computed and displayed to the user. A restriction is a feature f that matches at least one item of the current selection, i.e., the intersection between S and the set of items $items(f)$ is not empty ($S \cap items(f) \neq \emptyset$). Each restriction is generally accompanied by the number of items it matches ($\|S \cap items(f)\|$). Restrictions are organized by facets, and for the sake of conciseness, most facets are initially collapsed, and expanded on demand. On the one hand, restrictions provide a summary of the current selection. On the other hand, each restriction is a selector for a subset of the selection. 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 selection. This allows for the informed choice of a restriction: e.g., the lowest price or the highest resolution that is available given previous selected restrictions.

Exploration in faceted search is based on set operations between selections and restrictions. At each navigation step, a new selection is derived by applying a set operation between the current selection S and a restriction f chosen by the user. Typically, the set operation is intersection, i.e. $S := S \cap items(f)$. Extensions of faceted search may allow for the exclusion of a restriction ($S := S \setminus items(f)$), or the union with a restriction ($S := S \cup items(f)$). After a navigation step, a new set of restrictions is displayed to reflect the new selection. The list of chosen restrictions is generally displayed, and any of them can be removed by users, leading to a larger selection. This is useful to relax a constraint, for example in order to get more items and restrictions. The list of chosen restrictions can be seen as a *query*, which, in general, is limited to a conjunction of features, while restricted forms of negation and disjunction are sometimes available.

Dynamic Taxonomies (DT) (Sacco 2000, 2006, Sacco & Tzitzikas 2009) are a model 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 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 selection.

Logical Information Systems (LIS) (Ferré & Ridoux 2000, 2004, Ferré 2009) are another model of faceted search that has been developed in our team since 1999, on the basis of Formal Concept Analysis (Ganter & Wille 1999) and logic-based information retrieval (van Rijsbergen 1986). 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 (Ferré & Ridoux 2007). Using logics enhances the expressiveness of features and queries, as well as the design and engineering of complex taxonomies (see Chapter 8 in (Sacco & Tzitzikas 2009)). In LIS, the selection is defined as the set of answers, called *extension*, of the query, and changes of the selection are done through changes to 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 Limits of Set-based Faceted Search for the Semantic Web

The notions of faceted search can be transposed to the Semantic Web. Items and values are resources (URIs or literals), and facets are properties. The association between an item and a facet-value is a triple, where the subject is the item, the predicate is the facet, and the object is the value. Because of the relational nature of semantic data, new kinds of features and set operations have been introduced in *semantic faceted search* (e.g., /facet (Hildebrand et al. 2006), BrowseRDF (Oren et al. 2006), SOR (Lu et al. 2007), gFacet (Heim et al. 2010), VisiNav (Harth 2010)). In addition to facet-value pairs, a feature can be the name of a resource, a class as a type, the domain of a property, or the range of a property (e.g., BrowseRDF). Table 1 defines the syntax and semantics (set of items) of the various kinds of features, where r denotes any RDF resource (URI or literal), c denotes a RDFS class, p denotes a RDF property, and S_0 denotes the set of all items (possibly all resources of a RDF dataset). In semantic expressions, we use the following definitions of set-based operations involving a property p and a RDF graph G:

$$p(.,S) := \{i \in S_0 \mid \exists j \in S : (i,p,j) \in G\}$$
$$p(S,.) := \{j \in S_0 \mid \exists i \in S : (i,p,j) \in G\}$$

feature	syntax f	semantics $items(f)$	examples
name	r	$\{r\}$	`<JohnSmith>`, `"John"`, `2011`
type	`a` c	rdf:type$(.,\{c\})$	`a person`
facet-value	p `:` r	$p(.,\{r\})$	`year : 2011`
inverse facet-value	p `of` r	$p(\{r\},.)$	`mother of <JohnSmith>`
domain	p `: ?`	$p(.,S_0)$	`year : ?`
range	p `of ?`	$p(S_0,.)$	`mother of ?`

Table 1 Syntax, semantics, and examples of the various kinds of features.

Those operations can be used in addition to intersection, union, and exclusion (e.g., /facet, SOR, gFacet, VisiNav). The operation $p(S,.)$ is crossing forward p from the selection S, while the operation $p(.,S)$ is crossing backwards. For example, starting from a set S of persons, `gen:lastname`$(S,.)$ returns their lastnames, while starting from a set of lastnames, `gen:lastname`$(.,S)$ returns the set of persons having one of those lastnames. Table 2 defines the various kinds of operations that can be used to navigate from one selection to another. Crossings apply to domain and range restrictions, while other operations apply to arbitrary restrictions. In order to better expose the limits of set-based faceted search, we introduce in the table a syntax for those operations, where S denotes the current selection.

operation	syntax	semantics
reset	`?`	S_0
intersection	S `and` f	$S \cap items(f)$
exclusion	S `and not` f	$S \setminus items(f)$
union	S `or` f	$S \cup items(f)$
crossing backwards	p `:` S	$p(.,S)$
crossing forwards	p `of` S	$p(S,.)$

Table 2 Syntax and semantics of the various kinds of set-based operations.

The syntactic form of selections in Table 2 implicitly defines the language of queries (q) that can be reached by set-based faceted search:

$$q \rightarrow \texttt{?} \mid q \texttt{ and } f \mid q \texttt{ and not } f \mid q \texttt{ or } f \mid p \texttt{ : } q \mid p \texttt{ of } q$$

This grammar already defines a rich language of accessible queries, but it has strong limits in terms of flexibility and expressiveness. This can be seen at first sight by the fact that the right-hand side of intersection, difference, and union are restricted to features, instead of arbitrary queries. This linearly recursive definition of queries comes from the linear navigation of set-based faceted search. The consequence of this linearity is that not all combinations of intersection, union, and crossings are reachable, which is counter-intuitive and limiting for end users. For example, the following kinds of selections are not reachable, where the R_i represent the set of items of some features:

- unions of complex selections, e.g., $(R_1 \cap R_2) \cup (R_3 \cap R_4)$;
- or intersections of crossings from complex selections, e.g., $p_1(., R_1 \cap R_2) \cap p_2(., R_3 \cap R_4)$.

Note that a selection $S_1 \cap p(., S_2)$ cannot in general be obtained by first navigating to S_1, then crossing forwards p, navigating to S_2, and finally crossing backwards p, because it is not equivalent to $p(., p(S_1, .) \cap S_2)$ unless p is inverse functional.

Existing approaches to semantic faceted search often have additional limitations, which are sometimes hidden behind a lack of formalization. In some systems (e.g., BrowseRDF, gFacet), a same facet (a property) cannot be used several times, which is fine for functional properties but not for relations such as "child": p `: (`f_1 `and` f_2`)` is reachable but not `(`p `:` f_1`) and (`p `:` f_2`)`. In other systems (e.g., /facet), a property whose domain and range are the same cannot be used as a facet, which includes all family and friend relationships for instance.

4 Expressive Queries and their Transformations

The contribution of our approach, *Query-based Faceted Search* (QFS), is to significantly improve the expressiveness of faceted search, while retaining its properties of safeness (no dead-end), and ease-of-use. The key idea is to define navigation links at the syntactic level as query transformations, rather than at the semantic level as set operations. Indeed, the syntactic expression of a query retains more information than its semantics (a set of items) because a query has a single set of items, but a set of items can be the semantics of many different queries. The navigation from selection to selection, as well as the computation of restrictions related to the current selection, are retained because a set of items of the current query can be computed at any time.

In Section 4.1, we first define the syntax and semantics of LISQL (LIS Query Language). LISQL generalizes in a natural way the query language behind set-based faceted search (see Section 3), by allowing for the free combination of features, intersection, difference, union, and crossings. We then define in Section 4.2 a set of query transformations so that every LISQL query can be reached in a finite sequence of such transformations. This is in contrast with previous contributions in faceted search that introduce new selection transformations, and

leave the query language implicit. We think that making the language of reachable queries explicit is important for reasoning on and comparing different faceted search systems. In Section 4.3, we give a translation from LISQL to SPARQL, the reference query language of the Semantic Web. This provides both a way to compute the answers of queries with existing tools, and a way to evaluate the level of expressiveness achieved by LISQL.

4.1 The LIS Query Language (LISQL)

query	syntax q	semantics $items(q)$
name	r	$\{r\}$
type	`a` c	rdf:type(., $\{c\}$)
all	`?`	S_0
crossing backwards	p `:` q_1	$p(., S_1)$
crossing forwards	p `of` q_1	$p(S_1, .)$
complement	`not` q_1	$S_0 \setminus S_1$
intersection	q_1 `and` q_2	$S_1 \cap S_2$
union	q_1 `or` q_2	$S_1 \cup S_2$

Table 3 Syntax and semantics of LISQL queries.

LISQL is obtained by merging the syntactic categories of features and queries in the grammar of Section 3, so that every query can be used in place of a feature.

Definition 4.1: The syntax and semantics of the LISQL constructs is defined in Table 3, where r is a resource, c is a class, p is a property, S_0 is the set of all items, and q_1, q_2 are LISQL queries s.t. $S_1 = items(q_1)$ and $S_2 = items(q_2)$.

The definition of LISQL allows for the arbitrary combination of intersection, union, complement, and crossings. In order to further improve the expressiveness of LISQL from tree patterns to graph patterns, we add variables (e.g., `?X`) as an additional construct. Variables serve as co-references between distant parts of the query, and allows for the expression of cycles. For example, the query that selects people who are an employee of their own father can be expressed as `a person and father : ?X and employee of ?X`, or alternately as `a person and ?X and employee of father of ?X`. The semantics of queries with variables is given with the translation to SPARQL in Section 4.3, because it cannot be defined inductively, like in Table 3.

Syntactic constructs are given in increasing priority order (see Table 3), and brackets or indentation are used in concrete syntax for disambiguation. The most general query `?` is a neutral element for intersection, and an absorbing element for union. In the following, we use the example query q_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X`, which uses all constructs of LISQL, and selects the set of "persons born in 1601 or 1649 at some place in England, and whose father is born at another place". The same LISQL query with indentation instead of brackets displays as follows (the `and` connectors are omitted).

```
a person
birth :
   year :
      1601 or 1649
   place :
      ?X
      part of England
father : birth : place : not ?X
```

This notation better renders the structure of the query, and is therefore used in our prototype Sewelis (see Section 6).

4.2 Query Transformations

We have generalized the query language by allowing complex queries in place of features: e.g., q_1 `and` q_2 instead of q `and` f. However, because the number of suggested restrictions in faceted search must be finite, it is not possible to suggest arbitrarily complex queries as restrictions. More precisely, the *vocabulary* of features must be finite. In QFS, we retain the same set of features as in Section 3 (i.e., names, types, pairs facet-value, domain, and range), which is a finite subset of LISQL for any given dataset.

The key notion we introduce to reconcile this finite vocabulary and the reachability of arbitrary LISQL queries is the notion of *focus* in a query. This notion allows our approach to escape the linearity of set-based navigation, and therefore to reach queries with arbitrary syntax trees.

Definition 4.2: A *focus* of a LISQL query q is a node of the syntax tree of q, or equivalently, a subquery of q. The set of foci of q is noted $\Phi(q)$; the *root focus* corresponds to the root of the syntax tree, and represents the whole query. The subquery at focus $\phi \in \Phi(q)$ is noted $q[\phi]$.

In the following, when it is necessary to refer to a focus in a query, the corresponding subquery is underlined with the focus name as a subscript, like in `mother of` $\underline{?}_\phi$. Foci are used in QFS to specify on which subquery a query transformation should be applied. For example, the query `(`f_1 `and` f_2`) or (`f_3 `and` f_4`)` can be reached from the query `(`f_1 `and` f_2`) or` f_3 by applying intersection with restriction f_4 to the subquery f_3, instead of to the whole query. Similarly, the query p_1 `: (`f_1 `and` f_2`) and` p_2 `: (`f_3 `and` f_4`)` can be reached by applying the intersection with restriction f_4 to the subquery f_3. This removes the problem of unreachable selections in set-based faceted search presented in Section 3.

Definition 4.3: A *query transformation* transforms a query q into the query $q[\phi := q_1]$ by replacing the subquery at focus $\phi \in \Phi(q)$ by another query q_1. One note $q[t_1] \dots [t_n]$ if several transformations are successively applied. The also define the following abbreviations for common query transformations:

$$\begin{array}{ll} [\phi \texttt{ and } q_1] & = [\phi := q[\phi] \texttt{ and } q_1] \\ [\phi \texttt{ and not } q_1] & = [\phi := q[\phi] \texttt{ and not } q_1] \\ [\phi \texttt{ or } q_1] & = [\phi := q[\phi] \texttt{ or } q_1] \end{array}$$

We show in the following equations how the intersection $q[\phi \texttt{ and } \delta]$ with any LISQL query δ that is not a feature can be recursively decomposed into a finite sequence of intersections with features, and exclusions or unions with the most general query `?`.

δ	$q[\phi \texttt{ and } \delta]$
`?`	q
p `:` q_1	$q[\phi \texttt{ and } p \texttt{ : } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1]$
p `of` q_1	$q[\phi \texttt{ and } p \texttt{ of } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1]$
`not` q_1	$q[\phi \texttt{ and not } \underline{?}_{\phi_1}][\phi_1 \texttt{ and } q_1]$
q_1 `and` q_2	$q[\phi \texttt{ and } \underline{q_1}_{\phi_1}][\phi_1 \texttt{ and } q_2]$
q_1 `or` q_2	$q[\phi \texttt{ and } \underline{q_1}_{\phi_1}][\phi_1 \texttt{ or } \underline{?}_{\phi_2}][\phi_2 \texttt{ and } q_2]$

For example, the complex query q_{ex} = `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X` can be reached through the navigation path:

$\underline{?}_{\phi_0}$
$[\phi_0$ `and a person`$]$
$[\phi_0$ `and birth :` $\underline{?}_{\phi_1}]$
$[\phi_1$ `and year :` $\underline{\texttt{1601}}_{\phi_2}]$
$[\phi_2$ `or` $\underline{?}_{\phi_3}]$
$[\phi_3$ `and 1649`$]$
$[\phi_1$ `and place :` $\underline{?}_{\phi_4}]$
$[\phi_4$ `and ?X`$]$
$[\phi_4$ `and part of England`$]$
$[\phi_0$ `and father :` $\underline{?}_{\phi_5}]$
$[\phi_5$ `and birth :` $\underline{?}_{\phi_6}]$
$[\phi_6$ `and place :` $\underline{?}_{\phi_7}]$
$[\phi_7$ `and not` $\underline{?}_{\phi_8}]$
$[\phi_8$ `and ?X`$]$.

The classical facet-value features (p `:` r and p `of` r) appear to be redundant for navigation as their intersection can be decomposed, but they are still useful for visualization in a faceted search interface.

Sequences of query transformations are analogous to the use of graphical query editors, but the key difference is that a valid query, answers, and restrictions will be returned at each navigation step, providing feedback, understanding-at-a-glance, no dead-end, and all benefits of exploratory search. Despite the syntax-based definition of navigation steps, they have a clear semantic counterpart. Intersection is the same as in standard faceted search, only making it available on the different entities involved in the current query. In the above example, intersection is alternately applied to the person, his birth, his birth's place, his father, etc. The set of relevant restrictions is obviously different at different foci. The union transformation introduces an alternative to some subquery (e.g., an alternative birth's year). The exclusion transformation introduces a set of exceptions to the subquery (e.g., excluding some father's birth's place). In Section 5, we precisely define which query transformations are suggested at each navigation step, and we prove that the resulting navigation graph is safe (no dead-end), and complete (every "safe" query is reachable).

4.3 Translation to and Comparison with SPARQL

We propose a (naive) translation of LISQL queries to SPARQL queries. It involves the introduction of variables that are implicit in LISQL queries. This translation provides an alternative way to compute LISQL query answers, in addition to Table 3. As this translation applies to LISQL queries with co-reference variables, it becomes possible to compute their set of items.

Definition 4.4: The *SPARQL translation* of a LISQL query q is $sparql(q)$ = `SELECT DISTINCT ?`x `WHERE {` $S_0(x)$ $GP(x, q)$ `}`, where the graph pattern $S_0(x)$ binds x to an element of the set of all items S_0, and the function GP inductively defines the graph pattern of q with variable x representing the root focus.

$$\begin{array}{ll} GP(x, \texttt{?}v) & = S_0(v) \texttt{ FILTER (?}x \texttt{ = ?}v\texttt{)} \\ GP(x, r) & = \texttt{FILTER (?}x \texttt{ = } r\texttt{)} \\ GP(x, \texttt{a } c) & = \texttt{?}x \texttt{ a } c\texttt{.} \\ GP(x, p \texttt{ : } q_1) & = \texttt{?}x\ p\ \texttt{?}y\texttt{.}\ GP(y, q_1) \\ & \quad \text{where } y \text{ is a fresh variable} \\ GP(x, p \texttt{ of } q_1) & = \texttt{?}y\ p\ \texttt{?}x\texttt{.}\ GP(y, q_1) \\ & \quad \text{where } y \text{ is a fresh variable} \\ GP(x, \texttt{?}) & = \texttt{\{ \}} \\ GP(x, \texttt{not } q_1) & = \texttt{NOT EXISTS \{}\ GP(x, q_1)\ \texttt{\}} \\ GP(x, q_1 \texttt{ and } q_2) & = GP(x, q_1)\ GP(x, q_2) \\ GP(x, q_1 \texttt{ or } q_2) & = \texttt{\{}\ GP(x, q_1)\ \texttt{\} UNION \{}\ GP(x, q_2)\ \texttt{\}} \end{array}$$

The graph pattern $S_0(x)$ may depend on the application. By default, it is defined as (`?`x `a rdfs:Resource.`), and allows to select all kinds of resources, including classes, properties, and literals. The use of $S_0(x)$ in graph patterns ensures that variables are bound in filters and negations.

We now discuss the translations of LISQL queries compared to SPARQL in general. They have only one variable in the SELECT clause because of the nature of faceted search, i.e., navigation from set to set. From SPARQL 1.0, LISQL misses the optional graph pattern, and the named graph pattern. Optional graph patterns are mostly useful when there are several variables in the SELECT clause. LISQL has the `NOT EXISTS` construct of SPARQL 1.1. If we look at the graph patterns generated for intersection and union, the two subpatterns necessarily share at least one variable, x. This is a restriction compared to SPARQL, but one that makes little

difference in practice as disconnected graph patterns are hardly useful in practice.

The translation $sparql(q_{ex})$ of the above example query $q_{ex} =$ `a person and birth : (year : (1601 or 1649) and place : (?X and part of England)) and father : birth : place : not ?X` is the following.

```
SELECT DISTINCT ?x
WHERE {
   ?x1 a gen:person.
   ?x1 gen:birth ?x2.
      ?x2 gen:year ?x3.
         { FILTER (?x3 = 1601)}
         UNION { FILTER (?x3 = 1649) }
      ?x2 gen:place ?x4.
         ?X a rdfs:Resource.
            FILTER (?x4 = ?X)
         gen:England gen:part ?x4.
   ?x1 gen:father ?x5.
      ?x5 gen:birth ?x6.
         ?x6 gen:place ?x7.
            NOT EXISTS {
               ?X a rdfs:Resource.
                  FILTER (?x7 = ?X)
            }
```

This query can be manually improved as follows:

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

This example shows that LISQL is more concise, and makes a minimal use of variables. It also 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 or indentation. The LISQL syntax follows the usual syntax for expressions (infix operators and brackets/indentation to fix priorities), and we think that this makes it easier to read and learn.

5 A Safe and Complete Navigation Graph

In this section, we formally define the navigation space over a RDF dataset as a graph, where vertices are navigation places, and edges are navigation links. A navigation place is made of a query q and a focus ϕ of this query. The focus determines the selection of items to be displayed, and the set of restrictions for that selection. A navigation link is defined by a query transformation and, possibly, a focus change. We prove the safeness of navigation graphs in Section 5.1, and their completeness in Section 5.2. Finally, we discuss the efficiency of our approach relative to set-based faceted-search in Section 5.3. Before defining the navigation graph itself, we first define the set of items and the set of restrictions for some query q and some focus $\phi \in \Phi(q)$. The set of items is defined as the set of items of the query $\mathit{flip}(q, \phi)$, which is the reformulation of q from the point of view of the focus ϕ. For example, the reformulation, called the *flip*, of the query `a woman and mother of` $\underline{\texttt{name : "John"}}_{\phi}$ is the query `name : "John" and mother : a woman`.

Definition 5.1: The *flip* of a query q at a focus $\phi \in \Phi(q)$ is defined as $\mathit{flip}(q, \phi) = \mathit{flip}'(q, \phi, \texttt{?})$. The function flip' inductively deconstructs the query q until reaching the focus ϕ, and uses the additional (third) argument k as an accumulator for the resulting flipped query. In the following equations, the subquery that contains focus ϕ is underline.

$$
\begin{aligned}
\mathit{flip}'(p \texttt{ : } \underline{q_1}, \phi, k) &= \mathit{flip}'(q_1, \phi, p \texttt{ of } k)\\
\mathit{flip}'(p \texttt{ of } \underline{q_1}, \phi, k) &= \mathit{flip}'(q_1, \phi, p \texttt{ : } k)\\
\mathit{flip}'(\underline{q_1} \texttt{ and } q_2, \phi, k) &= \mathit{flip}'(q_1, \phi, k \texttt{ and } q_2)\\
\mathit{flip}'(q_1 \texttt{ and } \underline{q_2}, \phi, k) &= \mathit{flip}'(q_2, \phi, k \texttt{ and } q_1)\\
\mathit{flip}'(\underline{q_1} \texttt{ or } q_2, \phi, k) &= \mathit{flip}'(q_1, \phi, k)\\
\mathit{flip}'(q_1 \texttt{ or } \underline{q_2}, \phi, k) &= \mathit{flip}'(q_2, \phi, k)\\
\mathit{flip}'(\texttt{not } \underline{q_1}, \phi, k) &= \mathit{flip}'(q_1, \phi, k)\\
\mathit{flip}'(\underline{q_\phi}, \phi, k) &= q \texttt{ and } k
\end{aligned}
$$

When the focus is in the scope of an union, only the alternative that contains the focus is used in the flipped query. This is necessary to have the correct set of restrictions at that focus, and this is also useful to access the different subselections that compose an union. For example, in the query `a man and (`$\underline{\texttt{firstname : "John"}}_{\phi}$ `or lastname : "John")`, the focus ϕ allows to know the set of men whose firstname is John without removing the second alternative in the current query. When the focus is in the scope of a complement, this complement is ignored in the flipped query. This is useful to access the subselection to be excluded. For example, in the query `a man and not` $\underline{\texttt{father : ?}}_{\phi}$, the focus ϕ allows to know the set of men who have a father, i.e., those who are to be excluded from the selection of men.

Definition 5.2: The *items* of a query q at focus ϕ is defined as the items of the *flip* of q at focus ϕ, i.e., $\mathit{items}(q, \phi) = \mathit{items}(\mathit{flip}(q, \phi))$.

This enables the definition of the set of restrictions at each focus in the normal way. The navigation graph can then be formally defined.

Definition 5.3: The *restrictions* of a query q at focus ϕ is defined as the features that share items with the query q at focus ϕ:

$$restr(q, \phi) = \{f \mid items(q, \phi) \cap items(f) \neq \emptyset\}.$$

Definition 5.4: Let D be a RDF dataset. The *navigation graph* $G_D = (P, L)$ of D has its set of navigation places (vertices) defined by

$$P = \{(q, \phi) \mid q \in LISQL, \phi \in \Phi(q)\},$$

and its set of navigation links (edges) defined by Table 4 for every place $p = (q, \phi)$. The notation $p' = p[l]$ denotes the navigation place obtained by traversing the navigation link l from the navigation place p. One can note $p[l_1] \dots [l_n]$ when several links are traversed.

The number of navigation places is infinite because there are infinitely many LISQL queries, but the number of outgoing navigation links is finite at each navigation place because the vocabulary of features is finite, and the number of foci and variables in a query is finite. By default, the initial navigation place is $p_0 = (\underline{?}_\phi, \phi)$.

Before stating and proving safeness and completeness of the navigation graph, we state a few useful lemmas. The first lemma relates the flip of transformed queries to the transformation of flipped queries.

Lemma 5.1: *For every query q, focus $\phi \in \Phi(q)$, and query q', we verify:*

1. $flip(q[\phi \texttt{ and } \underline{q'}_{\phi'}], \phi') = flip(q, \phi) \texttt{ and } q'$,
2. $flip(q[\phi \texttt{ and not } \underline{q'}_{\phi'}], \phi') = flip(q, \phi) \texttt{ and } q'$, *(see remarks after Definition 5.1)*
3. $flip(q[\phi \texttt{ or } \underline{q'}_{\phi'}], \phi') = flip(q[\phi := \texttt{?}], \phi) \texttt{ and } q'$.

The second lemma states that intersection navigation links behave as in standard faceted search: intersection with a feature f leads to a navigation place, whose set of items is the intersection between the previous selection and the set of items matching that feature f.

Lemma 5.2: *For every query q, focus $\phi \in \Phi(q)$, and feature f, the following equality holds:*

$$items((q, \phi)[\texttt{and } f]) = items(q, \phi) \cap items(f).$$

Proof: $items((q, \phi)[\texttt{and } f])$
$= items(q[\phi \texttt{ and } \underline{f}_{\phi'}], \phi')$ (Definition 5.4)
$= items(flip(q[\phi \texttt{ and } \underline{f}_{\phi'}], \phi'))$ (Definition 5.2)
$= items(flip(q, \phi) \texttt{ and } f)$ (Lemma 5.1)
$= items(flip(q, \phi)) \cap items(f)$ (Definition 4.1)
$= items(q, \phi) \cap items(f)$ (Definition 5.2) □

The third lemma states that deletion navigation links can only make the set of items larger, and therefore cannot lead to dead-ends.

Lemma 5.3: *For every query q, focus $\phi \in \Phi(q)$, the following equality holds:*

$$items((q, \phi)[\texttt{delete}]) \supseteq items(q, \phi).$$

5.1 Safeness

Safeness is an important property to be retained from faceted search. A navigation graph is safe if no navigation path leads to a dead-end, unless it starts with a dead-end. Safeness prevents frustation in user experience. We prove that navigation graphs are safe, apart from the use of focus change (see discussion below).

Theorem 1: *Let D be a RDF dataset. The navigation graph G_D is* safe *except for some focus changes, i.e., for every path of navigation links without focus change from (q, ϕ) to (q', ϕ'), $items(q, \phi) \neq \emptyset$ implies $items(q', \phi') \neq \emptyset$.*

Proof: It suffices to prove that every navigation link l that is not a focus change is safe, i.e., that $items((q, \phi)[l]) \neq \emptyset$ assuming that $items(q, \phi) \neq \emptyset$. The proof is based on Definitions 5.2, 5.3, 5.4 and Lemmas 5.1, 5.2.

intersection: $items((q, \phi)[\texttt{and } f])$
$= items(q, \phi) \cap items(f)$ (Lemma 5.2)
$\neq \emptyset$ because $f \in restr(q, \phi)$ (Definition 5.3).

exclusion: $items((q, \phi)[\texttt{and not ?}])$
$= items(q[\phi \texttt{ and not } \underline{?}_{\phi'}], \phi')$ (Definition 5.4)
$= items(flip(q[\phi \texttt{ and not } \underline{?}_{\phi'}], \phi'))$ (Definition 5.2)
$= items(flip(q, \phi) \texttt{ and ?}) = items(flip(q, \phi))$
(Lemma 5.1)
$= items(q, \phi) \neq \emptyset$.

union: $items((q, \phi)[\texttt{or ?}])$
$= items(q[\phi \texttt{ or } \underline{?}_{\phi'}], \phi')$ (Definition 5.4)
$= items(q[\phi := q[\phi] \texttt{ or } \underline{?}_{\phi'}], \phi')$ (Definition 4.3)
$= items(q[\phi := \underline{?}_\phi], \phi)$ (? is absorbing for union)
$= items((q, \phi)[\texttt{delete}])$ (Definition 5.4)
$\supseteq items(q, \phi) \neq \emptyset$ (Lemma 5.3)
$\neq \emptyset$.

name: $items((q, \phi)[\texttt{name ?}v])$
$= items(q[\phi \texttt{ and } \underline{?v}_{\phi'}], \phi')$ (Definition 5.4)
$= items(flip(q[\phi \texttt{ and } \underline{?v}_{\phi'}], \phi'))$ (Definition 5.2)
$= items(flip(q, \phi) \texttt{ and ?}v)$ (Lemma 5.1)
$= items(flip(q, \phi))$ (because v is a fresh variable)
$= items(q, \phi) \neq \emptyset$.

reference: $items((q, \phi)[\texttt{ref ?}v]) \neq \emptyset$ by Definition 5.4.

delete: $items((q, \phi)[\texttt{delete}])$
$\supseteq items(q, \phi)$ (Lemma 5.3)
$\neq \emptyset$. □

We justify to allow for unsafe focus changes by considering the following navigation scenario. The current query has the form $q = \underline{f_1} \texttt{ or } \underline{f_2}_\phi$, i.e., the union of two restrictions. The feature f_3 is a restriction of q such that $items(f_2) \cap items(f_3) = \emptyset$, i.e., only items of f_1 match f_3. The intersection with f_3 leads to the query $q' = (f_1 \texttt{ or } f_2) \texttt{ and } \underline{f_3}_{\phi'}$, and a focus change on f_2

navigation link	notation (l)	target ($(q', \phi') = (q, \phi)[l]$)	conditions
focus change	`focus` ϕ'	(q, ϕ')	for every focus $\phi' \in \Phi(q)$
intersection	`and` f	$(q[\phi$ `and` $\underline{f}_{\phi'}], \phi')$	for every $f \in restr(q, \phi)$
exclusion	`and not ?`	$(q[\phi$ `and not` $\underline{?}_{\phi'}], \phi')$	
union	`or ?`	$(q[\phi$ `or` $\underline{?}_{\phi'}], \phi')$	
name	`name ?`v	$(q[\phi$ `and` $\underline{?v}_{\phi'}], \phi')$	for some fresh variable v
reference	`ref ?`v	$(q[\phi$ `and` $\underline{?v}_{\phi'}], \phi')$	for every $v \in vars(q)$ s.t. $items(q', \phi') \neq \emptyset$
deletion	`delete`	$(q[\phi := \texttt{?}], \phi)$	

Table 4 Definition of the set of navigation links linking a navigation place (q, ϕ) to the target navigation place (q', ϕ').

leads to an empty selection. We could prevent intersection with f_3 but this would be counter-intuitive because it is a valid restriction for (q, ϕ). We could simplify the query q' by removing the second alternative f_2 ($q' = f_1$ `and` f_3), or forbid the focus change, but we think users should have full control on the query they have built. Finally, allowing for the unsafe focus change is a simple way to inform users that no item of f_2 matches the new restriction feature f_3.

5.2 Completeness

Completeness is a key contribution of our approach, and is based on the explicit definition of a query language. A navigation graph is complete if there is a navigation path to every query that is not a dead-end, starting from the initial navigation place p_0. Completeness is important because it completely removes the need to manually edit queries, which makes guided navigation as expressive as the query language, here LISQL. This suggests that, in order to improve the expressiveness of semantic faceted search, one should first extend the query language, then extend the navigation graph with additional navigation links (i.e., query transformations), and finally prove safeness and completeness.

To be precise, completeness is proved for *safe* queries, i.e., queries without an unsafe focus change.

Definition 5.5: A query q is said to be *safe under* $\phi \in \Phi(q)$, which we note $safe(q, \phi)$, if for every focus $\phi' \in \Phi(q)$ that is equal to or under ϕ in the syntax tree of q, we have $items(q, \phi') \neq \emptyset$. Query q is said *safe*, which we note $safe(q)$, if it is safe under its root focus. By extension, we say that a navigation place (q, ϕ) is safe if $safe(q, \phi)$ holds.

Before stating and proving the main theorem on completeness, we need a few lemmas on the conservation of the safeness of queries when they are simplified. The proofs, not given here, are based on the translation of queries to SPARQL (Definition 4.4).

Lemma 5.4: *For every query q, and focus $\phi \in \Phi(q)$, if $safe(q, \phi)$, then the following propositions hold:*

1. $safe(q[\phi := \texttt{?}], \phi)$,
2. $q[\phi] = (q_1$ `and` $q_2) \Rightarrow safe(q[\phi := q_1], \phi)$,
3. $q[\phi] = (q_1$ `or` $q_2) \Rightarrow safe(q[\phi := q_1], \phi)$,
4. $q[\phi] = (p$ `:` $q_1) \Rightarrow safe(q[\phi := p$ `: ?`$], \phi)$,
5. $q[\phi] = (p$ `of` $q_1) \Rightarrow safe(q[\phi := p$ `of ?`$], \phi)$.

Theorem 2: *Let D be a RDF dataset. The navigation graph G_D is* complete *except for some queries having an unsafe focus change, i.e., for every safe query q, there is a navigation path from the initial navigation place p_0 to the navigation place $(\underline{q}_\phi, \phi)$.*

Proof: Suppose we have a navigation link [`and` q] for every safe query q, with the same definition as intersection with a restriction [`and` f] (Table 4). Then it would be possible to navigate (in one step) from the initial place p_0 to the place (`?` `and` $\underline{q}_\phi, \phi)$, which is equivalent to $(\underline{q}_\phi, \phi)$.

Therefore, we can prove completeness by showing how the hypothetical navigation link [`and` q] can be decomposed into a valid and finite navigation path, for every safe query q. The following table defines such a finite decomposition by induction on the structure of q.

q	[`and` $\underline{q}_\phi$]
r	[`and` r]
`a` c	[`and a` c]
`?`v	[`name ?`v] (if v is new)
	[`ref ?`v] (otherwise)
`?`	ϵ
p `:` q_1	[`and` p `: ?`$_{\phi_1}$][`focus` ϕ_1][`and` q_1][`focus` ϕ]
p `of` q_1	[`and` p `of ?`$_{\phi_1}$][`focus` ϕ_1][`and` q_1][`focus` ϕ]
`not` q_1	[`and not ?`$_{\phi_1}$][`and` q_1][`focus` ϕ]
q_1 `or` q_2	[`and` $\underline{q_1}_{\phi_1}$][`or ?`$_{\phi_2}$][`and` q_2][`focus` ϕ]
q_1 `and` q_2	[`and` q_1][`and` q_2][`focus` ϕ]

It remains to prove that every navigation step of the decomposition is a valid navigation link, according to Table 4. The first step is to prove that every intermediate navigation place is safe. This can be done by recurrence, where the recurrence hypothesis says that for every subquery, the initial and final places are safe. This is true for the whole query q (the base case), because q is assumed safe under ϕ, and p_0 is trivially safe. For subqueries that are decomposed into several navigation steps, the intermediate vertices can be proved safe by applying Lemma 5.4.

Finally, every atomic navigation link can be proved valid. Focus change, exclusion, union, and deletion are always valid navigation links. For intersection and reference, we can use the fact that every intermediate place is safe, and therefore that every intermediate place has a non-empty set of items. This is enough for reference. For intersection, we can use Lemma 5.2 to prove that the feature f is a restriction. □

This proof also provides an algorithm for finding a path from the initial navigation place to the target query q. It exhibits a linear complexity: the path has a length that is linear in the size of the target query.

The restriction of completeness to safe queries is not a problem in practice because every query q such that $items(q) \neq \emptyset$ (the query has answers) and not $safe(q)$ (the query has no answer at some focus) can be simplified into a safe yet equivalent query. It suffices to delete from the query empty alternatives, and empty exclusions. An empty exclusion is a subquery (`not` $\underline{q_{1\phi_1}}$) s.t. $items(q, \phi_1) = \emptyset$; and an empty alternative is either q_1 or q_2 in a subquery ($\underline{q_{1\phi_1}}$ `or` $\underline{q_{2\phi_2}}$) s.t. $items(q, \phi_1) = \emptyset$ or $items(q, \phi_2) = \emptyset$. The simplified query is equivalent to the original query in that it has the same set of items at every remaining focus, and it is safe.

5.3 Efficiency

Each navigation step from a navigation place (q, ϕ) requires the computation of the set of items $items(q, \phi)$, the set of restrictions $restr(q, \phi)$, and the set of navigation links as specified in Definition 5.4. In many cases, the set of items can be obtained efficiently from the previous set of items, and the last navigation link. If the last navigation link was an intersection, Lemma 5.2 shows that the set of items is the result of the intersection that is performed during the computation of restrictions, like in standard faceted search. For an exclusion or a naming, the set of items is unchanged. For a reference, the set of items was already computed at the previous step. Otherwise, for an union or a focus change, the set of items is computed with a LISQL query engine, possibly reusing existing query engines for the Semantic Web (see Section 4.3).

Computing the set of restrictions is equivalent to set-based faceted search, i.e., amounts to compute set intersections between the set of items of the current navigation place and the precomputed set of items of features. The same datastructures and algorithms can therefore be used. As features are LISQL queries, their set of items can be computed like for queries, possibly with optimizations given that features are simple queries. Finally, determining the set of navigation links requires little additional computation. A navigation link is available for each focus of the query (focus change), and each restriction (intersection). Three navigation links for exclusion, union, and naming are always available. Only for reference navigation links it is necessary, for each variable in the query, to compute the set of items of the target navigation place, in order to check it is not empty. This additional cost is limited as the number of variables in a LISQL query is very small in practice, and is bounded by the number of foci of the query.

6 User Interface and Interaction

Query-based Faceted Search has been implemented as a prototype, Sewelis[4]. Figure 2 shows a screenshot of Sewelis. 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 transformations (QT), a suggestion area (S) that is mainly composed of a facet hierarchy (F), a set of value boxes (V), and an answer list (A). A query engine can be derived from Sewelis by retaining only the components Q and A. A standard faceted search system can be derived by retaining only the components A, F, and V.

The query box displays the current LISQL query q, where the current focus ϕ is rendered by highlighting the subquery $q[\phi]$. In Figure 2, the whole query is highlighted because the current focus is the root focus. For a better readability of queries, we use indentation instead of brackets, URIs are represented by their labels if available or abbreviated as qualified names if possible, a concrete syntax is used for some datatypes (e.g., `42` instead of `"42"^^xsd:integer`), and resources are colored according to their kind (orange for classes, purple for properties, blue for other URIs, and green for literals).

The suggestion area (S) contains the set of restrictions for the current set of items. In fact, that set of items is a subset of the restrictions, and is displayed in the answer list (A). The classic restrictions, pairs facet-value, are found in value boxes (V), one for each applicable property (p `: ?`) and inverse property (p `of ?`). If that property is transitive (an instance of the class `owl:TransitiveProperty`), then the values are organized hierarchically accordingly. For instance, the value box of `ancestor : ?` displays a descendancy chart, while the value box of `ancestor of ?` displays an ancestry chart. Similarly, the value box of `part of ?` displays a taxonomy of locations. Other kinds of restrictions are placed in the facet box (F). This includes co-reference variables (e.g., `?X`), classes as types (e.g., `a man`), and properties as domains (e.g., `parent : ?`) and ranges (e.g., `child of ?`). Classes and properties are hierarchically organized according to the `rdfs:subClassOf` and `rdfs:subPropertyOf` transitive properties. For instance, in Figure 2, the class `a man` is a subclass of `a person`; and the properties `father : ?` and `mother : ?` are subproperties of `parent : ?`, which is itself a subproperty of `ancestor : ?`. In order to increase user feedback about suggestions, each restriction is prefixed by the number of answers that match that restriction. If a restriction matches all answers, that number is highlighted like the focus. If two restrictions match the same subset of answers, this is indicated by highlighting the two numbers with a same color. Finally, a dim font is used for restrictions that are included in the current query, hence emphasizing the other restrictions as "new". For example, in Figure 2, the restriction `a man` (bold font) is a way to make the query more specific, while `a person` (dim font) is already in the query (at the current focus). The buttons "More" and "Less" are out the

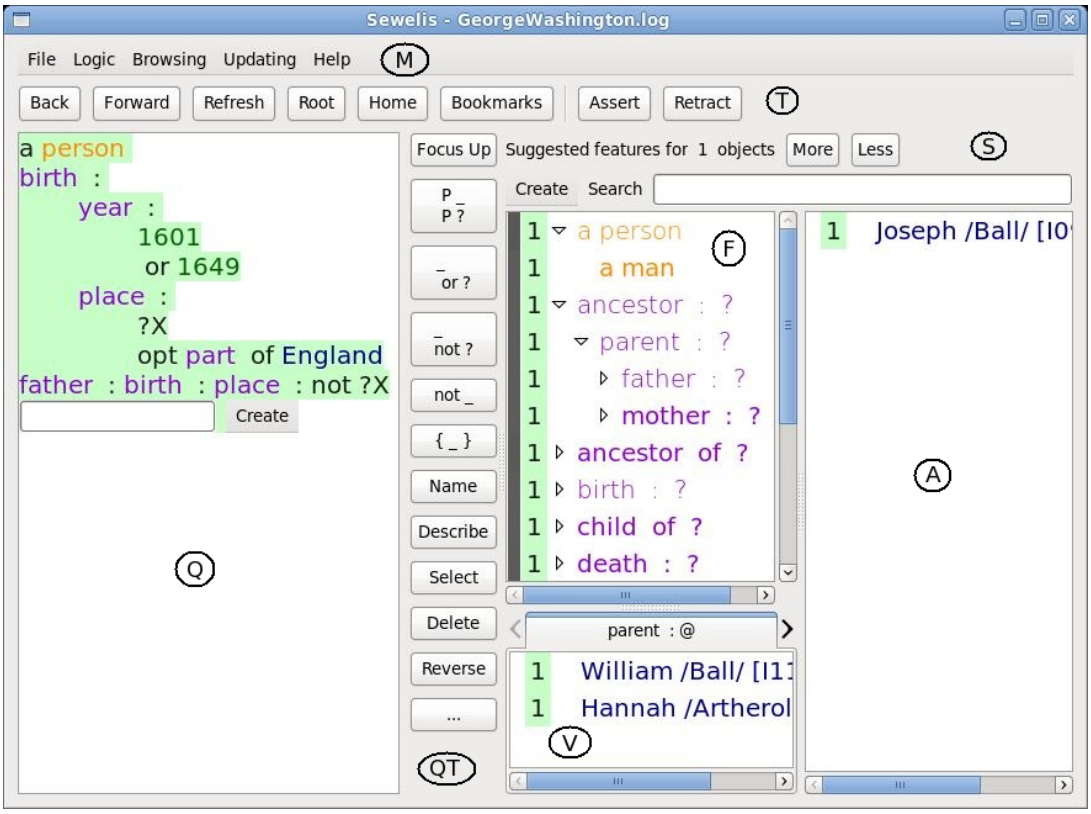


Figure 2 A screenshot of the user interface of Sewelis. It shows the selection of "persons born in 1601 or 1649 somewhere in England, and whose father was born at another place".

scope of this paper, and are mostly used when editing the RDF graph (Hermann et al. 2011).

Navigation links, i.e. the application of query transformations, are available on most components (T,Q,QT,S). Focus changes can be triggered by clicking on the subquery of interest in Q, by keystrokes (CTRL + arrows), or by pushing the button "Focus Up" in QT. Intersection with a feature and reference to a variable can be performed by double-clicking a restriction in the suggestion area (S). Other navigation links are available as buttons in the component QT (Query Transformations): "_ or ?" for union, "_ and not ?" for exclusion, "Name" for naming, and "Delete" for deletion. Other useful transformations are available in QT and T areas: "Back" and "Forward" for navigating in the history of navigation places, "Root" for jumping to the initial navigation place that contains all resources of the dataset, "Home" for jumping to the user-defined home query, "P _ and P ?" for adding a value to the same property (e.g., for describing a second child), "not _" for (un)applying negation on the current subquery, "{ _ }" for (un)quoting the current subquery as a literal, "Describe" for replacing a resource by its full description in the query, "Select" for selecting the subquery and removing the rest of the current query, "Reverse" for reformulating the query from the current focus. Other buttons ("Assert", "Retract", "...") are used for edition.

The entry field at the top of the suggestion area (S) and below the focus in the query box (Q) enables to find and select a restriction by auto-completion. This is useful when the number of restrictions is high, and the user has a clue on the text of the restriction. Matching is performed on the syntax of restrictions as rendered in the user interface, which is based on labels (using `rdfs:label`). For instance, the URI of George Washington can be retrieved by entering any of "Georges Wash", "geo wa", or even "Wash ge". The list of possible completion is updated after every keystroke. The "Create" menu near entry fields gives access to domain-specific dialogs for choosing dates, times, and filenames.

In Sewelis, data can be loaded either by importing RDF files in various formats (RDF/XML, N-Triples, Turtle), or by dereferencing URIs according to the Linked Data[5] principles. The former is available in the "File" menu, and the latter is available on every restriction that includes a URI through the contextual menu. The same contextual menu also provides means to improve the presentation of restrictions by defining labels and namespaces. Sewelis supports RDFS inference, as well as some OWL inference on properties (`owl:TransitiveProperty`, `owl:SymmetricProperty`, `owl:inverse`).

7 Usability Evaluation

This section reports on the evaluation of QFS in terms of usability[6]. 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 QFS offers expressiveness and ease-of-use at the same time.

Dataset. The datasets were chosen so that subjects had some familiarity with the concepts, but not with the individuals. We found genealogical datasets about former US presidents, and converted them from GED to RDF. We used the genealogy of Benjamin Franklin for the training, and the genealogy of George Washington for the test. The latter describes 79 persons by their birth and/or death events, which are themselves described by their year and place, by their firstname, lastname, and sex, and by their relationships (father, mother, child, spouse) to other persons. Places are linked by a transitive *part-of* relationship, allowing for the display of place hierarchies in Sewelis.

Methodology. The subjects consisted of 20 graduate students in computer science. They had prior knowledge of relational databases but neither of Sewelis, nor of faceted search, nor of Semantic Web, nor of US presidents. None was familiar with the dataset used in the evaluation. The evaluation was conducted in three phases. First, the subjects learned how to use Sewelis through a 20min tutorial, and had 10 more minutes for free use and questions. Second, subjects were asked to answer a set of questions, using Sewelis. We recorded their answers, the queries they built, and the time they spent on each question. Finally, we got feedback from subjects through a SUS questionnaire (System Usability Scale (Brooke 1996)) and open questions. The test was composed of 18 questions, with smoothly increasing difficulty. Table 5 groups the questions in 7 categories: the first 2 categories are covered by standard faceted search, while the 5 other categories are not in general. The first category, *Visualization*, did not require the creation of a query. The exploration of the suggested restrictions was sufficient: e.g., "How many men are there?". In the second category, *Selection*, we asked to count or list items that have a particular feature: e.g., "How many women are named Mary?". In the third category, *Path*, subjects had to follow a path of properties: e.g., "Which man is married with a woman born in 1708?". The fourth category, *Disjunction*, required to use unions: e.g., "Which women have for mother Jane Butler or Mary Ball?". The fifth category, *Negation*, required to use exclusions: e.g., "How many women have a mother whose death's place is not Warner Hall?". The sixth category, *Inverse*, required the backward crossing of a property: e.g., "Who was born in the same place as Robert Washington?". In the seventh category, *Cycle*, required the use of co-references: e.g., "How many persons have the same firstname as one of their parent?".

Results. Figure 3 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. For example, in 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. The difference between the number of correct queries and correct answers is explained by the fact that some subjects forgot to set the focus on the whole query after building the query, which we know from the navigation trace of subjects.

All subjects but one had correct answers to more than half of the questions. Half of the subjects had the correct answers to at least 15 questions out of 18. Two subjects answered correctly to 17 questions, their unique 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. Among all questions, the worst success rate is 50 percent. The subjects spent an average time of 40 minutes on the test, the quickest one spent 21 minutes and the slowest one 58 minutes.

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 questions 2 and 3 (we know this from the navigation traces). 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 QFS does not entail a loss of usability compared to standard faceted search for the same tasks.

For other categories, all subjects but two managed to answer correctly at least one question of each category. 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. The decrease in category *Negation* is explained by a design flaw in the interface. For category *Cycle*, we conjecture some lassitude at the end of the test. Nevertheless, all but two subjects answered correctly to at least one of *Cycle* questions. The peak of response time in category *Inverse* is explained by the lack of inverse property examples in the tutorial. It is noticeable that subjects, nevertheless, managed to solve the *Inverse* questions with a reasonable success rate, and a decreasing response time.

SUS Questionnaire. Table 6 shows the answers to the SUS questions, which are quite positive. The first noticeable thing is that, despite the relative complexity of the user interface, subjects do not find the system *unnecessarily complex* nor *cumbersome to use*. We think this is because the principles of QFS are very regular, i.e., they

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? (3)
Path	9 Which man has his father married with Alice Cooke? (5) 11 Which man is married with a woman born in 1708? (7)
Disjunction	8 Which women have for mother Jane Butler or Mary Ball? (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? (6) 15 Who died during the year when Augustine Warner 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.

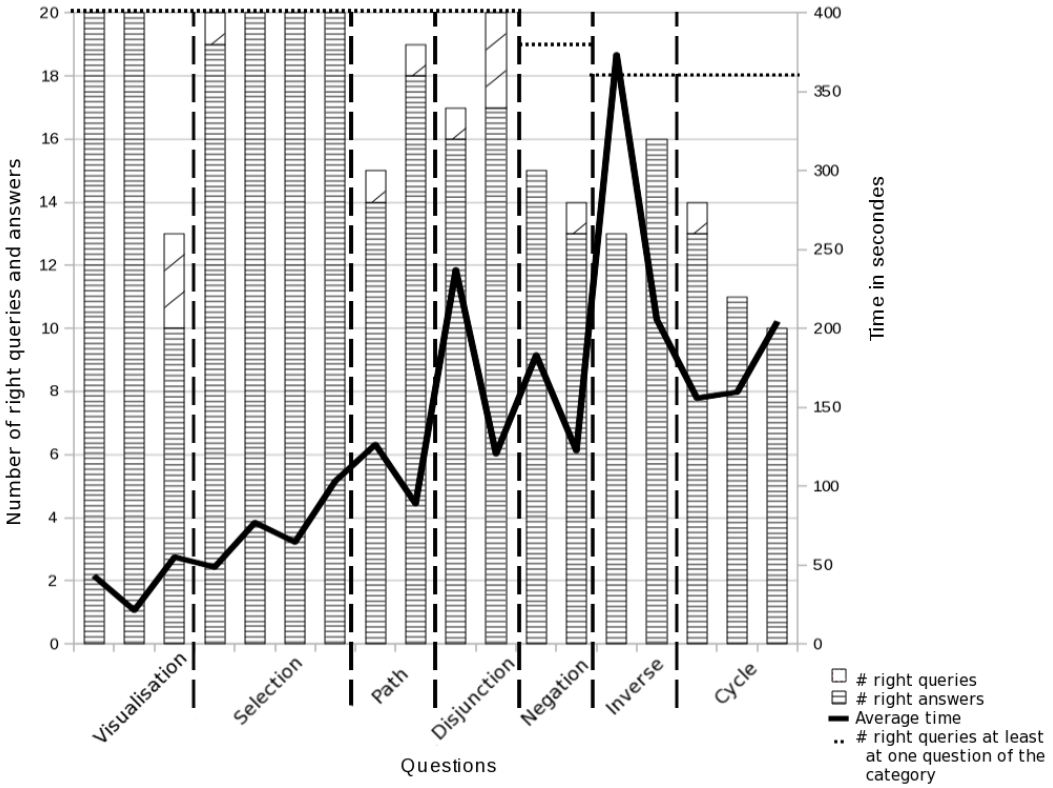


Figure 3 Average time and number of correct queries and answers for each question

follow few rules with no exception. The second noticeable thing, which may be a consequence of the first, is that subjects *felt confident using the system* and found no *inconsistency*. Finally, even if it is necessary for subjects to learn how to use the system, they *thought that the system was easy to use*, and that they *would learn to use it very quickly*. The results of the test demonstrate that they are right, even for features that were not presented in the tutorial (the Inverse category).

SUS Question	(polarity)	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 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.

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 (Mäkelä et al. 2006), 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 (Suominen et al. 2007). Therefore, their contributions are somewhat orthogonal to ours, and could certainly complement them. Other approaches, such as SlashFacet (Hildebrand et al. 2006) and BrowseRDF (Oren et al. 2006), 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 q and crossing p `:` `?` and then p `of` `?` leads to p `:` p `of` q instead of q `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 SPARQL 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 Sewelis, 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, LISQL is used to render the query in a way that fits query-based faceted search (see Section 4.1).

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 (e.g., `not father : ?`) in BrowseRDF. No other system allows to form cycles as we do with co-references.

The value boxes of SlashFacet can handle only one taxonomy of values, whereas we can use any transitive property that link the values together. For instance, when values are persons, we can use either `ancestor :` (descendancy chart), or `ancestor 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 (Sirin & Parsia 2007) for the syntactic similarity of complex classes with LISQL queries.

8.2.1 Comparison with SPARQL

Haase et al. (2004) 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 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 that SPARQL and LISQL have in common are path expressions (e.g., "the name of the author of some publication X"), union, 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. However, it covers the difference use case with the complement construct (`not`), and recursion through transitive properties. The difference use case is covered in extensions of SPARQL with the operator MINUS of Angles & Gutierrez (2008), or the operator NOT EXISTS of SPARQL 1.1. The recursion use case is covered in nSPARQL (Pérez et al. 2008), an extension of SPARQL with *nested regular expressions*. The reification use case is covered by SPARQL: e.g., "the person who has classified the publication X". As defined in Section 4.1, LISQL does not cover it, but its implementation in Sewelis does. The LISQL query for the previous example is (`a publication and ?X and topic [classifier : ?] : ?`), where the subquery into square brackets after `topic` put a constraint on the reified triple whose predicate is `topic`. This query can be navigated to, in the same way as other queries.

In total, SPARQL scores 9.5/14, LISQL scores 8/14, as defined in Section 4.1, and scores 10/14, as implemented in Sewelis. In fact, SPARQL and LISQL have a similar expressiveness, and most differences can be removed by extending either language: adding difference and recursion to SPARQL; adding multiple foci and optional pattern to LISQL.

8.2.2 Comparison with SPARQL-DL

Syntactically, LISQL queries are similar to complex classes as defined in OWL-DL. This suggests that SPARQL-DL (Sirin & Parsia 2007) 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 unions and complements but no variables (hence no cycles) and the root focus, can be translated to a SPARQL-DL query in the form `Type(?x,`q`)`, where q 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 part of England)` can be translated to

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

On the other hand, a LISQL query that contains variables but neither union nor complement, can be translated in a similar way to SPARQL-DL, 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 unions or complements.

In fact, SPARQL-DL and LISQL work at different levels, 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 *Query-based Faceted Search* (QFS) 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 navigation graph, where navigation places are queries, and navigation links are query transformations. The navigation graph is proved to be safe, because whatever the path of navigation links, the current set of items is never empty. It is also proved complete w.r.t. the query language, because for every safe query, there is a navigation path that leads to it. Finally, it is as efficient as standard faceted search w.r.t. the computation of facets and restrictions. The completeness proof

is the key result here because it draws an equivalence between expressive querying and exploratory search, therefore totally freeing users from editing queries, even the most complex ones.

The user interface of QFS 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 unions, complements, and co-references. Query transformations determines a new query language, LISQL, that is similar to SPARQL in terms of expressiveness, and with a more compact syntax. 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.

QFS has been implemented as a prototype, Sewelis. 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, while offering the expressiveness of query languages such as SPARQL.

We think that QFS is not tied to LISQL, and could be adapted to other query languages. Indeed, the definition of a navigation graph only requires the definition of the answers of a query, and the definition of query transformations. The hard part is then to prove that the resulting navigation graph is safe and complete, which we have successfully done here for LISQL.

As future work, our main objective is to fully match the expressiveness of SPARQL 1.1 by extending QFS to the few missing features: multi-dimensional queries and the OPTIONAL construct, aggregations and expressions, and built-in predicates. Other objectives are to integrate Sewelis with existing SW tools (e.g., Solr/Lucence-index for fast literal-indexing), and to perform more user evaluation in order to improve its user interface.

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. We also thank the reviewers for their useful remarks.

References

Angles, R. & Gutierrez, C. (2008), The expressive power of SPARQL, *in* A. P. S. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 5318, Springer, pp. 114–129.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. & Patel-Schneider, P. F., eds (2003), *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press.

Brooke, J. (1996), SUS: A quick and dirty usability scale, *in* P. Jordan, B. Thomas, B. Weerdmeester & A. McClelland, eds, 'Usability evaluation in industry', London: Taylor and Francis, pp. 189–194.

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

Ferré, S. & Ridoux, O. (2000), A file system based on concept analysis, *in* Y. Sagiv, ed., 'Int. Conf. Rules and Objects in Databases', LNCS 1861, Springer, pp. 1033–1047.

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

Ferré, S. & Ridoux, O. (2007), Logical information systems: from taxonomies to logics, *in* 'Int. Work. Dynamic Taxonomies and Faceted Search (FIND)', IEEE Computer Society, pp. 212–216.

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

Ganter, B. & Wille, R. (1999), *Formal Concept Analysis — Mathematical Foundations*, Springer.

Haase, P., Broekstra, J., Eberhart, A. & Volz, R. (2004), A comparison of RDF query languages, *in* S. M. *et al.*, ed., 'Int. Semantic Web Conf.', LNCS 3298, Springer, pp. 502–517.

Harth, A. (2010), 'VisiNav: A system for visual search and navigation on web data', *J. Web Semantics* **8**(4), 348–354.

Hearst, M., Elliott, A., English, J., Sinha, R., Swearingen, K. & Yee, K.-P. (2002), 'Finding the flow in web site search', *Communications of the ACM* **45**(9), 42–49.

Heim, P., Ertl, T. & Ziegler, J. (2010), Facet graphs: Complex semantic querying made easy, *in* L. A. et al., ed., 'Extended Semantic Web Conference', LNCS 6088, Springer, pp. 288–302.

Hermann, A., Ferré, S. & Ducassé, M. (2011), Guided creation and update of objects in rdf(s) bases, *in* M. A. Musen & Ó. Corcho, eds, 'Int. Conf. Knowledge Capture (K-CAP)', ACM, pp. 189–190.

Hildebrand, M., van Ossenbruggen, J. & Hardman, L. (2006), /facet: A browser for heterogeneous semantic web repositories, *in* I. C. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 4273, Springer, pp. 272–285.

Hitzler, P., Krötzsch, M. & Rudolph, S. (2009), *Foundations of Semantic Web Technologies*, Chapman & Hall/CRC.

Kaufmann, E. & Bernstein, A. (2010), 'Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases', *J. Web Semantics* **8**(4), 377–393.

Lu, J., Ma, L., Zhang, L., Brunner, J., Wang, C., Pan, Y. & Yu, Y. (2007), SOR: A practical system for ontology storage, reasoning and search (demo), *in* 'Int. Conf. Very Large Databases (VLDB)', VLDB Endowment, ACM, pp. 1402–1405.

Mäkelä, E., Hyvönen, E. & Saarela, S. (2006), Ontogator - a semantic view-based search engine service for web applications, *in* I. F. C. *et al.*, ed., 'Int. Semantic Web Conf.', LNCS 4273, Springer, pp. 847–860.

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

Oren, E., Delbru, R. & Decker, S. (2006), Extending faceted navigation to RDF data, *in* I. C. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 4273, Springer, pp. 559–572.

Pérez, J., Arenas, M. & Gutierrez, C. (2008), nSPARQL: A navigational language for RDF, *in* A. P. S. *et al*, ed., 'Int. Semantic Web Conf.', LNCS 5318, Springer, pp. 66–81.

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

Sacco, G. M. (2006), Some research results in dynamic taxonomy and faceted search systems, *in* 'Faceted Search Work. at ACM SIGIR 2006', ACM.

Sacco, G. M. & Tzitzikas, Y., eds (2009), *Dynamic taxonomies and faceted search*, The information retrieval series, Springer.

Sirin, E. & Parsia, B. (2007), SPARQL-DL: SPARQL query for OWL-DL, *in* C. Golbreich, A. Kalyanpur & B. Parsia, eds, 'Work. OWL Experiences and Directions (OWLED)', Vol. 258, CEUR-WS.

Suominen, O., Viljanen, K. & Hyvönen, E. (2007), User-centric faceted search for semantic portals, *in* E. Franconi, M. Kifer & W. May, eds, 'Eu. Semantic Web Conf.', LNCS 4519, Springer, pp. 356–370.

Tran, T., Wang, H. & Haase, P. (2009), 'Hermes: Data web search on a pay-as-you-go integration infrastructure', *Web semantics: Science, Services and Agents on the World Wide Web* **7**, 189–203.

van Rijsbergen, C. J. (1986), A new theoretical framework for information retrieval, *in* 'Int. ACM SIGIR Conference on Research and Development in Information Retrieval', ACM, pp. 194–200.

Note

[1]SPARQL `http://www.w3.org/TR/rdf-sparql-query/`
[2]The SCRIBO graphical editor `http://www.scribo.ws/-xwiki/bin/view/Blog/SparqlGraphicalEditor`
[3]GED files `http://jay.askren.net/Projects/SemWeb/`
[4]Sewelis: see `http://www.irisa.fr/LIS/softwares/-sewelis/` for a presentation, screencasts, a Linux executable, and sample data.
[5]Linked Data `http://linkeddata.org/`
[6]Usability evaluation: details can be found on `http://-www.irisa.fr/LIS/alice.hermann/camelis2.html`
[7]mSpace `http://mspace.fm/`
[8]Longwell `http://simile.mit.edu/wiki/Longwell`

Appendix D

SQUALL: a Controlled Natural Language as Expressive as SPARQL 1.1 (2014)

This journal article [**Ferré, 2014**] has been accepted and published online in the *Data and Knowledge Engineering* journal in 2014. It extends two previous papers at the Workshop on Controlled Natural Languages (CNL 2012) [**Ferré, 2012**], and at the International Conference on Applications of Natural Language to Information Systems (NLDB 2013) [**Ferré, 2013**]. It defines SQUALL as a high-level natural syntax for SPARQL 1.1. SQUALL is first translated to a logical intermediate representation, which is then translated to SPARQL. A Montague grammar is used to define the syntax and semantics of SQUALL. SQUALL covers almost all features of SPARQL 1.1, and is therefore more expressive than LISQL in Sewelis. SQUALL's naturalness is evaluated on questions of the QALD challenge.

SQUALL: the expressiveness of SPARQL 1.1 made available as a controlled natural language [1]

Sébastien Ferré[a]

[a] *IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes cedex, France*

Abstract

The Semantic Web (SW) is now made of billions of triples, which are available as Linked Open Data (LOD) or as RDF stores. The SPARQL query language provides a very expressive way to search and explore this wealth of semantic data. However, user-friendly interfaces are needed to bridge the gap between end-users and SW formalisms. Navigation-based interfaces and natural language interfaces require no or little training, but they cover a small fragment of SPARQL's expressivity. We propose SQUALL, a query and update language that provides the full expressiveness of SPARQL 1.1 through a flexible controlled natural language (e.g., solution modifiers through superlatives, relational algebra through coordinations, filters through comparatives). A comprehensive and modular definition is given as a Montague grammar, and an evaluation of naturalness is done on the QALD challenge. SQUALL is conceived as a component of natural language interfaces, to be combined with lexicons, guided input, and contextual disambiguation. It is available as a Web service that translates SQUALL sentences to SPARQL, and submits them to SPARQL endpoints (e.g., DBpedia), therefore ensuring SW compliance, and leveraging the efficiency of SPARQL engines.

Keywords: Query language, Semantic Web, Expressiveness, Controlled natural language, SPARQL 1.1

1. Introduction

An open challenge of the Semantic Web [3] is *semantic search*, i.e., the ability for users to browse and search semantic data according to their needs. Semantic search systems can be classified according to their *usability*, the *expressive power* they offer, their *compliance* to Semantic Web standards, and their *scalability*. The most expressive approach by far is to use SPARQL [4], the standard RDF query language. SPARQL 1.1 [5] features graph patterns, filters, unions, differences, optionals, aggregations, expressions, subqueries, ordering, etc. However, SPARQL is also the least usable approach, because of the gap between users and the formal languages that RDF and SPARQL are. There

[1] This paper extends previous papers [1, 2] with substantial improvement of the SQUALL language, its presentation, and its evaluation.

Email address: ferre@irisa.fr (Sébastien Ferré)

are mainly two approaches to make semantic search more usable: navigation and natural language (NL). Navigation is used in *semantic browsers* (e.g., Fluidops Information Workbench[2]), and in *semantic faceted search* (e.g., SlashFacet [6], BrowseRDF [7], Sewelis [8]). Semantic faceted search can reach a significant expressiveness [8], but still much below SPARQL 1.1, and it does not scale easily to large datasets such as DBpedia[3]. Natural language is used in Ontology-based Query Answering (OQA) systems [9] in various forms, going from full natural language (e.g., FREyA [10], PowerAqua [11]) to mere keywords (e.g., NLP-Reduce [12]) through controlled natural languages (e.g., Ginseng [13]). Existing systems devote the most effort to bridging the gap between lexical forms and ontology triples (mapping and disambiguation), and process only the simplest questions, i.e., generate SPARQL queries with only one or two triples. Most of them support none of aggregations (e.g., counting), comparatives, or superlatives, even though those features are relatively frequent [14]. This means that even if full natural language is allowed as input, expressiveness is in fact strongly limited.

A less studied aspect is the update of RDF datasets, i.e., the insertion and deletion of triples. SPARQL 1.1 offers an update language to this purpose but with the same usability problem as the query language. Proposals for more usable interfaces have been made in faceted search (e.g., UTILIS [15]), and as a controlled natural language (e.g., ACE [16]). We think that update (and creation) of RDF data is as important as querying for end-users because it makes them first-class citizens, rather than consumers only.

In this paper, we define and evaluate SQUALL, a Semantic Query and Update High-Level Language[4]. Its contribution is: (1) to offer the full expressiveness of SPARQL 1.1 Query/Update (SPARQL for short) apart from a few details, (2) to cover a significant fragment of natural language (English), and (3) to be defined in a domain-independent way and in a concise way (its grammar has about 120 rules). SQUALL qualifies as a Controlled Natural Language (CNL) [17, 18] because it combines a fragment of natural language syntax, and the unambiguous semantics of formal languages. The main advantage of CNLs over formal languages is a better readability and understandability by people whose background knowledge does not cover logic or computer languages. SQUALL provides a lot of syntactic flexibility in that a same SPARQL query/update can be expressed in many different ways. To the best of our knowledge, no existing CNL target SPARQL queries and updates. Other CNLs for the Semantic Web rather target ontologies (e.g., ACE [19], SOS and Rabbit [20]). Because the focus of this paper is on syntactic and semantic expressiveness, we only assume a domain-independent basic default lexicon that uses qualified names (e.g., `dbo:Film`) as content words. In this setting, SQUALL is less natural at the lexical level, but applicable to SPARQL endpoints without any preparation. However, our approach makes it possible to define a customized lexicon (i.e., mapping words to possibly complex semantic forms), and is in principle compatible with mapping techniques used in OQA systems (i.e., using external resources such as ontologies and WordNet). SQUALL is also compatible with guided input (like in Gingseng [13]), which is recognized as important to solve the *habitability* problem in NL interfaces [12, 9].

[2] `http://iwb.fluidops.com/`
[3] `http://dbpedia.org`
[4] SQUALL's homepage at `http://www.irisa.fr/LIS/softwares/squall`.

SQUALL is available as two Web services[5]. A *translation form* takes a SQUALL sentence and returns its SPARQL translation. A *query form* takes a SPARQL endpoint URL, namespace definitions, and a SQUALL sentence, sends the SPARQL translation to the endpoint, which returns the list of query answers. The translation of SQUALL to SPARQL ensures compliance w.r.t. SW standards, and scalability by leveraging the efficiency of SPARQL engines.

Section 2 is a short introduction to Semantic Web formalisms (RDF and SPARQL). Section 3 gives an overview of the coverage of SPARQL features by SQUALL through examples. Section 4 develops a comprehensive definition of the syntax and semantics of SQUALL, where it is shown how each feature is covered by NL constructs. The result of syntactic parsing is a semantic intermediate representation, whose translation to SPARQL is addressed in Section 5. Section 6 evaluates SQUALL's expressiveness by defining a backward translation from each SPARQL construct to SQUALL. Section 7 evaluates the NL coverage, the naturalness, and the performance of SQUALL on questions from the QALD challenge (Query Answering over Linked Data) [14]. Finally, Section 8 compares SQUALL to related work, and Section 9 concludes and discusses perspectives.

2. Semantic Web: RDF and SPARQL

The Semantic Web (SW) is founded on several representation languages, such as RDF, RDFS, and OWL, which provide increasing inference capabilities [3]. 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/1999/02/22-rdf-syntax-ns#type`), so that it is often denoted by a qualified name, e.g., `rdf:type`, where `rdf:` is the RDF namespace. In the N3 notation[6], the default namespace `:` can be omitted for qualified names that do not collide with reserved keywords (*bare qualified names*).

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 (`Bob knows Alice`) says that "Bob knows Alice", where `Bob` and `Alice` are the bare qualified names of two individuals, and `knows` is the bare qualified name of a property, i.e., a binary relation. The triple (`Bob rdf:type man`) says that "Bob has type man", or simply "Bob is a man". Here, the resource `man` is used as a class, and `rdf:type` is a property from the RDF namespace. The triple (`man rdfs:subClassOf person`) says that "man is a subclass of person", or simply "every man is a person". The set of all triples of a knowledge base forms an RDF graph.

RDF query languages [21] 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 standard RDF query language, SPARQL 1.1 [5], reuses the `SELECT FROM WHERE` shape of SQL queries, using graph patterns in the `WHERE` clause. A graph pattern G is one of:

[5] `http://lisfs2008.irisa.fr/ocsigen/squall`
[6] `http://www.w3.org/TeamSubmission/n3/`

- a triple pattern (*s p o* .) made of RDF terms and variables (e.g., `?x`),
- a join of two patterns (G_1 G_2),
- an union of two patterns (G_1 `UNION` G_2),
- a difference of two patterns (G_1 `MINUS` G_2),
- an optional pattern (`OPTIONAL` G_1),
- a filter pattern (`FILTER` C), where C is a constraint, i.e., a Boolean expression based on predicates (e.g., comparing, matching), functions (e.g., arithmetic, string concatenation), and the (un)satisfiability of a graph pattern ((`NOT`) `EXISTS` G_1),
- an assignement to a variable: either of the result of an expression (`BIND (`*expr* `AS` x`)`), or of a RDF resource (`VALUES` x `{` *res* `}`),
- a named graph pattern (`GRAPH` g G_1), where g denotes a named graph, in which the graph pattern G_1 should be matched,
- a service graph pattern (`SERVICE` *uri* G_1), where *uri* points to another SPARQL endpoint against which the graph pattern G_1 should be matched,
- a subquery.

Aggregations and expressions can be used in the `SELECT` clause (e.g., `COUNT(?x)`, `SUM(?x)`, `2 * ?x`), and `GROUP BY` clauses can be added to a query. Solution modifiers can also be added to the query for ordering results (`ORDER BY`) or returning a subset of results (`OFFSET`, `LIMIT`). Other query forms allow for closed questions (`ASK`), for returning the description of a resource (`DESCRIBE`), or for returning RDF graphs as results instead of tables (`CONSTRUCT`). SPARQL has been extended into an update language to insert/delete triples in/from a graph (`INSERT`, `DELETE`), and to manage RDF graphs (`LOAD`, `CLEAR`, etc.). The most general update form is `DELETE` D `INSERT` I `WHERE` G, where I and D can be sets of triple patterns plus named graph patterns, and G is a graph pattern that defines bindings for variables occuring in I and D.

3. SQUALL overview through SPARQL features

This section presents an overview of the SQUALL language by giving for each SPARQL feature its counterpart in SQUALL. It aims to give the reader a first taste of the language, and also a first assessment of its expressiveness compared to SPARQL. This list of SPARQL features is adapted and extended from a comparison of RDF query languages [21]. For each feature, SQUALL sentences are given as illustrations, with relevant parts underlined. For the sake of simplicity, we assume that all resources belong to a same namespace so that bare qualified names can be used (e.g., "person", "author", "Paper42"). The SPARQL translation of SQUALL sentences can be obtained from the translation form at `http://lisfs2008.irisa.fr/ocsigen/squall/`.

Triple patterns. Each noun or non-auxiliary verb plays the role of a class or predicate in a triple pattern. If a question is about the class or predicate itself, the verbs "belongs" and "relates" are respectively used.

- "Which person is the author of a publication whose publication_year is 2012?"
- "To which nationality does John_Smith belong?" (here, "nationality" is a meta-class whose instances are classes of persons: e.g., "French", "German").
- "What relates John_Smith to Mary_Well?"

Updates. Updates are obtained by declarative sentences. A sequence of declarative sentences generates a sequence of updates. Graph-level updates (e.g., `LOAD`, `CLEAR`) are obtained by imperative sentences.

- "Paper42 has author John_Smith and has publication_year 2012."
- "John_Smith know-s Mary_Well. Mary_Well know-s John_Smith."
- "Load <`http://example.org/data.rdf`> into Graph1."
- "Clear all named graphs."

Queries. `SELECT` queries are obtained by open questions, using one or several question words ("which" as a determiner, "what" or "who" as a noun phrase). Queries with a single selected variable can also be expressed as imperative sentences. `ASK` queries are obtained by closed questions, using either the word "whether" in front of a declarative sentence, or using auxiliary verbs and subject-auxiliary inversion.

- "Which person is the author of which publication?"
- "Give me the author-s of Paper42."
- "Whether John_Smith know-s Mary_Well?"
- "Does Mary_Well know the author of Paper42?"

Solution modifiers. The ordering of results (`ORDER BY`) and partial results (`LIMIT`, `OFFSET`) are expressed with superlatives.

- "Which person-s have the 10 greatest age-s?"
- "Who are the author-s of the publication-s whose publication_year is the 2nd latest?"
- "Which person is the author of the most publication-s?"

Join. The coordination "and" can be used with all kinds of phrases. It generates complex joins at the relational algebra level.

- "John_Smith and Mary_Well have age 42 and are an author of Paper42 and Paper43."

Union. Unions of graph patterns are expressed by the coordination "or", which can be used with all kinds of phrases, like "and".

- "Which teacher or student teach-es or attend-s a course whose topic is NL or DB?"

Option. Optional graph patterns are expressed by the adverb "maybe", which can be used in front of all kinds of phrases, generally verb phrases.

- "The author-s of Paper42 have which name and maybe have which email?"

Negation. The negative constraint on graph patterns (`NOT EXISTS`) is expressed by the adverb "not", which can be used in front of all kinds of phrases, and in combination with auxiliary verbs. In updates, negation entails the deletion of triples.

- "Which author of Paper42 has not affiliation Salford_University?"
- "John_Smith is not a teacher and does not teach Course101."

Quantification. Quantifiers have no direct counterpart in SPARQL, and can only be expressed indirectly with negation or aggregation. In SQUALL, they are expressed by determiners like "a", "every", "no", "some", "at least 3", "the". The latter "the" is interpreted existentially in queries, and universally in updates. The universal quantifier in updates allows for batches of updates, and corresponds to the use of a `WHERE` clause in SPARQL updates.

- "Every author of Paper42 has affiliation the university whose location is Salford."
- "Which publication has more than 2 author-s whose affiliation is Salford_University?"

Built-ins. Built-in predicates and functions used in SPARQL filters and expressions are expressed by pre-defined nouns, verbs, and relational adjectives. They can therefore be used like classes and properties.

- "Which person has a birth_date whose month is 3 and whose year is greater than 2000?"
- "Give me the publication-s whose title contains "natural language"?"

Expressions. Operators and functions are defined as coordinations so that they can be applied on different kinds of phrases: e.g., relational nouns, noun phrases.

- "Which publication has the lastPage - the firstPage + 1 greater than 10?" (page number)
- "Return concat(the firstname, " ", the lastname) of all author-s of Paper42." (fullname)

Aggregation and grouping. Aggregation is expressed by the question determiner "how many", by relational nouns such as "sum", and by adjectives such as "total", "average". Grouping clauses are introduced by the word "per".

- "How many publication-s have author John_Smith?"
- "What is the number of publication-s per author?"
- "What is the average age of the author-s of a publication per affiliation?"

Property paths. Property sequences and inverse properties are covered by the flexible syntax of SQUALL. Alternative and negative paths are respectively covered by the coordination "or" and the adverb "not". Reflexive and transitive closures of properties have no obvious linguistic counterpart, and are expressed so far by property suffixes among "?", "+", and "*". SQUALL does not yet support the transitive closure of complex property paths (e.g., `(^author/author)+` for co-authors of co-authors, etc.).

- "Which publication-s cite+ Paper42?" (i.e., *Which publications cite Paper42 or cite a publication that cites Paper42, etc?*)

Named graphs. The `GRAPH` (resp. `SERVICE`) construct of SPARQL, which serves to restrict graph pattern solutions to a named graph (resp. to a distant service), can be expressed using "in graph" (resp. "from service") as a preposition. A prepositional phrase can be inserted at any location in a sentence, and its scope is the whole sentence.

- "Who is the author of the most publication-s in graph Salford_Publications?"
- "In which graph is John_Smith the author of at least 10 publication-s?"
- "What is the `dbpedia-owl:capital` of `dbpedia:France` from service `<http://dbpedia.org/>`?"

Graph literals. The SPARQL query forms `CONSTRUCT` and `DESCRIBE` return graphs, i.e. sets of triples, instead of sets of solutions. A `DESCRIBE` query is expressed by the imperative verb "describe" followed by a resource or a universally-quantified noun phrase. A `CONSTRUCT` query is expressed by using a subordinate clause introduced by "that", which is reified into a graph literal.

- "Describe the author-s of Paper42."
- "For every publication with an author X and with an author Y different from X, return that X has coauthor Y and Y has coauthor X."

Collection patterns. SPARQL has a special notation for collection literals (e.g., `(1 2 3)`) but not for collection patterns (e.g., Paper42 has an author list whose last element is John Smith). In SPARQL, collection patterns are expressed by combining triple patterns, blank nodes, and property paths: e.g., `:Paper42 :authorList [ rdf:rest* [ rdf:first :John_Smith ; rdf:nil ] ]`. SQUALL offers concise and powerful patterns by reusing Prolog's notations plus the ellipsis, and translating them into SPARQL.

- "What has authorList [..., John_Smith]?"
- "Paper42 has authorList [..., who]?" (i.e., *Who is the last author of Paper42?*)
- "Paper42 has authorList [..., who, ...]?" (i.e., *Who are the authors of Paper42?*)
- "Paper42 has authorList [_, who, ...]?" (i.e., *Who is the second author of Paper42?*)

4. Syntax and semantics

In this section, we formally define the syntax and semantics of SQUALL in the style of Montague grammars. Montague grammars [22] are an approach to natural language semantics that is based on formal logic and λ-calculus. It is named after the American logician Richard Montague, who pioneered this approach [23]. A Montague grammar is a context-free generative grammar, where each rule is decorated by a λ-expression that denotes the semantics of the syntactic construct defined by the rule. The semantics is defined in a fully compositional style, i.e., the semantics of a construct is always a composition of the semantics of sub-constructs. The obtained semantics for a valid SQUALL sentence is represented in an intermediate logical language, rather than directly in terms of an existing query language for the Semantic Web. This is a common practice in the compilation of high-level programming languages, and has a number of advantages. First, it makes the semantics easier to write and understand because defined at a more abstract level. As a side effect, it would also make it easier to redefine SQUALL for other natural languages than English. Second, it gives freedom in the choice of the implementation. For instance, the operational semantics of the intermediate language can be given by translating it to an existing language, e.g., SPARQL; by interpreting it in a relational algebra engine; or by using continuation passing-style, like in Prolog. In Section 5, we describe a solution for the first approach.

SQUALL sentences are decomposed into noun phrases, verb phrases, relatives, determiners, prepositional phrases, etc. As an illustration, we consider a complex sentence that covers many features of SQUALL: "For which researcher-s X, in graph DBLP every publication whose author is X and whose publication_year is greater than 2000 has at least 2 author-s?". Its syntactic analysis is

[$_S$for [$_{NP}$[$_{Det}$which] [$_{NG1}$[$_{Noun1}$researcher-s] [$_{App}$[$_{Label}$X]]]],
[$_S$[$_{PP}$[$_{PrepNoun}$in graph] [$_{NP}$[$_{ProperNoun}$DBLP]]]
[$_S$[$_{NP}$[$_{Det}$every] [$_{NG1}$[$_{Noun1}$publication]
[$_{Rel}$[$_{Rel}$whose [$_{NG2}$[$_{Noun2}$author]] [$_{VP}$is [$_{NP}$[$_{Label}$X]]]]
and [$_{Rel}$whose [$_{NG2}$[$_{Noun2}$publication_year]]
[$_{VP}$is [$_{Rel}$greater than [$_{NP}$[$_{Literal}$2000]]]]]]]]
[$_{VP}$has [$_{Det}$at least 2] [$_{Noun2}$author-s]]]]],

and its semantic intermediate representation is

```
select λr.(triple r rdf:type :researcher)
    λr.(context GRAPH :DBLP (forall
        λp.(exists λy.(and
            (triple p rdf:type :publication) (triple p :author r)
            (triple p :publication_year y) (pred > y 2000)))
        λp.(exists λn.(and
            (aggreg COUNT λ[].λa.(triple p :author a) n)
            (pred >= n 2))))).
```

In the following, we successively define the semantic types and expressions (Section 4.1) used for the intermediate representation, notations for Montague grammars (Section 4.2), SQUALL's lexical units and our default lexicon (Section 4.3), and

type	definition	variable names	name of the type
e	-	x, y, z	entity
s	-	s	statement
$p1$	$e \to s$	d	description
$p2$	$e \to e \to s$	r	relation
$s1$	$p1 \to s$	q	quantifier
$s2$	$p1 \to p1 \to s$	$q2$	binary quantifier
m_α	$\alpha \to \alpha$	m	α-modifier
c_α	$\alpha \to \alpha \to \alpha$	c	α-coordination

Table 1: List of semantic types along with their definition (except for base types), the common name of their variables, and a short description of the type.

SQUALL's syntactic rules (Section 4.4). We also explain how syntactic ambiguities are resolved (Section 4.5), how non-local aspects are handled by semantic transformations (Section 4.6), and how the intermediate representation is semantically validated (Section 4.7).

4.1. Semantic types and expressions

Montague grammar are based on *simply-typed λ-calculus* [24]. Every syntagm is associated to some semantic type, and those types constrain the way semantic expressions can be combined. The two base types are e for *entities*, and s for *statements*. The main type constructor is $\alpha \to \beta$ for functions from expressions of type α to expressions of type β. The sub-types α and β can themselves be function types, recursively. For instance, an expression of type $p1 = e \to s$ expects an entity, and returns a statement: it can be seen as a statement missing an entity. For example, the verb phrase "knows Mary" has semantic type $p1$ because it misses an entity (e.g., "John") as the subject so as to make a complete statement. Table 1 lists and defines the semantic types that are used in this paper.

There are two kinds of semantic expressions associated to function types: *applications* and *abstractions*. The application of an expression f of type $\alpha \to \beta$ to an expression e of type β is noted $f\ e$, and has type β. The abstraction of an expression e of type β by a variable x of type α is noted $\lambda x.e$, and has type $\alpha \to \beta$. In the notation of expressions, abstraction has priority over application, and application is left-associative: e.g., $e_1\ \lambda x.e_2\ e_3 = ((e_1\ (\lambda x.e_2))\ e_3)$. Expressions obtained by composition can be simplified according to λ-calculus, through β-reduction ($(\lambda x.s)\ y =_\beta s[x \leftarrow y]$, where $s[x \leftarrow y]$ denotes the substitution of x by y in s), and η-expansion ($d =_\eta \lambda x.(d\ x)$, if d is a function).

For convenience, we also introduce a type constructor for lists, i.e., $[\alpha]$ for lists whose elements have type α. In semantic expressions, $[x; y; z]$ denotes a list with 3 elements, $[]$ is the empty list, and $(x :: l)$ is a list whose first element is x, and whose rest is l[7].

Constants in semantic expressions play the role of semantic constructors in the intermediate representation of SQUALL sentences. Table 2 lists all the necessary constants, called *primitives*, to define the semantics of SQUALL. Although they take their name

[7]The reader may have recognized the notations from ML [25].

primitive	type	description
triple	$e \to e \to e \to s$	triple pattern
command	$n \to [e] \to s$	command
pred	$n \to [e] \to s$	predicate call
func	$n \to [e] \to p1$	function call
aggreg	$n \to ([e] \to p1) \to p1$	aggregation
modif	$n \to m_s$	solution modifier (e.g., ordering)
context	$n \to e \to m_s$	context (e.g., named graph, service)
arg	$n \to e \to m_s$	defining extra-argument
true	s	tautology
not	m_s	negation
option	m_s	optional
and	c_s	conjunction
or	c_s	disjunction
implies	c_s	implication
where	c_s	conditional
exists	$s1$	existential quantifier
forall	$s2$	universal quantifier
the	$s2$	definite quantifier
forarg	$n \to s1$	using extra-argument
ask	m_s	query constructor
select	$s1$	interrogative quantifier
select_where	$s2$	binary interrogative quantifier
return	$p1$	interrogative imperative
label	$p2$	entity labelling by a user variable
ref	$e \to e$	entity reference by a user variable
graphliteral	$s \to e$	graph reification as entity

Table 2: List of semantic primitives, along with their type and description.

from English, they are in fact determined by SPARQL features, and independent to the source natural language. In types, the base type n is introduced as a sub-type of e for primitive arguments that correspond to names (e.g., predicates, functions, solution modifiers, extra-arguments). The table is given here in full for reference, but the meaning of each primitive is only explained on their first introduction in the Montague grammar. For convenience, we define additional constants on top of those primitives for comparators and counting:

$\mathbf{eq} : p2 = \lambda x.\lambda y.(\mathbf{pred} = [x; y])$
$\mathbf{gt} : p2 = \lambda x.\lambda y.(\mathbf{pred} > [x; y])$
$\mathbf{between} : e \to p2 = \lambda x.\lambda y_1.\lambda y_2.(\mathbf{and}\ (\mathbf{pred} >= [x; y_1])\ (\mathbf{pred} <= [x; y_2]))$
$\mathbf{count} : p1 \to p1 = \lambda d.\lambda n.(\mathbf{aggreg}\ \texttt{COUNT}\ \lambda[].\lambda y.(d\ y)\ n)$

4.2. Notations for Montague grammars

A Montague grammar is made of rules. Each rule has a syntactic part, and a semantic part. The syntactic part is a generative rule (e.g., $S \to \mathbf{there\ is}\ NP$), where grammatical words are in bold, and syntagms are in italic. The semantic part is a λ-expression

between curly brackets (e.g., { *np* λx.**true** }), where constants are in bold, RDF terms are in teletype, and variables are in italic. For each syntagm (e.g., *NP*) in the right-hand side of the generative rule, its semantics is denoted by a variable whose name is a lowercase prefix of the syntagm name (e.g., *np*). Indices are used to disambiguate between several occurences of a same syntagm (e.g., np_1, np_2). For brevity, alternative words or syntagms are combined with a slash (e.g., **is**/**are**), instead of writing several rules. Each syntagm is associated to a single semantic type, which is specified in the first rule defining it (e.g., $S : s$). A default semantic can be defined at the same time (e.g., $Rel : p1$ (default : λx.**true**)), and allows to make the syntagm optional in right-hand sides of rules (e.g., *Rel*?). In trivial rules like $X \rightarrow Y$ { *y* }, the semantic part may be left implicit. To support modularity and avoid duplication of rules, we use higher-order syntagms, i.e., syntagms parametrized by other syntagms. There are two parametrization modes: $X(Y)$ is used when the semantics of Y is incorporated into the semantics of X, and $X\{Y\}$ is used when the semantics of Y is not part of X, and is returned apart. The latter is useful to parse non-contiguous constructs (e.g., "`between` ... `and` ..."). A beauty of Montague grammars is that they can be directly encoded in functional programming languages because they are based on lambda-calculus. Therefore, the grammar detailed in the following sections constitutes an executable specification of SQUALL's semantics.

4.3. Lexical units and default lexicon

Table 3 lists the different lexical units that are used in SQUALL's grammar, along with their semantic type. That list could be different for other natural languages than English, although there would probably be a large overlap. An entity can be any of a proper noun, a literal, a number, and a label. A description (type $p1$) can be any of a common noun, an adjective, an intransitive verb, and an imperative verb. A relation can be any of a relation noun, a transitive adjective, and a transitive verb. A statement modifier can be any of a preposition, and a superlative. Functions and operators return a description of their result given a list of entities (arguments). Aggregators return a description of their result (the aggregated value) given a description of what has to be aggregated ($p1 \rightarrow p1$). However, when there are dimensions (e.g., a `GROUP BY` clause), the description of what has to be aggregated depends on the values taken on each dimension ($[e]$), hence the type $([e] \rightarrow p1) \rightarrow p1$ for aggregators. This type is a complex analogue of the type of quantifiers ($p1 \rightarrow s$).

Because the focus of this paper is about the SQUALL language and its correspondence with SPARQL, we do not elaborate much about the lexical analysis, i.e., about the translation from concrete word forms to a Semantic Web vocabulary. There is active research on this task (see Section 8), and integrating their results is left to future work. The current implementation of SQUALL has a default definition of lexical units based only on URIs, and SPARQL built-ins (see below), but the way it is coded makes it relatively easy to extend those definitions for a particular vocabulary. Just to give an exemple, assuming a data property `ex:color`, the adjective "`green`" can be defined by the following rule:

$$Adj1 \rightarrow \textbf{green} \quad \{ \lambda x.(\textbf{triple}\ x\ \texttt{ex:color}\ \texttt{"green"}) \}$$

This rule says that a "green" thing is a thing that has color `"green"`.

By default, all URIs can be used as a proper noun (e.g., `res:France`); class URIs can be used as common nouns (e.g., `:Person`); property URIs can be used as relation

syntagm	type	name (examples)
ProperNoun	e	proper noun (John, France)
Literal	e	literal value (2013, "Hello")
Number	e	number (3, 0.5)
Label	e	label (X, Y2, ?foo)
Noun1	$p1$	common noun (man, country)
Noun2	$p2$	relation noun (mother, birth date)
Adj1	$p1$	adjective (green, French)
Adj2	$p2$	transitive adjective (knowing, known by)
Verb1	$p1$	intransitive verb (works, drink)
Verb2	$p2$	transitive verb (knows, wrote)
VerbImp	$p1$	imperative verb (load, print)
QueryImp	$p1$	imperative question prefix (return, give me)
Func	$[e] \to p1$	function (sqrt, pgcd)
Nulop	$p1$	nullary operator (now, a random number)
Unop	$e \to p1$	unary prefix operator (-)
Mulop	$e \to e \to p1$	$\times$-priority infix operator (*, /)
Addop	$e \to e \to p1$	$+$-priority infix operator (+, -)
NounAggreg	$([e] \to p1) \to p1$	aggregation noun (count, sum)
AdjAggreg	$([e] \to p1) \to p1$	aggregation adjective (total, average)
Prep	$e \to m_s$	preposition (to, from)
PrepNoun{*Det*}	$e \to m_s$	preposition (in {a} graph, from {the} service)
AdjSuper	$e \to m_s$	superlative adjective (10 greatest, 2nd)
DetSuper	$e \to m_s$	superlative determiner (the most, the 2nd most)

Table 3: List of the lexical units along with their semantic type, name, and concrete examples.

nouns (e.g., `:author`) and transitive verbs (e.g., `:livesIn`), and function URIs can be used as functions. Absolute URIs, relative URIs, and qualified names are written like in SPARQL. In the default namespace, bare qualified names can also be used, like in notation N3 (e.g., `author` instead of `:author`). Qualified names allow for relatively natural sentences, as shown in examples in this paper and on the Web page.

$$\begin{array}{l} \textit{ProperNoun} \to \textit{URI} \;\; \{\; \textit{uri} \;\} \\ \textit{Noun1} \to \textit{URI} \;\; \{\; \lambda x.(\textbf{triple}\ x\ \texttt{rdf:type}\ \textit{uri}) \;\} \\ \textit{Noun2}\,/\,\textit{Verb2} \to \textit{URI} \;\; \{\; \lambda x.\lambda y.(\textbf{triple}\ x\ \textit{uri}\ y) \;\} \\ \textit{Func} \to \textit{URI} \;\; \{\; \lambda lx.\lambda y.(\textbf{func}\ \textit{uri}\ lx\ y) \;\} \end{array}$$

In SPARQL triple patterns, a variable can be used in place of a class or a property. To support this feature, a triple `?x rdf:type ?c` is expressed as "X belongs to C"; and a triple `?x ?p ?y` is expressed as "P relates X to Y". "belong(s)" is a predefined intransitive verb, and "relate(s)" is a predefined transitive verb. They both rely on the predefined preposition "to" to specify the object of the triple (see Section 4.4.6).

$$\begin{array}{l} \textit{Verb1} \to \textbf{belong(s)} \;\; \{\; \lambda x.(\textbf{forarg}\ \texttt{to}\ \lambda c.(\textbf{triple}\ x\ \texttt{rdf:type}\ c)) \;\} \\ \textit{Verb2} \to \textbf{relate(s)} \;\; \{\; \lambda p.\lambda x.(\textbf{forarg}\ \texttt{to}\ \lambda y.(\textbf{triple}\ x\ p\ y)) \;\} \end{array}$$

SPARQL built-ins can be used under various forms, according to their semantic type. Unary predicates can be used as common nouns (e.g., "literal" for `isLiteral(.)`), adjectives (e.g., "numeric" for `isNumeric(.)`), and intransitive verbs. Binary predicates can be used as relation noun (e.g., "prefix" for `strStarts(.,.)`), transitive adjective (e.g., "matching" for `REGEX(.,.)`), and transitive verbs (e.g., "contain(s)" for `contains(.,.)`). All SPARQL functions can be used as SQUALL functions. Unary functions can also be used like binary predicates (e.g., "length" for `strlen(.)`), and nullary functions can be used as nullary operators (e.g., "a random number" for `RAND()`). Aggregations can be used as aggregation nouns and adjectives (e.g., the adjective "total" for `SUM`). SPARQL commands like `DESCRIBE` and `LOAD` are expressed as imperative verbs (resp., "describe", "load"). The SPARQL constructs `GRAPH` and `SERVICE` are expressed by the noun prepositions "in *Det* graph" and "from *Det* service". Indeed, the latter constructs modify a statement given an entity (either a named graph or a service). The other default prepositions are "into" and "to" to be used in the commands `LOAD` and `ADD`/`MOVE`/`COPY`. The SPARQL solution modifiers combine the ordering of solutions (`ORDER BY`), and the selection of a range of solutions (`LIMIT` and `OFFSET`). They are expressed as superlative adjectives and determiners. For example, the adjective highest translates to the SPARQL modifier `ORDER BY DESC(`x`) LIMIT 1`, where x is the entity passed to highest. We finally give a few representative examples about defining SPARQL built-ins as lexical units.

Noun1 $\rightarrow$ **literal** { $\lambda x.$(**pred** `isLiteral` $[x]$) }
AdjAggreg $\rightarrow$ **total** { $\lambda d.\lambda x.$(**aggreg** `SUM` d x) }
VerbImp $\rightarrow$ **load** { $\lambda y.$(**forarg** `into` $\lambda z.$(**command** `LOAD` $[y; z]$)) }
AdjSuper $\rightarrow$ **highest** { $\lambda x.\lambda s.$(**modif** "`ORDER BY DESC(`x`) LIMIT 1`" s) }
PrepNoun{*Det*} $\rightarrow$ **in** *Det* **graph** { $\lambda x.\lambda s.$(**context** `GRAPH` x s) }

4.4. Syntactic rules

In this section, we describe the syntactic rules of SQUALL in a modular fashion, where each module corresponds either to some SPARQL feature, or to some natural language feature, and often to a combination of both. This has the benefits to split the whole grammar (about 120 rules) in smaller parts, and to emphasize the relationships between SQUALL and SPARQL. This is also a way to assess the coverage of SPARQL by SQUALL (see Section 6 for a more rigorous evaluation of expressiveness). The reader should feel free to skip some modules, or to ignore the formal grammars on first reading.

4.4.1. Triples as sentences

A triple (s p o) can be seen as a basic sentence. The tradition in linguistics [26] is to analyse s and o as noun phrases (*NP*), p o as a verb phrase (*VP*), and the whole triple as a sentence (S). Here, noun phrases are either proper nouns or literal values, but they are given the type $s1$ to prepare for the use of quantifiers in Section 4.4.2. Verb phrases are based either on an intransitive verb (*Verb1*) followed by an optional complement phrase (*CP*), or on a transitive verb (*Verb2*) followed by an object phrase (*OP*), which is a noun phrase followed by an optional complement phrase. Complements are defined in Section 4.4.6, and are assumed empty at this point.

S : s
  $\rightarrow NP\ VP$ { $np\ vp$ }
NP : $s1$
  $\rightarrow Term$ { $\lambda d.(d\ term)$ }
$Term$: e
  $\rightarrow ProperNoun$
  $\rightarrow Literal$
VP : $p1$
  $\rightarrow VPdo$
$VPdo$: $p1$
  $\rightarrow Verb1\ CP?$ { $\lambda x.(cp\ (vp\ x))$ }
  $\rightarrow Verb2\ OP$ { $\lambda x.(op\ \lambda y.(verb2\ x\ y))$ }
OP : $s1$
  $\rightarrow NP\ CP?$ { $\lambda d.(np\ \lambda y.(cp\ (d\ y)))$ }
CP : m_s (default : $\lambda s.s$)

4.4.2. Quantifiers as determiners

Quantifiers are commonplace in natural languages in the form of determiners, whereas they are notoriously difficult to express in SPARQL or SQL [27]. Determiners behave as binary quantifiers (type $s2$), but unary quantifiers (type $s1$) can be cast as binary quantifiers by conjuncting the two descriptions. The definite article **the** has its own semantic primitive because it is interpreted either as the existential quantifier or as the universal quantifier depending on its syntactic context (see Section 4.6). A noun phrase can be formed by a determiner followed by a noun group (*NG1*). A binary noun phrase (*NP2*) is a noun phrase abstracted over an entity, and is made of a relation noun group (*NG2*). Noun groups are based on a noun (a relational noun for *NG2*), and may be modified by an adjective, an apposition (*App*), and a relative clause (*Rel*) (to be defined in the following sections). The keywords **for** and **there** introduce global quantifiers, in a style close to mathematical logic.

The grammar rules are defined so that the scope of quantifiers are leftmost-outermost, and are restricted to the scope of the related verb. Therefore, "every man love-s some woman" means there is possibly a different woman for each man; while "there is a woman that every man love-s" means there is a single woman that is loved.

Det : $s2$
  $\rightarrow$ $Det1$ { $\lambda d_1.\lambda d_2.(det1\ \lambda x.(\textbf{and}\ (d_1\ x)\ (d_2\ x)))$ }
  $\rightarrow$ **every/all** { $\lambda d_1.\lambda d_2.(\textbf{forall}\ d_1\ d_2)$ }
  $\rightarrow$ **the** { $\lambda d_1.\lambda d_2.(\textbf{the}\ d_1\ d_2)$ }
$Det1$: $s1$
  $\rightarrow$ **a/an** { $\lambda d.(\textbf{exists}\ d)$ }
  $\rightarrow$ **some** { $\lambda d.(\textbf{exists}\ d)$ }
  $\rightarrow$ **no** { $\lambda d.(\textbf{not}\ (\textbf{exists}\ d))$ }
$NP \rightarrow Det\ NG1$ { $\lambda d.(det\ ng1\ d)$ }
  $\rightarrow$ $NP2$ **of** NP { $\lambda d.(np\ \lambda x.(np2\ x\ d))$ }
$NP2$: $e \rightarrow s1$
  $\rightarrow$ $Det\ NG2$ { $\lambda x.\lambda d.(det\ \lambda y.(ng2\ x\ y)\ d)$ }
$NG1$: $p1$
  $\rightarrow$ $Adj1?\ Noun1\ App?\ Rel?$ { $\lambda x.(\textbf{and}\ (app\ x)\ (noun1\ x)\ (adj1\ x)\ (rel\ x))$ }
$NG2$: $p2$
  $\rightarrow$ $Adj1?\ Noun2\ App?$ { $\lambda x.\lambda y.(\textbf{and}\ (app\ y)\ (noun2\ x\ y)\ (adj1\ y))$ }
S $\rightarrow$ **for** NP, S { $np\ \lambda x.s$ }
  $\rightarrow$ **there is/are/was/were** NP { $np\ \lambda x.\textbf{true}$ }

4.4.3. Labels as appositions and anaphoras

Labels are used to cover the phenomenon of anaphoras in natural languages. They are introduced as appositions into noun groups, and referenced as terms. References are resolved at the semantic level, taking into account the scope of variables to check their validity, and to resolve ambiguities (see Section 4.7). Labels are the analogues of variables in SPARQL queries, but are much less often necessary in the syntax of a natural language.

App : $p1$ (default : $\lambda x.\textbf{true}$)
  $\rightarrow$ $Label$ { $\lambda x.(\textbf{label}\ x\ label)$ }
$Term \rightarrow Label$ { **ref** $label$ }

4.4.4. Relative clauses

Relative clauses modify the nouns, and semantically, are statements abstracted over an entity. Therefore, they can be derived from the syntax and semantics of sentences, replacing a noun phrase by a keyword among **that/which/who/whom**, and moving it at the beginning of the relative clause. Relative clauses can also be similar to a verb phrase, only using adjectives (including participles) instead of verbs.

Rel : $p1$ (default : $\lambda x.\mathbf{true}$)
→ **that/which/who** VP { $\lambda x.(vp\ x)$ }
→ **that/which/whom** $NP\ Verb2\ CP?$ { $\lambda y.(np\ \lambda x.(cp\ (verb2\ x\ y)))$ }
→ $NP2$ **of which** VP { $\lambda x.(np2\ x\ vp)$ }
→ **whose** $NG2\ VP$ { $\lambda x.(\mathbf{exists}\ \lambda y.(\mathbf{and}\ (ng2\ x\ y)\ (vp\ y)))$ }
→ $Adj1\ CP?$ { $\lambda x.(cp\ (adj1\ x))$ }
→ $Adj2\ OP$ { $\lambda x.(op\ \lambda y.(adj2\ x\ y))$ }
→ **such that** S { $\lambda x.s$ }

4.4.5. Auxiliary verbs

A particular syntax and semantics is associated to the auxiliary verbs "to be" and "to have", and verb phrases take three forms accordingly. Auxiliaries can be negated, and they are given the semantics of a sentence modifier, either negation or identity. The verb "to be" can be followed by nothing (keyword **there**), a relative clause (mostly adjective forms), and a *copula noun phrase*. Copula noun phrases (NPC and $NPC2$) have a syntax similar to noun phrases (NP and $NP2$), but a different semantics (type $p1$ instead of $s1$). They are parametrized by a semantic-less determiner syntagm, here an article. Copula noun phrases are also used in queries (Section 4.4.7) and aggregations (Section 4.4.12).

The verb "to have" can be followed by a relation noun (playing the role of a transitive verb) and an object phrase (e.g., "John_Smith has spouse Mary_Well"), or by a relational noun group and an optional relative clause (e.g., "John_Smith has every child that is a doctor"). The prepositions "with/without" play the same role as "to have", but for relative clauses instead of verb phrases (e.g., "every man with a child").

$Aux(Root)$: m_s
→ $Root$ { $\lambda s.s$ }
→ $Root$ **not** { $\lambda s.(\mathbf{not}\ s)$ }
→ $Root$**n't** { $\lambda s.(\mathbf{not}\ s)$ }
$VP \rightarrow Aux$(**does/do/did**) $VPdo$ { $\lambda x.(aux\ (vp\ x))$ }
→ Aux(**is/are/was/were**) $VPbe$ { $\lambda x.(aux\ (vp\ x))$ }
→ Aux(**has/have/had**) $VPhave$ { $\lambda x.(aux\ (vp\ x))$ }

$VPbe$: $p1$
→ **there** { $\lambda x.\mathbf{true}$ }
→ Rel { $\lambda x.(rel\ x)$ }
→ $NPC(\mathbf{a/an/the})$ { $\lambda x.(npc\ x)$ }
$NPC(Det)$: $p1$
→ $Term$ { $\lambda x.(\mathbf{eq}\ x\ term)$ }
→ $Det\ NG1$ { $\lambda x.(ng1\ x)$ }
→ $NPC2(Det)$ **of** NP { $\lambda y.(np\ \lambda x.(npc2\ x\ y))$ }
$NPC2(Det)$: $e \rightarrow p1$
→ $Det\ NG2$ { $\lambda x.\lambda y.(ng2\ x\ y)$ }
$VPhave$: $p1$
→ $Noun2\ OP$ { $\lambda x.(op\ \lambda y.(noun2\ x\ y))$ }
→ $NP2\ Rel?$ { $\lambda x.(np2\ x\ rel)$ }
$Rel \rightarrow$ **with** $VPhave$ { $\lambda x.(vp\ x)$ }
→ **without** $VPhave$ { $\lambda x.(\mathbf{not}\ (vp\ x))$ }

4.4.6. Prepositional phrases

Prepositional phrases (PP) are used to handle verbs that expect arguments in addition to subject and object (e.g., "John belongs to the `rdfs:Class` that has `rdfs:label` "human""), and also to express *truth-contexts* such as the named graphs and distant services of SPARQL (e.g., "from the graph whose author is John Paris is the capital of France"). A prepositional phrase is made of a preposition and a noun phrase. The noun phrase is used like subjects and objects, and the preposition modifies the statement in function of the variable z introduced by the noun phrase (e.g., passing the extra-argument to the verb, see Section 4.6). In the case of a noun preposition (*PrepNoun*), the noun of the noun phrase helps to determine the semantics of the preposition (e.g., "from {the} graph"). A propositional phrase can occur at any position in a sentence: at the beginning or at the end of the sentence, before or after the verb. The nesting of S, VP, OP, and CP ensures the leftmost-outermost rule for the scope of quantifiers, and the free position of complements allows for their flexible ordering. Finally, as a relative clause is a sentence with a displaced noun phrase, a form of relative clause is generated for each form of prepositional phrase (e.g., "to which John belongs", "in which graph Paris is the capital of France").

$S \rightarrow PP\ S$ { $pp\ s$ }
$VP \rightarrow PP\ VP$ { $\lambda x.(pp\ (vp\ x))$ }
$OP \rightarrow PP\ OP$ { $\lambda d.(pp\ (op\ d))$ }
$CP \rightarrow PP\ CP?$ { $\lambda s.(pp\ (cp\ s))$ }
PP : m_s
→ $Prep\ NP$ { $\lambda s.(np\ \lambda z.(prep\ z\ s))$ }
→ $PrepNoun\{Det\}\ App?\ Rel?$ { $\lambda s.(det\ \lambda z.(\mathbf{and}\ (app\ z)\ (rel\ z))\ \lambda z.(prep\ z\ s))$ }
$Rel \rightarrow Prep$ **which** S { $\lambda x.(prep\ x\ s)$ }
→ $PrepNoun\{\mathbf{which}\}\ S$ { $\lambda x.(prep\ x\ s)$ }

4.4.7. Updates and queries

Updates (U) are either a sentence ended by a full-stop (e.g., "John knows Mary."), or an imperative verb followed by an object phrase and a full-stop (e.g., "Clear all named graphs."), or a sequence of updates. Queries (Q) can be closed or open. Closed queries can be formed by ending a declarative sentence by a question mark (e.g., "John knows Mary?"), possibly prefixing it with the keyword **whether**. The semantic primitive **ask** modifies the statement into an interrogative one. Open queries are expressed either with imperative locutions (e.g., return, give me) followed by an object phrase that describes what has to be returned (e.g., "Give me all film-s whose director is Tim_Burton."), or with the interrogative words **which/what/who/whom**. The semantic primitive **return** specifies the variable to be projected in results. The semantic primitives **select** and **select_where** quantify over a variable to be projected (the `SELECT` clause in SPARQL). A multi-dimensional query is a sentence with several occurences of wh-words (e.g., "Which person is the director of which film?").

$$
\begin{array}{ll}
U & : s \\
 & \quad \rightarrow S\ .\ \{\ s\ \} \\
 & \quad \rightarrow \mathit{VerbImp}\ \mathit{OP}\ .\ \{\ \mathit{op}\ \lambda x.(\mathit{verbimp}\ x)\ \} \\
 & \quad \rightarrow U\ U\ \ \{\ \textbf{and}\ u_1\ u_2\ \} \\
Q & : s \\
 & \quad \rightarrow S\ ?\ \ \{\ \textbf{ask}\ s\ \} \\
 & \quad \rightarrow \textbf{whether}\ S\ ?\ \ \{\ \textbf{ask}\ s\ \} \\
 & \quad \rightarrow \mathit{QueryImp}\ \mathit{OP}\ .\ \{\ \mathit{op}\ \lambda x.(\textbf{return}\ x)\ \} \\
\mathit{NP} & \rightarrow \textbf{what/who/whom}\ \ \{\ \lambda d.(\textbf{select}\ d)\ \} \\
\mathit{Det} & \rightarrow \textbf{which}\ \ \{\ \lambda d_1.\lambda d_2.(\textbf{select\_where}\ d_1\ d_2)\ \}
\end{array}
$$

To allow for a more natural verbalization of closed questions, the grammar has been extended to support subject-auxiliary inversion (e.g., "Does John know Mary?", "Is John the father of a doctor?"). Those grammar rules can be derived from previous rules, and we do not detail them here.

4.4.8. Solution modifiers as superlatives

SPARQL solution modifiers include ordering results w.r.t. a variable or expression (`ORDER BY`), and selecting a sub-range of solutions (`LIMIT` and `OFFSET`). In SQUALL, they are expressed as superlative adjectives (e.g., highest like in "Which mountain has the highest elevation?"), and they modify a statement in function of the entity the adjective applies to. Those adjectives are special in that they must occur first and only once in noun groups. See Section 4.6 for a deeper explanation of their semantics.

$$
\begin{array}{l}
\mathit{NG1} \rightarrow \mathit{AdjSuper}\ \mathit{NG1}\ \ \{\ \lambda x.(\mathit{adj}\ x\ (\mathit{ng1}\ x))\ \} \\
\mathit{NG2} \rightarrow \mathit{AdjSuper}\ \mathit{NG2}\ \ \{\ \lambda x.\lambda y.(\mathit{adj}\ y\ (\mathit{ng2}\ x\ y))\ \}
\end{array}
$$

4.4.9. Relational algebra as coordinations

There is a one-to-one mapping between operations from the relational algebra of SPARQL (join, union, optional, negation), coordinations (resp., and, or, not, maybe), and

Boolean operators that we use in our intermediate representation. We define Boolean expressions ($Bool(X)$) in a generic way by parametrizing them with atomic expressions X. In this way, they can be defined once, and applied to many syntagms. For example, the grammar rule $S \rightarrow Bool(S)$ means that sentences can be coordinated (e.g., "John own-s a book X and Mary own-s X."). In SQUALL, all the following syntagms can be fully coordinated: *S*, *NP*, *NP2*, *Rel*, *VP*, *OP*, *CP*, *PP*, *Prep*, *Noun1*, *Noun2*, *Adj1*, *Adj2*, *Verb1*, and *Verb2*. This helps to make sentences more concise by factorizing common phrases (e.g., "Do John and Mary own or rent every book whose topic is "CNL"?"). Parentheses can be used to override the usual priorities between Boolean operators, but they are hardly ever necessary in practice. Because Boolean expressions are applied to syntagms having different types α, the Boolean semantic primitives must be extended from statements to those types. For example, $(\mathbf{and}_{p1}\ d_1\ d_2) = \lambda x.(\mathbf{and}\ (d_1\ x)\ (d_2\ x))$, and $(\mathbf{and}_{p2}\ r_1\ r_2) = \lambda x.\lambda y.(\mathbf{and}\ (r_1\ x\ y)\ (r_2\ x\ y))$. More generally, every additional argument is passed down to sub-expressions.

$$
\begin{array}{l}
Bool(X:\alpha)\ :\ \alpha \\
\quad \rightarrow Bool(X)\ \mathbf{or}\ Bool(X)\ \ \{\ \mathbf{or}_\alpha\ bool_1\ bool_2\ \} \\
\quad \rightarrow Bool(X)\ \mathbf{and}\ Bool(X)\ \ \{\ \mathbf{and}_\alpha\ bool_1\ bool_2\ \} \\
\quad \rightarrow \mathbf{maybe}\ Bool(X)\ \ \{\ \mathbf{option}_\alpha\ bool\ \} \\
\quad \rightarrow \mathbf{not}\ Bool(X)\ \ \{\ \mathbf{not}_\alpha\ bool\ \} \\
\quad \rightarrow (\ Bool(X)\ )\ \ \{\ bool\ \} \\
\quad \rightarrow X\ \ \{\ x\ \}
\end{array}
$$

4.4.10. Conditionals

Conditionals are introduced by the keywords **if-then**, and are semantically defined with implications (e.g., "If John has a spouse X then X is a woman."). The coordination **where** is interpreted as a reverse implication in updates, and as a conjunction in queries (e.g., "Which film F has a director D where D is an actor of F?").

$$
\begin{array}{rl}
S & \rightarrow \mathbf{if}\ S\ \mathbf{then}\ S\ \ \{\ \mathbf{implies}\ s_1\ s_2\ \} \\
& \rightarrow S\ \mathbf{where}\ S\ \ \{\ \mathbf{where}\ s_1\ s_2\ \}
\end{array}
$$

4.4.11. Expressions

In SPARQL, expressions are found in filters, in the `SELECT` clause, and in solution modifiers (`ORDER BY`, `HAVING`). They combine operators, functions, literals, and variables. In SQUALL, expressions ($Expr(X)$) are defined in a generic way by parametrizing them with a syntagm for atomic expressions X, like for Boolean expressions. This allows to use expressions in all kinds of noun phrases (see below). Syntactically, expressions are classically made of atomic expressions (X), terms (include literals and references), additive and multiplicative infix operators, prefix unary operators, nullary operators (e.g., the SPARQL function `NOW`), functions, and parentheses. Semantically, an expression has type $k_\alpha = (e \rightarrow \alpha) \rightarrow \alpha$, i.e., is defined in Continuation Passing Style (CPS). In short, each expression takes a *continuation* as an argument, i.e., a function that says what to do with the result of the evaluation of the expression. The definition of k_α makes expressions analogue to unary quantifiers, and hence to noun phrases. Indeed, the SQUALL sentence

"The height * the width of some rectangle R is 100" is equivalent to "For the height H of some rectangle R, for the width W of R, H * W is 100". The definition of the semantics of expressions rely on a combinator *apply* that applies a continuation k to the result of an operation or function.

$$apply : (e \to \alpha) \to p1 \to \alpha = \lambda k.\lambda d.\lambda z^*.(\textbf{the}\ \lambda y.(d\ y)\ \lambda y.(k\ y\ z^*))$$

The number of extra arguments z^* is equal to the arity of type α. The quantifier **the** ensures that expressions will be put in the `WHERE`-clause of updates.

$$\begin{array}{l}
Expr(X : k_\alpha)\ :\ k_\alpha \\
\quad \to Expr(X)\ Addop\ Expr(X) \\
\qquad \{\ \lambda k.(expr_1\ \lambda x_1.(expr_2\ \lambda x_2.(apply\ k\ (addop\ x_1\ x_2))))\ \} \\
\quad \to Expr(X)\ Mulop\ Expr(X) \\
\qquad \{\ \lambda k.(expr_1\ \lambda x_1.(expr_2\ \lambda x_2.(apply\ k\ (mulop\ x_1\ x_2))))\ \} \\
\quad \to Unop\ Expr(X)\ \ \{\ \lambda k.(expr\ \lambda x.(apply\ k\ (unop\ x)))\ \} \\
\quad \to Nulop\ \ \{\ \lambda k.(apply\ k\ nulop)\ \} \\
\quad \to Func(Expr(X),...,Expr(X)) \\
\qquad \{\ \lambda k.(expr_1\ \lambda x_1.(...\ expr_n\ \lambda x_n.(apply\ k\ (func\ [x_1;\ldots;x_n]))\ ...))\ \} \\
\quad \to (\ Expr(X)\ )\ \ \{\ expr\ \} \\
\quad \to Term\ \ \{\ \lambda k.(k\ term)\ \} \\
\quad \to X\ \ \{\ x\ \}
\end{array}$$

Expressions apply to all kinds of noun phrases: *NP*, *NP2*, *NPC*, and *NPC2*. Because expressions and the different noun phrases have different types, noun phrases need to be wrapped as atomic expressions, and whole expressions need to be unwrapped back as noun phrases.

$$\begin{array}{l}
NP \to Expr(NP\ \ \{\ \lambda k.\lambda d.(np\ \lambda x.(k\ x\ d))\ \}) \\
\ \{\ expr\ \lambda v.\lambda d.(d\ v)\ \} \\
NP2 \to Expr(NP2\ \ \{\ \lambda k.\lambda x.\lambda d.(np2\ x\ \lambda y.(k\ y\ x\ d))\ \}) \\
\ \{\ expr\ \lambda v.\lambda x.\lambda d.(d\ v)\ \} \\
NPC(Det) \to Expr(NPC(Det)\ \ \{\ \lambda k.\lambda x.(\textbf{exists}\ \lambda y.(\textbf{and}\ (npc\ y)\ (k\ y)))\ \}) \\
\ \{\ \lambda x.(expr\ \lambda v.(\textbf{eq}\ x\ v))\ \} \\
NPC2(Det) \to Expr(NPC2(Det)\ \ \{\ \lambda k.\lambda x.(\textbf{exists}\ \lambda y.(\textbf{and}\ (npc2\ x\ y)\ (k\ y)))\ \}) \\
\ \{\ \lambda x.\lambda y.(expr\ \lambda v.(\textbf{eq}\ y\ v)\ x)\ \}
\end{array}$$

4.4.12. Aggregations and grouping

Aggregations are another way to compute values, beside expressions. However, they behave differently w.r.t. syntax and semantics because an aggregator applies to a description (i.e., a set of values). Moreover, one or several dimensions can be specified (`GROUP BY` clauses in SPARQL) so that an aggregation may return a set of aggregated values (type $p1$ instead of e). For instance, "the average size of the person-s per age" returns an average size for each instanciated age. Hence the type $g = ([e] \to p1) \to p1$ of aggregators, where $[e]$ represents the list of dimension variables, the first $p1$ represents the

set of values to be aggregated, and the second $p1$ represents the set of aggregated values. Aggregators can be seen as quantifiers, which quantify existentially over each dimension. In fact, an adjective aggregator can be used as a determiner, where the aggregated values are constrained by an optional relative clause (e.g., "John has a number greater than 3 of paper-s"). Alternately, aggregators can be the head of noun groups so that aggregated values can be used as subjects and objects of verbs (e.g., "Is every number of publication-s per author lesser than 100?"), and so that aggregations can be nested (e.g., "What is the average number of publication-s per author?"). Each dimension is defined as a relation from the *facts* (e.g., "publications" in previous examples) to the dimension variable z. A list of dimensions is a relation from the facts to a list lz of dimension variables.

$$
\begin{array}{l}
\mathit{Det1} \rightarrow \textbf{a number } \mathit{App}?\ \mathit{Rel}?\ \textbf{of} \\
\qquad \{\ \lambda d.(\textbf{the } \lambda x.(\textbf{count } d\ x)\ \lambda x.(\textbf{and } (\mathit{app}\ x)\ (\mathit{rel}\ x)))\ \} \\
\quad \rightarrow \textbf{a/an } \mathit{AdjAggreg}\ \mathit{App}?\ \mathit{Rel}? \\
\qquad \{\ \lambda d.(\textbf{the } \lambda x.(\mathit{aggreg}\ \lambda lz.\lambda y.(d\ y)\ x)\ \lambda x.(\textbf{and } (\mathit{app}\ x)\ (\mathit{rel}\ x)))\ \} \\
\mathit{NG1} \rightarrow \mathit{NounAggreg}\ \mathit{App}?\ \textbf{of } \mathit{NPC}(\textbf{the})\ \mathit{Dims}? \\
\qquad \{\ \lambda v.(\textbf{and } (\mathit{app}\ v)\ (\mathit{aggreg}\ \lambda lz.\lambda y.(\textbf{and } (\mathit{npc}\ y)\ (\mathit{dims}\ y\ lz))\ v))\ \} \\
\quad \rightarrow \mathit{AdjAggreg}\ \mathit{App}?\ \mathit{NG1}\ \mathit{Dims}? \\
\qquad \{\ \lambda v.(\textbf{and } (\mathit{app}\ v)\ (\mathit{aggreg}\ \lambda lz.\lambda y.(\textbf{and } (\mathit{ng1}\ y)\ (\mathit{dims}\ y\ lz))\ v))\ \} \\
\quad \rightarrow \mathit{AdjAggreg}\ \mathit{App}?\ \mathit{NG2}\ \textbf{of } \mathit{NPC}(\textbf{the})\ \mathit{Dims}? \\
\qquad \{\ \lambda v.(\textbf{and } (\mathit{app}\ v)\ (\mathit{aggreg}\ \lambda lz.\lambda y.(\textbf{exists } \lambda x.(\\
\qquad\qquad \textbf{and } (\mathit{npc}\ x)\ (\mathit{ng2}\ x\ y)\ (\mathit{dims}\ x\ lz)))\ v))\ \} \\
\mathit{Dims}\ :\ e \rightarrow [e] \rightarrow s \quad (\text{default} : \lambda y.\lambda[].\textbf{true}) \\
\quad \rightarrow \mathit{Dim}\ \ \{\ \lambda y.\lambda[z].(\mathit{dim}\ y\ z)\ \} \\
\quad \rightarrow \mathit{Dim}\ \textbf{and } \mathit{Dims}\ \ \{\ \lambda y.\lambda(z :: lz).(\textbf{and } (\mathit{dim}\ y\ z)\ (\mathit{dims}\ y\ lz))\ \} \\
\mathit{Dim}\ :\ p2 \\
\quad \rightarrow \textbf{per } \mathit{NG2}\ \ \{\ \lambda y.\lambda z.(\mathit{ng2}\ y\ z)\ \} \\
\quad \rightarrow \textbf{per } \mathit{NPC}(\textbf{the})\ \ \{\ \lambda y.\lambda z.(\mathit{npc}\ z)\ \}
\end{array}
$$

4.4.13. Comparisons

Although comparators have type $p2$ and can therefore be verbalized as transitive verbs (e.g., is greater than), or as transitive adjectives (e.g., greater than), they deserve a special treatment because of the rich combinations offered by natural languages. We define below a number of constructs that do not increase SQUALL's expressivity, but that improves a lot SQUALL's naturalness and concision when expressing comparisons. We only detail here the "greater than" comparator (constant **gt**), but other comparators are straightforward to add.

Relatives are extended by using comparators as transitive adjectives, and by adding "between" as a ditransitive adjective. Their semantics is the same as for transitive adjectives.

$$
\begin{array}{l}
\mathit{Rel} \rightarrow \textbf{greater than } \mathit{OP}\ \ \{\ \lambda x.(\mathit{op}\ \lambda y.(\textbf{gt}\ x\ y))\ \} \\
\quad \rightarrow \textbf{between } \mathit{OP}\ \textbf{and } \mathit{OP}\ \ \{\ \lambda x.(\mathit{op}_1\ \lambda y_1.(\mathit{op}_2\ \lambda y_2.(\textbf{between}\ x\ y_1\ y_2)))\ \}
\end{array}
$$

Comparators are also used as determiners after the auxiliary verb "to have" to compare two properties of an entity (e.g., "Which rectangle has a greater height than width?"), or to compare a same property between two entities (e.g., "Which woman has a greater size than every man?").

$$
\begin{array}{l}
\mathit{DetComp}\{X, Y\} \ : \ s2 \\
\quad \rightarrow \textbf{a greater } X \textbf{ than } Y \\
\qquad \{\ \lambda d_1.\lambda d_2.(\textbf{exists } \lambda y_1.(\textbf{exists } \lambda y_2.(\textbf{and } (d_1\ y_1)\ (d_2\ y_2)\ (\textbf{gt } y_1\ y_2)))) \ \} \\
\mathit{VPhave} \\
\quad \rightarrow \mathit{DetComp}\{\mathit{Noun2}, \mathit{Noun2}\} \\
\qquad \{\ \lambda x.(\mathit{detcomp}\ \lambda y_1.(\mathit{noun2}_1\ x\ y_1)\ \lambda y_2.(\mathit{noun2}_2\ x\ y_2)) \ \} \\
\quad \rightarrow \mathit{DetComp}\{\mathit{Noun2}, \mathit{NP}\} \\
\qquad \{\ \lambda x_1.(\mathit{np}\ \lambda x_2.(\mathit{detcomp}\ \lambda y_1.(\mathit{noun2}\ x_1\ y_1)\ \lambda y_2.(\mathit{noun2}\ x_2\ y_2))) \ \}
\end{array}
$$

There are also determiners to compare numbers of things, where the values to be compared are the result of a `COUNT`-aggregation. The semantic statement (**count** $d\ n$) tells that n is the number of entities satisfying description d. It is used to define generalized determiners such as "more than 2" or "between 3 and 5", the interrogative determiner "how many", and superlative determiners such as "the most". From these, it becomes possible to express queries such: "How many person-s are the author of at least 10 publication-s?" or "Which person is the author of the most publication-s?".

$$
\begin{array}{l}
\mathit{Det1} \rightarrow \mathit{Number} \ \ \{\ \lambda d.(\textbf{count } d\ \mathit{number})\ \} \\
\quad \rightarrow \textbf{more than } \mathit{Number} \ \ \{\ \lambda d.(\textbf{exists } \lambda n.(\textbf{and } (\textbf{count } d\ n)\ (\textbf{gt } n\ \mathit{number})))\ \} \\
\quad \rightarrow \textbf{between } \mathit{Number} \textbf{ and } \mathit{Number} \\
\qquad \{\ \lambda d.(\textbf{exists } \lambda n.(\textbf{and } (\textbf{count } d\ n)\ (\textbf{between } n\ \mathit{number}_1\ \mathit{number}_2)))\ \} \\
\quad \rightarrow \textbf{how many} \ \ \{\ \lambda d.(\textbf{select } \lambda n.(\textbf{count } d\ n))\ \} \\
\quad \rightarrow \mathit{DetModif} \ \ \{\ \lambda d.(\textbf{exists } \lambda n.(\mathit{modif}\ n\ (\textbf{count } d\ n)))\ \}
\end{array}
$$

We also define binary determiners to compare two numbers of things, which lead to new forms of noun phrases:

- "Do more woman than man own a cat?"
- "Who has more daughter-s than son-s whose age is lesser than 18?"
- "Which woman X has more daughter-s than the mother of X?"

$$
\begin{array}{l}
\mathit{DetCompCount}\{X, Y\} \ : \ s2 \\
\quad \rightarrow \textbf{more } X \textbf{ than } Y \ \ \{\ \lambda d_1.\lambda d_2.(\textbf{exists } \lambda n_1.(\textbf{exists } \lambda n_2.(\\
\qquad \textbf{and } (\textbf{count } d_1\ n_1)\ (\textbf{count } d_2\ n_2)\ (\textbf{gt } n_1\ n_2)))) \ \} \\
\mathit{NP} \rightarrow \mathit{DetCompCount}\{\mathit{NG1}, \mathit{NG1}\} \ \ \{\ \lambda d.(\mathit{detcomp} \\
\quad \lambda x_1.(\textbf{and } (\mathit{ng1}_1\ x_1)\ (d\ x_1))\ \lambda x_2.(\textbf{and } (\mathit{ng1}_2\ x_2)\ (d\ x_2)))\ \} \\
\mathit{NP2} \rightarrow \mathit{DetCompCount}\{\mathit{NG2}, \mathit{NG2}\} \ \ \{\ \lambda x.\lambda d.(\mathit{detcomp} \\
\ \lambda y_1.(\textbf{and } (\mathit{ng2}_1\ x\ y_1)\ (d\ y_1))\ \lambda y_2.(\textbf{and } (\mathit{ng2}_2\ x\ y_2)\ (d\ y_2)))\ \} \\
\quad \rightarrow \mathit{DetCompCount}\{\mathit{NG2}, \mathit{NP}\} \ \ \{\ \lambda x_1.\lambda d.(\mathit{np}\ \lambda x_2.(\mathit{detcomp} \\
\ \lambda y_1.(\textbf{and } (\mathit{ng2}\ x_1\ y_1)\ (d\ y_1))\ \lambda y_2.(\textbf{and } (\mathit{ng2}\ x_2\ y_2)\ (d\ y_2))))\ \}
\end{array}
$$

4.4.14. Graph literals

In SPARQL, `CONSTRUCT`-queries return graphs, i.e., sets of triples. In order to support this kind of query in SQUALL, we allow the reification of statements as entities using the primitive **graphliteral** in the semantics, and the word **that** followed by a sentence in the syntax to introduce a subordinate clause. The reification is invalid if the statement is anything else than a conjunction of triples, and if any variable remains unbound. Like in notation N3, graph literals (called formulae in N3) can be used everywhere an entity can be used. However, when translating to SPARQL, they can only be returned as the query result (e.g., "For every person X whose age is greater or equal to 18, return that X is an adult.").

$$\mathit{Term} \rightarrow \textbf{that}\ S\ \ \{\ \textbf{graphliteral}\ s\ \}$$

4.4.15. Collections

RDF collections are single-chained lists based on properties `rdf:first` and `rdf:rest`, plus the empty list `rdf:nil`. They are useful to represent closed and ordered sets of entities, such as the authors of a document. Turtle and SPARQL provide syntactic sugar for fixed-length collections, but the representation of other graph patterns is extremely tedious. For example, the SPARQL pattern to access the last element of a collection is: `[ rdf:rest* [ rdf:first ?last ; rdf:rest rdf:nil ] ]`. We extend SQUALL's noun phrases so as to completely avoid the use of the RDF vocabulary about collections. We reuse the rich Prolog's syntax for list patterns [28], and extend it with the ellipsis ... to allow jumping over an arbitrary number of elements. List patterns are delimited by square brackets, list elements are separated by commas, and sublists are raised after a bar. An underscore _ stands for an arbitrary element (joker), and an ellipsis ... stands for an arbitrary sequence of elements. For example, "Which book has authorList [..., John_Smith]?" returns the books whose last author is John Smith, and "Whether [_, every member of Team1, ...] is the authorList of a book whose topic is AI?" returns whether every member of Team1 is the second author of a book whose topic is AI.

List noun phrases (*NPL*) are noun phrases that quantify over lists. To simplify the definition of their semantics, we rely on two combinators *cons* and *sublist* which build unary quantifiers (type $s1$), respectively, from a first and a rest, and from a sublist:

$\mathit{cons} = \lambda e.\lambda l.\lambda d.(\textbf{exists}\ \lambda x.(\textbf{and}\ (\textbf{triple}\ x\ \texttt{rdf:first}\ e)\ (\textbf{triple}\ x\ \texttt{rdf:rest}\ l)\ (d\ x)))$
$\mathit{sublist} = \lambda l.\lambda d.(\textbf{exists}\ \lambda x.(\textbf{and}\ (\textbf{triple}\ x\ \texttt{rdf:rest}*\ l)\ (d\ x)))$

$\mathit{Term} \rightarrow [\ ]\ \ \{\ \texttt{rdf:nil}\ \}$
$\mathit{NP} \rightarrow \_\ \ \{\ \lambda d.(\textbf{exists}\ d)\ \}$
$\quad \rightarrow [\ \mathit{NPL}\ ]\ \ \{\ \mathit{npl}\ \}$
$\mathit{NPL}\ :\ s1$
$\quad \rightarrow \mathit{NP},\ \mathit{NPL}\ \ \{\ \lambda d.(\mathit{np}\ \lambda e.(\mathit{npl}\ \lambda l.(\mathit{cons}\ e\ l\ d)))\ \}$
$\quad \rightarrow \mathit{NP}\ |\ \mathit{NP}\ \ \{\ \lambda d.(\mathit{np}_1\ \lambda e.(\mathit{np}_2\ \lambda l.(\mathit{cons}\ e\ l\ d)))\ \}$
$\quad \rightarrow \mathit{NP}\ \ \{\ \lambda d.(\mathit{np}\ \lambda e.(\mathit{cons}\ e\ \texttt{rdf:nil}\ d))\ \}$
$\quad \rightarrow ...,\ \mathit{NPL}\ \ \{\ \lambda d.(\mathit{npl}\ \lambda l.(\mathit{sublist}\ l\ d))\ \}$
$\quad \rightarrow ...\ |\ \mathit{NP}\ \ \{\ \lambda d.(\mathit{np}\ \lambda l.(\mathit{sublist}\ l\ d))\ \}$
$\quad \rightarrow ...\ \ \{\ \lambda d.(\mathit{sublist}\ \texttt{rdf:nil}\ d)\ \}$

4.5. Handling of syntactic ambiguities

The price for the natural and flexible syntax of SQUALL is ambiguity, i.e., the fact that some sentences can be parsed in different ways possibly leading to different semantics. In SQUALL, ambiguities are resolved by the following rules:

1. when forming a construct X from one or two constructs of same syntagm X (e.g., coordinating 2 *NP*s, modifying a sentence with a *PP*), algebraic operators have priority (in decreasing priority order: **not**, **maybe**, **and**, **or**, **where**) over sentence modifiers (*PP* as a prefix, and global quantifiers **for** *NP*), and right-associativity applies for binary coordinations;
2. "smaller" syntagms have priority over "larger" syntagms, i.e., in decreasing priority order: *Det*∗, *Rel*, *NG*∗, *NP*∗, *PP*, *CP*, *OP*, *VP*, *S*;
3. in case of remaining ambiguity between two phrases of the same syntagm, the shorter phrase is chosen.

Round brackets can be used in coordinations and expressions to escape those rules. Rule 2 implies that "a man or woman" is interpreted as "a (man or woman)" rather than "(a man) or woman", as *NG1* has priority over *NP*. Rule 3 implies that in "*A* know-s a researcher that *X* cite-s in graph *G*", the *PP* "in graph *G*" binds to the shorter *VP* "cite-s ..." rather than to the longer *VP* "know-s ...".

4.6. Semantic transformations of statements

Semantic transformations need to be applied to the intermediate representation produced through Montague grammars, because some primitives are contextual in nature, i.e., have non-local effects. This concerns the interrogative quantifiers (primitives **select** and **select_where**), prepositions (primitive **arg**), and modifiers (primitive **modif**). For example, the SQUALL sentence "Every member of Group1 belongs to which nationality?" (equivalent to "To which nationality does every member of Group1 belong?"), gets the following intermediate representation (some lexical units have not been expanded by their definition for clarity):

forall $\lambda x.$(*member* x :`Group1`)
  $\lambda x.$(**select_where** $\lambda z.$(*nationality* z) $\lambda z.$(**arg** `to` z (*belong* x)))

Two problems appear in this example. First, the primitive **select_where** should be out of the scope of the quantifier **forall**. Second, the sub-expression (*belong* x) should reduce to (**triple** x `rdf:type` z), but z is not an argument of *belong*.

The solution we have found to retain a compositional style (i.e., to avoid global variables), while maintaining a local treatment of each feature, is to define statements as *state monads* [29], i.e. as functions that take a state as a parameter, and that return a modified state in addition to the statement itself. The additional state enables to pass information downward, and upward in the semantic expression tree. The state is here made of extra-arguments added by prepositions, selectors added by interrogative quantifiers, and modifiers. Selectors are passed upward up to the root of the syntax tree. Modifiers are also passed upward, but are catched by the determiner of noun phrases (*NP*), to account for sentences such as "Does the person with the lowest age has a greater size than the person with the highest age?". Prepositions are passed downward with primitive **arg** and catched by verbs with primitive **forarg** (e.g., the preposition "to" is

catched by the verbs "belongs" and "relates"). Applying those principles to the above example, we get the fully reduced intermediate representation.

select $\lambda z.$(**and** (**triple** z `rdf:type` `:nationality`)
(**forall** $\lambda x.$(**triple** x `:member` `:Group1`) $\lambda x.$(**triple** x `rdf:type` z)))

Note that **select_where** has been transformed into **select**, and has now the outermost scope. Note also that primitives **arg** and **forarg** have been eliminated in the transformed intermediate representation. Similarly, primitive **the** (resp. **where**) is replaced by primitive **exists** (resp. **and**) in a query context (in the scope of **ask**, **select**, and the left argument of **forall** and **implies**); and is replaced by **forall** (resp. the inverse of **implies**) otherwise.

4.7. Semantic validation

Like in programming languages, SQUALL sentences may be syntactically correct, but semantically invalid. Examples of semantic errors in programming languages are undeclared variables, or type errors. In SQUALL, semantic errors can be:

- out-of-scope references: e.g. the reference X in "John has no child X and X is a doctor.";
- unbound variables: e.g. "Return 1 + some thing.", where there is no way to bind the second operand of the addition.

In order to detect semantic errors, and return them to users, statements are validated w.r.t. accessibility and boundedness. References are first resolved in function of defined labels, and accessible variables, which eliminates primitives **label** and **ref** from the intermediate representation. Accessibility validation consists in checking that every variable is used in the scope of its quantifier. Discourse Representation Theory (DRT) [30], and its combination with Montague grammars [31], is used to define those scopes, and plays an important role in the naturalness of SQUALL sentences. For example, this validation phase accepts the sentence "Some man X loves a woman and X is a doctor.", and rejects "Every man X loves a woman and X is a doctor.". Boundedness validation consists in checking that dataflow constraints are satisfied. For example, all arguments of predicates and functions must be bound before evaluating them, while triple patterns have no such constraints. This validation phase ensures that queries and updates can effectively be evaluated, i.e., are semantically well-defined. For example, the boundedness validation accepts "Which person has every child that is a doctor?", and rejects "Who has every child that is a doctor?" because the latter does not provide any means to bind the variable quantified by Who[8]. Boundedness validation is also used to replace equalities (**pred** $= [x; y]$) with the identity function when only one argument is bound (e.g., **func** `ID` $[x]$ y, if only x is bound).

5. Translation to SPARQL

The generation of a SPARQL query or update from the intermediate representation of semantics is much simpler than syntactic and semantic analysis because it mostly consists in mapping logical constructs to SPARQL constructs, which are at the same

[8] Unless a default class for people is specified, and used in the semantics of who.

level of abstraction. Note that the intermediate representation would make it easy to support another target RDF query language. As an illustration, the SPARQL translation of the SQUALL sentence "For which researcher-s X, in graph DBLP every publication whose author is X and whose :publication_year is greater than 2000 has at least 2 author-s?", from the introduction of Section 4, is as follows:

```
SELECT DISTINCT ?x1 WHERE {
    ?x1 a :researcher .
    FILTER NOT EXISTS {
        GRAPH :DBLP {
            ?x2 a :publication .
            ?x2 :author ?x1 .
            ?x2 :year ?x3 .
            FILTER (?x3 > 2000) . }
        FILTER NOT EXISTS {
            GRAPH :DBLP {
                { SELECT DISTINCT ?x2 (COUNT(?x5) AS ?x4)
                  WHERE { ?x2 :author ?x5 . }
                  GROUP BY ?x2 }
                FILTER (?x4 >= 2) . } } } }
```

The two nested `FILTER NOT EXISTS` encode the universal quantifier "every", and the subquery with aggregation encodes the numeric quantifier "at least 2".

The fully-reduced intermediate representation, after semantic transformations and validation (Sections 4.6 and 4.7), is only made of the following semantic primitives: **triple**, **pred**, **func**, **aggreg**, **modif**, **command**, **context**, **true**, **not**, **and**, **or**, **option**, **implies**, **exists**, **forall**, **ask**, **select**, **return**, **graphliteral** (see Table 2). We define in the following a mapping of those primitives to SPARQL 1.1 constructs. The notation $[\![s]\!]$ denotes the SPARQL translation of the statement s. Such a mapping provides at the same time a concrete semantics for SQUALL, and a practical way to evaluate SQUALL sentences by leveraging existing SPARQL engines and endpoints. We first consider the translation of interrogative sentences, and then the translation of imperative and declarative sentences. While the former always generate SPARQL queries, the latter generate both queries and updates. In all cases, the translation is not designed to produce optimal SPARQL code, but to be simple. This should not be a high concern given that SPARQL engines perform optimizations before evaluating queries and updates.

5.1. Interrogative sentences

Interrogative sentences have intermediate representations that start with primitives **ask** or **select**, which respectively generate `ASK`-queries and `SELECT`-queries in SPARQL. The notation $[\![X \mid s]\!]_Q$ denotes an auxiliary translation for multidimensional queries (nested **select**) where X is a row of variables, and the notation $[\![s]\!]_G$ denotes the translation of a statement into a graph pattern. Every occurence of a SPARQL variable $?x$ assumes the generation of a fresh variable name. Those variables are used to instantiate the description parameters of quantifiers and aggregations.

$$[\![\textbf{ask } s]\!] = \texttt{ASK } \{ \ [\![s]\!]_G \ \} \qquad [\![X \mid \textbf{select } d]\!]_Q = [\![X \ ?x \mid d \ ?x]\!]_Q$$
$$[\![\textbf{select } d]\!] = [\![?x \mid d \ ?x]\!]_Q \qquad [\![X \mid s]\!]_Q = \texttt{SELECT } X \texttt{ WHERE } \{ \ [\![s]\!]_G \ \}$$

Most statements can be mapped to graph patterns. Predicates and functions translate to SPARQL filters, and aggregations translate to SPARQL aggregative sub-queries. Modifiers translate to solution modifiers using sub-queries. Contexts translate to named graph patterns, and service patterns. Algebraic constructors translate to their SPARQL counterpart, and quantifiers all translate to the implicit SPARQL existential quantifier and negation.

$$
\begin{aligned}
&[\![\textbf{triple } s\ p\ o]\!]_G = s\ \ p\ \ o\ \ .\\
&[\![\textbf{pred } p\ lx]\!]_G = \texttt{FILTER}\ \ p(lx)\\
&[\![\textbf{func } \texttt{ID}\ [x]\ y]\!]_G = \texttt{BIND}\ \ (x\ \ \texttt{AS}\ \ y)\\
&[\![\textbf{func } f\ lx\ y]\!]_G = \texttt{BIND}\ \ (f(lx)\ \ \texttt{AS}\ \ y)\\
&[\![\textbf{aggreg } g\ d\ x]\!]_G =\\
&\quad \{\ \texttt{SELECT}\ \ ?z^*\ \ (g(?y)\ \ \texttt{AS}\ \ x)\\
&\qquad \texttt{WHERE}\ \ \{\ [\![d\ \ ?z^*\ \ ?y]\!]_G\ \ \}\\
&\qquad \texttt{GROUP BY}\ \ ?z^*\ \ \}\\
&[\![\textbf{modif } m\ s]\!] = \{\ \texttt{SELECT * WHERE}\ \ \{\ [\![s]\!]_G\ \ \}\ \ m\ \ \}\\
&[\![\textbf{context } \texttt{GRAPH}\ x\ s]\!]_G = \texttt{GRAPH}\ \ x\ \ \{\ [\![s]\!]_G\ \ \}\\
&[\![\textbf{context } \texttt{SERVICE}\ x\ s]\!]_G = \texttt{SERVICE}\ \ x\ \ \{\ [\![s]\!]_G\ \ \}\\
&[\![\textbf{true}]\!]_G = \epsilon\\
&[\![\textbf{and } s_1\ s_2]\!]_G = [\![s_1]\!]_G\ \ [\![s_2]\!]_G\\
&[\![\textbf{or } s_1\ s_2]\!]_G = \{\ [\![s_1]\!]_G\ \ \}\ \ \texttt{UNION}\ \ \{\ [\![s_2]\!]_G\ \ \}\\
&[\![\textbf{option } s]\!]_G = \texttt{OPTIONAL}\ \ \{\ [\![s]\!]_G\ \ \}\\
&[\![\textbf{not } s]\!]_G = \texttt{FILTER NOT EXISTS}\ \ \{\ [\![s]\!]_G\ \ \}\\
&[\![\textbf{exists } d]\!]_G = [\![d\ ?x]\!]_G\\
&[\![\textbf{forall } d_1\ d_2]\!]_G = [\![\textbf{not } (\textbf{exists } \lambda x.(\textbf{and } (d_1\ x)\ (\textbf{not } (d_2\ x))))]\!]_G
\end{aligned}
$$

5.2. Imperative and declarative sentences

In imperative and declarative sentences, universal quantifiers (**forall**) and implications (**implies**) introduce a `WHERE`-clause, and existential quantifiers introduce new blank nodes (using SPARQL function `BNODE()`). In order to simplify the following mappings, we first define a few simplification rules that reduce all quantifiers to implications, and that collapse nested implications.

$$
\begin{aligned}
&\textbf{exists } d = \textbf{forall } \lambda x.(\textbf{func } \texttt{BNODE}\ []\ x)\ d\\
&\textbf{forall } d_1\ d_2 = \textbf{implies } (d_1\ ?x)\ (d_2\ ?x)\\
&\textbf{implies } s_1\ (\textbf{implies } s_2\ s_3) = \textbf{implies } (\textbf{and } s_1\ s_2)\ s_3
\end{aligned}
$$

Declarative sentences translate to updates that add and/or delete triples. The SPARQL form depends on the presence of a `WHERE`-clause.

$$
\begin{aligned}
&[\![\textbf{implies } s_1\ s_2]\!] = \texttt{DELETE}\ \ \{\ D\ \ \}\ \ \texttt{INSERT}\ \ \{\ I\ \ \}\ \ \texttt{WHERE}\ \ \{\ [\![s_1]\!]_G\ \ \}\\
&\qquad \text{given } [\![\texttt{DEFAULT} : s_2]\!]_U = (I, D)\\
&[\![s]\!] = \texttt{INSERT DATA}\ \ \{\ I\ \ \};\ \texttt{DELETE DATA}\ \ \{\ D\ \ \}\\
&\qquad \text{given } [\![\texttt{DEFAULT} : s]\!]_U = (I, D)
\end{aligned}
$$

This translation relies on the auxiliary translation $[\![g : s]\!]_U$ that generates two graph patterns (I, D), where I contains the triples to be inserted, and D contains the triples to

be deleted. The parameter g specifies in which graph triples should be inserted/removed.

$[\![\texttt{DEFAULT} : \textbf{triple}\ s\ p\ o]\!]_U = (s\ \ p\ \ o\ \ .\ , \epsilon)$
$[\![g : \textbf{triple}\ s\ p\ o]\!]_U = (\texttt{GRAPH}\ \ g\ \ \{\ \ s\ \ p\ \ o\ \ .\}, \epsilon)$
$[\![g : \textbf{and}\ s_1\ s_2]\!]_U = (I_1\ I_2, D_1\ D_2)$ given $[\![g : s_1]\!]_U = (I_1, D_1), [\![g : s_2]\!]_U = (I_2, D_2)$
$[\![g : \textbf{not}\ s]\!]_U = (D, I)$ given $[\![g : s]\!]_U = (I, D)$
$[\![g : \textbf{context}\ \texttt{GRAPH}\ x\ s]\!]_U = [\![x : s]\!]_U$

Imperative sentences can generate many forms of SPARQL queries and updates, depending on the imperative verb. Only some of those forms accept a `WHERE`-clause, and the mapping is therefore a partial function. For example, the command `CLEAR` cannot be used with a `WHERE`-clause, even though it would make sense: e.g., "Clear all named graphs in which graph John owns a unicorn.".

$[\![\textbf{command}\ \texttt{LOAD}\ [x; y]]\!] = \texttt{LOAD}\ \ x\ \ \texttt{INTO GRAPH}\ \ y$
$[\![\textbf{command}\ \texttt{CLEAR}\ [x]]\!] = \texttt{CLEAR}\ \ x$
$[\![\textbf{command}\ \texttt{DESCRIBE}\ [x]]\!] = \texttt{DESCRIBE}\ \ x$
$[\![\textbf{implies}\ s_1\ (\textbf{command}\ \texttt{DESCRIBE}\ [x])]\!] = \texttt{DESCRIBE}\ \ x\ \ \texttt{WHERE}\ \{\ [\![s_1]\!]_G\ \}$
$[\![\textbf{implies}\ s_1\ (\textbf{return}\ (\textbf{graphliteral}\ s_2))]\!] = \texttt{CONSTRUCT}\ \{\ I\ \}\ \texttt{WHERE}\ \{\ [\![s]\!]_G\ \}$
given $[\![\texttt{DEFAULT} : s_2]\!]_U = (I, \epsilon)$
$[\![\textbf{implies}\ s_1\ (\textbf{return}\ x)]\!] = \texttt{SELECT}\ \ x\ \ \texttt{WHERE}\ \{\ [\![s_1]\!]_G\ \}$
$[\![\textbf{and}\ s_1\ s_2]\!] = [\![s_1]\!]\ \ ;\ \ [\![s_2]\!]$

6. Evaluation of expressiveness by backward translation

We have defined in previous sections the translation from SQUALL to SPARQL, which demonstrate the adequacy of SQUALL for querying and updating RDF stores. However, it remains to evaluate the coverage of SPARQL expressiveness by SQUALL. We do so by sketching the backward translation from SPARQL to SQUALL. This demonstrates that SQUALL is as expressive as SPARQL 1.1 apart from arbitrary length paths and a few minor things (discussed at the end of this section). Note that the purpose of this backward translation is *not* to produce natural verbalizations of SPARQL queries, which is another issue [32], even though it could be used as a starting point.

SPARQL has a very rich syntax, but it can be simplified by transforming non-essential constructs to essential constructs. Turtle syntactic sugar is equivalent to basic graph patterns, and blank nodes can be replaced by variables. Property paths can also be expanded into graph patterns, except arbitrary length paths. In filters, usage of the function `IN` can be replaced by disjunction (`||`). A `VALUES`-clause can be moved into a graph pattern, and complex `VALUES`-patterns can be split into unions of joins of atomic `VALUES`-pattern (e.g., `VALUES ?x { :John }`). Expressions out of a graph pattern (e.g., `SELECT`, `ORDER BY`) can be moved into the graph pattern using `BIND` and a fresh variable. A `HAVING`-clause can be moved into the graph pattern as a filter, after reifying the aggregation as a subquery. References to graphs (e.g., `FROM`, `WITH`, `USING`) can be replaced by `GRAPH`-patterns to specify where triples should be queried or updated. Finally, in a `SELECT *` clause, the joker `*` can be replaced by all free variables in the graph pattern.

We present backward translation in a format similar to forward translation, but from SPARQL constructs to SQUALL phrases. SPARQL syntagms are typed with SQUALL

syntagms. The four forms of queries are translated to either interrogative or imperative sentences. Variables in the `SELECT`-clause are marked so that the determiner which is used upon their first occurence in the translation of the graph pattern. The solution modifier clause (e.g., `ORDER BY DESC(?x) LIMIT 1`) is translated into a adjective modifier to be used upon the first occurence of its variable (e.g., the highest thing ?x). The different forms of updates are translated to either declarative sentences, using a conditional for `WHERE`-clauses, or imperative sentences for graph-level updates.

Query : *Q*
- → `ASK` *Graph* { whether *graph* ? }
- → `DESCRIBE` *URI* { describe *uri* . }
- → `DESCRIBE` *Var* `WHERE` *Graph* { describe *var* where *graph* . }
- → `CONSTRUCT` *Graph* `WHERE` *Graph* { return that $graph_1$ where $graph_2$. }
- → `SELECT` *Var* ... *Var* `WHERE` *Graph* *SolModif* { *graph* ? }

Update : *U*
- → `INSERT DATA` *Graph* { *graph* . }
- → `DELETE DATA` *Graph* { not *graph* . }
- → `DELETE WHERE` *Graph* { if *graph* then not *graph* . }
- → `DELETE` *Graph* `INSERT` *Graph* `WHERE` *Graph* { if $graph_3$ then not $graph_1$ and $graph_2$. }
- → `LOAD` *URI* `INTO GRAPH` *URI* { load uri_1 into uri_2 . }
- → `CLEAR GRAPH` *URI* { clear *uri* . }
- → *Update* `;` *Update* { $update_1$ $update_2$ }

Graph patterns are translated to sentences. Triple patterns translate to simple sentences, depending on whether a class or a property is used, and whether a variable or a URI is used. Classes are translated to nouns, properties to transitive verbs, and aggregations to aggregation adjectives. A variable `?x` is translated as an apposition in a noun phrase (e.g., a thing ?x), on its first occurence, and then to a reference (e.g., ?x).

Graph : *S*
- → *Term* `a` *Var* `.` { *term* belongs to *var* }
- → *Term* `a` *URI* `.` { *term* is a *uri* }
- → *Term* *Var* *Term* `.` { *var* relates $term_1$ to $term_2$ }
- → *Term* *URI* *Term* `.` { $term_1$ *uri* $term_2$ }
- → `BIND(`*Expr* `AS` *Var*`)` { *var* is *expr* }
- → `VALUES` *Var* `{` *Term* `}` { *var* is *term* }
- → *Graph* *Graph* { ($graph_1$ and $graph_2$) }
- → *Graph* `UNION` *Graph* { ($graph_1$ or $graph_2$) }
- → *Graph* `MINUS` *Graph* { ($graph_1$ and not $graph_2$) }
- → `OPTIONAL` *Graph* { maybe *graph* }
- → `FILTER` *Constr* { *constr* }
- → `GRAPH` *Term* *Graph* { at graph *term* *graph* }
- → `SERVICE` *Term* *Graph* { from service *term* *graph* }
- → { `SELECT` Var_1 ... Var_n `(`*Aggreg*`(`Var_y`)` `AS` Var_x`)` `WHERE` *Graph* }
  { var_x is the *aggreg* things var_y such that *graph* per var_1 and ... and per var_n }

Constraints are also translated to sentences, using the same SQUALL coordinations for Boolean operators as for relational algebra operators. Constraints simply produce

sentences based on predicates, where graph patterns produce sentences based on classes and properties. Therefore, unary predicates translate to nouns (like classes), and binary predicates translate to transitive verbs (like properties). Expressions translate to noun phrases, as they play the same role as variables and RDF terms.

Constr : *S*
- → *Expr* > *Expr* { $expr_1$ is greater than $expr_2$ }
- → *Pred1*(*Expr*) { *expr* is a *pred1* }
- → *Pred2*(*Expr*,*Expr*) { $expr_1$ *pred2* $expr_2$ }
- → *Constr* || *Constr* { ($constr_1$ or $constr_2$) }
- → *Constr* && *Constr* { ($constr_1$ and $constr_2$) }
- → !*Constr* { not *constr* }
- → EXISTS *Graph* { *graph* }
- → NOT EXISTS *Graph* { not *graph* }

Expr : *NP*
- → *Expr* + *Expr* { ($expr_1$ + $expr_2$) }
- → *Func*(*Expr*, ..., *Expr*) { *func*($expr_1$,...,$expr_n$) }
- →*Literal* { *literal* }
- →*Var* { *var* }

We have looked in detail at SPARQL's grammar[9] in order to identify all restrictions of SQUALL compared to SPARQL. The main restriction is arbitrary length property paths (e.g., `(^author/author)+`), which involves a form a recursivity that is difficult to express in natural language. Moreover, if recursivity could be expressed in SQUALL, it would probably exceed SPARQL's expressivity because recursivity only applies to property paths, and not to arbitrary graph patterns. The other restrictions of SQUALL are of lesser importance:

- n-ary `DESCRIBE`: e.g., `DESCRIBE ?x ?y WHERE {?x a :man. ?x :spouse ?y.}`;
- n-ary `ORDER BY`: e.g., `ORDER BY ASC(?age) DESC(?size)`;
- missing functions: `&&`, `||`, `!`, `UUID`, `STRUUID`, `COALESCE`, `IF`, `sameTerm`;
- the third argument of `REGEX` is fixed to `'i'`;
- `DISTINCT` is always used with `SELECT`;
- `SILENT` is never used in updates and with `SERVICE`.

7. Evaluation on the QALD challenge

This section reports on the evaluation of SQUALL on the QALD-3 challenge, in which we took part. The QALD[10] challenge (Query Answering over Linked Data) provides "a benchmark for comparing different approaches and systems that mediate between a

[9] `http://www.w3.org/TR/sparql11-query/#sparqlGrammar`
[10] `http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/`

user, expressing his or her information need in natural language, and semantic data". The last campaign, QALD-3, provides hundreds of questions in natural language over two large and real linked datasets: DBpedia [33] and MusicBrainz [34]. The principle of the challenge is that a training set of 100 questions is provided, along with SPARQL translations and answers, and systems are evaluated on a test set that is made of 100 new questions. Systems are compared in terms of precision and recall for the test questions.

The QALD questions do not cover all features of SPARQL 1.1, and hence, do not permit to evaluate all features of SQUALL. However, we find that the DBpedia questions provide a rich set of questions in natural language on many topics, and cover a wide range of SPARQL features including aggregations and solution modifiers. The missing features are updates, named graphs, built-ins and expressions, graph literals (`CONSTRUCT` and `DESCRIBE` queries), and collections.

The objective of our participation in the QALD-3 question answering task was to evaluate the capability of SQUALL to express English questions in a natural and precise way, and the precision and recall of its SPARQL translations. Because our approach relies on a reformulation of the original questions, which is allowed by the QALD challenge, the QALD measures of precision and recall are not enough to evaluate it. Therefore, before presenting the performance of SQUALL in the QALD-3 challenge (Section 7.3), we first assess the grammaticality of QALD questions in SQUALL (Section 7.1), and the naturalness of their reformulation in SQUALL (Section 7.2). We do not evaluate the efficiency of our system in detail as this is not an issue: the 100 questions of the challenge are translated in 6s. We do not present either a usability study on the formulation of SQUALL queries because we consider that guided input is necessary to overcome the habitability problem [12], which is not the object of this paper and is already addressed in existing work [13].

In our evaluation, we focus on the 99 English test questions (question 18 does not exists in QALD-3) for the DBpedia dataset. Questions in other languages are out of the scope of this paper. Training questions are very similar to test questions, and the same conclusions can be drawn from them. For the MusicBrainz dataset, the definition of a custom lexicon would be necessary to keep SQUALL questions natural because many relations are reified as events.

7.1. Grammaticality of QALD questions

In this subsection, we assess the grammaticality of QALD questions in SQUALL. In other words, we answer for each question whether it can be parsed as a SQUALL sentence assuming the right lexicon and the right schema have been defined. For example, the first question "Which German cities have more than 250000 inhabitants?" is a correct SQUALL sentence, if we assume that "German" is an adjective, "cities" is a noun, and "inhabitants" is a relational noun. It also assumes that there is a relation from each city to each known inhabitant. This is unlikely in a dataset like DBpedia, but quite possible in other contexts (e.g., in a civil registry). Therefore, grammaticality evaluates the coverage of natural language by SQUALL, but not its adequation to a particular RDF dataset. It is all about syntax, and none about lexicon.

Among the 99 test questions, we found that 57 of them are grammatical. Most of the other 42 questions have only one ungrammatical element, and have therefore a small edit distance to a grammatical question. We list below the different cases of ungrammaticality, by decreasing frequency. In each case, we show an example where the

ungrammatical element is underlined, and add between brackets a possible replacement to make it grammatical.

- prepositions attached to nouns: "List all games by GMT." (relational adjective: made by);
- temporal and spatial phrases: "When did Michael jackson die?" (prepositional phrase: In which year);
- pronouns: "Which films starring Clint Eastwood did he direct himself?" (repeated proper noun: Clint Eastwood);
- proper noun as apposition: "Who created the comic Captain America?" (proper noun: Captain America);
- measure adjectives: "How tall is Michael Jordan?" (use relational noun: What is the height of);
- comparative adjectives: "Was the Cuban Missile Crisis earlier than the Bay of Pigs Invasion?" (use a relational noun: Did ... have an earlier date).
- composite nouns: "Give me the Apollo 14 astronauts." (noun group: astronauts with mission Appolo 14);
- void undefinite article: "Give me a list of all trumpet players that were bandleaders." (singular noun: a bandleader);
- perfect tenses: "How many space missions have there been?" (use past: were there);

The difficulty of those cases is less a syntactic problem than a semantic problem. For most cases, it would be relatively easy to extend SQUALL's grammar, but some of them involve implicit relations that can only be guessed from expertise on the domain and context. For example, "games by GMT" can mean "games produced by GMT" or "games developed by GMT".

7.2. Naturalness of SQUALL questions

The cost of using a CNL instead of spontaneous language is the need for reformulation, and the benefits is more precision and less ambiguity. We have reformulated the QALD questions into SQUALL sentences with the double objective to keep them as natural as possible, and to match the RDF schema of DBpedia. From the training phase, we had already learned some of the DBpedia vocabulary, and other URIs were found manually with Google searches and DBpedia browsing. We spent on average a few minutes per question for the reformulation phase. The SQUALL reformulations of the 99 test questions, as well as the reformulations of the 100 training questions, are available on SQUALL's homepage[11]. For illustration purposes, Table 4 lists a few original questions along with their SQUALL reformulation. From a detailed analysis of all reformulations, we derive the following conclusions.

[11] `http://http://lisfs2008.irisa.fr/ocsigen/squall/examples`: for each question, the SPARQL translation and answers from DBpedia can be obtained in two clicks.

1	"Which German cities have more than 250000 inhabitants?" "Which Town that has country res:Germany has a populationTotal greater than 250000?"
2	"Who was the successor of John F. Kennedy?" "Who was the successor of res:John_F._Kennedy?"
3	"Who is the mayor of Berlin?" "Who is the leader of res:Berlin?"
4	"How many students does the Free University in Amsterdam have?" "What is the numberOfStudents of res:Vrije_Universiteit?"
5	"What is the second highest mountain on Earth?" "Which Mountain has the 2nd highest elevation?"
7	"When was Alberta admitted as province?" "What is the dbp:admittancedate of res:Alberta?"
9	"Give me a list of all trumpet players that were bandleaders." "Give me all Person-s whose instrument is res:Trumpet and whose occupation is res:Bandleader."
12	"Give me all world heritage sites designated within the past five years." "Give me all WorldHeritageSite whose dbp:year is between 2008 and 2013."
15	"What is the longest river?" "Which River has the highest dbp:length?"
21	"What is the capital of Canada?" "What is the capital of res:Canada?"
26	"How many official languages are spoken on the Seychelles?" "How many officialLanguage-s of res:Seychelles are there?"
28	"Give me all movies directed by Francis Ford Coppola." "Give me all Film-s whose director is res:Francis_Ford_Coppola."
32	"How often did Nicole Kidman marry?" "How many spouse-s of res:Nicole_Kidman are there?"
74	"When did Michael Jackson die?" "What is the deathDate of res:Michael_Jackson?"

Table 4: A sample of original QALD-3 questions, followed by their SQUALL reformulation.

The concision of SQUALL is comparable to natural language. Table 5 compares the average length of questions in three languages: English (original QALD question), SQUALL (our reformulation of the questions), SPARQL (the golden standard provided by QALD organizers). Whereas SPARQL queries (excluding prologues) are two and a half times longer than natural language questions, SQUALL queries are only about 33% longer.

language	English	SQUALL	SPARQL
average question length	45	60	111

Table 5: Comparison of the average length of questions in the three languages.

The difference between natural language and SQUALL is in a large part explained by the namespaces in qualified names (e.g., `res:Berlin` instead of `Berlin`).

SQUALL queries look natural. The use of variables is hardly ever necessary in SQUALL (none was necessary in both training and test questions), while SPARQL queries are cluttered with many variables. No special notations were used, except for namespaces. Only grammatical words are used to provide syntax, and they are used like in natural language. There are 10 out of 99 questions where SQUALL is identical to natural language, up to proper names replaced by readable URIs, namespace prefixes, and plural marks (numbers are question ids): (2) "Who was the successor of res:John_F._Kennedy?", (14) "Give me all bandMember-s of res:Prodigy.", (21) "What is the capital of res:Canada?", (22) "Who is the dbp:governor of res:Wyoming?", (24) "Who was the dbp:father of res:Elizabeth_II?", (30) "What is the dbp:birthName of res:Angela_Merkel?", (54) "What are the dbp:nickname-s of res:San_Francisco?", (58) "What is the dbp:timezone of res:Salt_Lake_City?", (73) "How many child-s did res:Benjamin_Franklin have?", (76) "List the child-s of res:Margaret_Thatcher.".

Most discrepancies between natural language and SQUALL are a matter of vocabulary. Most discrepancies come from the fact that for each concept, a single word has been chosen in the DBpedia ontology, and related words are not available as URIs. Because our default lexicon uses URIs as nouns and verbs, some reformulation is necessary. In the simplest case, it is enough to replace a word by another: e.g., "wife" vs "dbp:spouse". In other cases, a verb has to be replaced by a noun, which requires changes in the syntactic structure: e.g., "Who developed Minecraft?" vs "Who is the developer of res:Minecraft?". An interesting example is "Who is the daughter of Bill Clinton married to?" vs "Who is the dbp:spouse of the child of res:Bill_Clinton?". The former question could be expressed in SQUALL if "marriedTo" was made an equivalent property to "dbp:spouse", and if "daughter" was made a subproperty of "child". In fact, this kind of discrepancy could be resolved, either by enriching the ontology with related words, or by preprocessing user sentences to replace spontaneous words by URIs. Alternately, the replacement could be done on the intermediate representation of even SPARQL query, with the benefit that the semantic role of words (e.g., class, property) would be explicit. The latter approach is analogue to and could reuse the techniques of ontology-based query answering systems (see Section 8).

Some discrepancies are deeper in that they exhibit conceptual differences between natural language and the ontology. We shortly discuss three cases:

- "List all episodes of the first season of the HBO television series The Sopranos!" vs "List all TelevisionEpisode-s whose series is res:The_Sopranos and whose seasonNumber is 1.". In natural language, an episode is linked to a season, which in turn is linked to a series. In DBpedia, an episode is linked to a series, on one hand, and to a season number, on the other hand. In DBpedia, a season is not an entity, but only an attribute of episodes.

- "Which caves have more than 3 entrances?" vs "Which Cave-s have an dbp:entranceCount greater than 3?". The natural question is a grammatical sentence in SQUALL, but it assumes that each cave is linked to each of its entrances. However, DBpedia only has a property "dbp:entranceCount" from a cave to its number of entrances.
- "Which classis does the Millepede belong to?" vs "What is the dbp:classis of res:Millipede?". The natural question is again a valid SQUALL sentence (after moving 'to' at the beginning), but it assumes that `res:Millipede` is an instance of a class that is itself an instance of `dbp:classis`. DBpedia does not define classes of classes, and therefore uses `dbp:classis` as a property from a species to its classis.

Those discrepancies are more difficult to solve. A first solution is to make the ontology better fit usage in natural language. A second solution is to apply transformations on the intermediate representation of natural SQUALL sentences so that it matches the ontology (e.g., transforming a count of entrances into the value of property dbp:entranceCount).

7.3. Performance in QALD-3 challenge

We here report on the results of our system SQUALL2SPARQL in QALD-3 challenge. They are published on the official website of the challenge. Other QALD-3 participants and their results are discussed in Section 8 (they are much lower but not directly comparable). We submitted the SPARQL translations produced by our system from our reformulations of the test questions to the QALD evaluation tool. Out of the 99 questions, we got the right answers for 80 questions (including the three OUT OF SCOPE questions), and partial answers for 13. Recall was 0.88, precision was 0.93, and the F-measure was 0.90. The errors come from:

- data heterogeneity (12 errors, questions 1, 6, 17, 19, 29, 33, 39, 60, 63, 72, 93, 96),
- the reformulation in SQUALL (2 errors, questions 14, 43),
- SQUALL2SPARQL (2 errors, questions 49, 59),
- the gold standard (2 errors, question 16, 75),
- the QALD endpoint (1 error, question 92).

Looking at heterogeneity errors in detail, it appears that most of them could be solved simply by: either adding generic super-properties in the DBpedia ontology, or by expanding common words (e.g., location, date) into UNION graph patterns. For example, in question 39 "Give me all companies in Munich.", the implicit relation "has location" can be translated in any of the three RDF properties: `dbo:location`, `dbo:headquarter`, `dbo:locationCity`. This explains why our reformulation in SQUALL "Give me all Company-es whose location is res:Munich." has recall 0.6 only (the default prefix was used for DBpedia ontology, so that `location` stands for `dbo:location`). If `location`, or another property, was defined as a super-property of the other properties, the same SQUALL question would have recall 1. Alternatively, assuming linguistic knowledge, the word "location" could be mapped to the graph pattern

```
{ ?x dbo:location ?y }
 UNION { ?x dbo:headquarter ?y }
 UNION { ?x dbo:locationCity ?y }
```

where `?x` and `?y` respectively stand for the subject and object of the relation. Such graph patterns could easily be exploited in the translation from the intermediate representation to SPARQL without the need to change the SQUALL language and its parsing.

Another problem related to heterogeneity is that some expected domain and range axioms are not verified in some cases. For example, in question 19 "Give me all people that were born in Vienna and died in Berlin.", 2 out of the 6 expected answers are not instances of the class `Person`. This is why our reformulation "Give me all Person-s whose birthPlace is res:Vienna and whose deathPlace is res:Berlin." missed 2 answers, even though it is arguably equivalent to the original formulation.

The errors coming from the reformulation of questions are due to the misspelling or misunderstanding of URIs. In question 14, res:Prodigy was used instead of res:The_Prodigy. In question 43, the property dbp:breed was used in the wrong direction.

The errors coming from SQUALL2SPARQL are due to an incorrect translation of the special verb "share". For example, Question 49 "Which other weapons did the designer of the Uzi develop?" was reformulated as "Which Weapon shares the dbp:designer with res:Uzi?", which returns Uzi itself as an answer. Another possible reformulation is "Which Weapon has the same dbp:designer as res:Uzi?", but it exhibits the same error.

The error from the endpoint is because the `BIND` construct of SPARQL is not (yet) supported by the QALD-3 endpoint. It is possible to write the SPARQL query to avoid it, but SQUALL2SPARQL relies on it to simplify the translation from SQUALL. Note that the correct answers are returned when using the official DBpedia endpoint.

8. Related work

In their evaluation of Natural Language Interfaces (NLI) and interfaces for the Semantic Web, Kaufmann and Bernstein [12] compare the usability of different approaches on a natural-formal scale (the *Formality Continuum*). This scale ranges from spontaneous natural language to formal query language, with Controlled Natural Language (CNL) in between. For formal languages (e.g., SPARQL) or CNLs, the lack of naturalness is compensated by assisting users in the formulation of queries (e.g., auto-completion). Interestingly, the participants of the QALD-3 challenge [14] cover a wide range of the formality continuum, even if most of them (4/6) fall in the "spontaneous natural language" category. We discuss each category in turn, starting with QALD-3 participants.

Spontaneous natural language. QALD-3 participants in this category are: CASIA [35] (F-measure = 0.36), RTV [36] (F-measure = 0.33), Intui2 [37] (F-measure = 0.32), and SWIP [38, 39] (F-measure = 0.17). Many other systems have been developed: e.g., PowerAqua [11], QAKiS [40], FREyA [10]. While they differ a lot in the details, they all share a similar architecture. First, a syntactic analysis is performed, usually based on standard NLP tools (e.g., Stanford NLP tools), in order to identify entities and relationships, and to generate a few triples. Second, a mapping from lexical forms to semantic forms is searched with the help of external resources, which are both linguistic (e.g., WordNet, Wikipedia, relational patterns), and ontologic (e.g., fixed ontology, Watson). Finally, one or several SPARQL queries are generated and ranked. In some cases, answers are directly produced and ranked without using a SPARQL query explicitly. Most of the effort is generally spent on the semantic mapping, and only a shallow syntactic parsing is performed. From QALD-3 results, CASIA and SWIP generate only 2-triples queries

(which cover 74 test questions), and RDF and Intui2 only 1-triple queries (60 questions). Only RTV supports counting (4 questions), and none of them supports comparatives (6 questions) or superlatives (6 questions). Among their wrong or missing results, roughly half can be accounted on syntax, and half on semantic mapping.

Controlled natural language. SQUALL is the only QALD-3 participant in this category (a former version of SWIP took part in QALD-1 as a CNL). It focuses on syntactic analysis, and SPARQL generation, and completely ignores (so far) semantic mapping. Therefore, it relies on a reformulation of original questions in the vocabulary of the target RDF dataset, and its score in the challenge (F-measure = 0.90) is hence not directly comparable to other participants' score. However, it demonstrates that complex natural questions can be reliably translated to SPARQL. SQUALL better handles complex syntactic constructs such as nested relative clauses, and coordinations. It can generate arbitrarily many triples, it supports all features of QALD questions (comparatives, superlatives, counting), and more (disjunction, negation, other aggregations, updates, etc.). Many CNLs have been defined in the past decades [17], but SQUALL is the only one that targets SPARQL queries and updates. Other CNLs for the Semantic Web rather target ontological axioms (facts and rules). ACE [19] is a general purpose CNL that can target various formalisms, and there are a number of more specialized CNLs targeting OWL axioms (e.g., SOS, Rabbit) [20]. In Kuhn's survey [17], SQUALL is evaluated as having: a high precision (P^5), a medium expressiveness compared to NL (E^3), natural sentences (N^4), and a short description (S^4). In comparison, a natural language is evaluated as $P^1E^5N^5S^1$, i.e., maximally expressive and natural, but minimally precise and short-defined; and SPARQL is evaluated as $P^5E^3N^1S^5$.

Guided navigation. Scalewelis [41] (F-measure = 0.33) is the only QALD-3 participant based on guided navigation. Like SQUALL, it is based on a reformulation of original queries, but users are guided step by step, and do not have to actually write the query. Guidance is based on actual data in the RDF store, and follows the principles of Query-based Faceted Search (QFS) [8], which allows for more expressiveness than other semantic faceted search systems (e.g., SlashFacet [6], BrowseRDF [7]). At each step, users are given a choice of relevant classes, properties, and entities to be inserted in the query under construction. Scalewelis' query language is a subset of SQUALL that supports nested relative clauses and conjunctive coordinations, but none of comparisons, superlatives, and aggregations. Errors in the QALD-3 challenge are explained by the lack of expressivity, and by the high heterogeneity of DBpedia's vocabulary. Ginseng [13], resp. GINO [42], guides users in the input of queries, resp. resource descriptions, through auto-completion of CNL sentences. However, their expressiveness is lower than SQUALL, and their guidance is based on an ontology rather than on actual data, which is less precise and requires a well-defined ontology.

9. Conclusion and perspectives

In the spectrum that goes from full natural language to formal languages like SPARQL, SQUALL (Semantic Query and Update High-Level Language) occupies a unique position. It offers the same expressiveness as SPARQL for querying and updating RDF data, and still qualifies as a controlled natural language (CNL). This means

that among the natural language interfaces, SQUALL is the one that is by far the most expressive; and that among the formal languages, SQUALL is the one that is the most natural. The current limit of SQUALL is that end-users have to comply with its controlled syntax, and have to know the RDF vocabulary (i.e., *Which are the classes and properties?*). However, the important result is that SQUALL can be used as a substitute for SPARQL because this entails no loss, neither in expressiveness, nor in precision.

Most existing systems try and solve the query answering challenge by addressing all aspects at once: syntactic analysis, mapping from lexical forms to semantic forms, contextual disambiguation, and SPARQL generation. We think that it may be more efficient, and that it would facilitate collaboration, to address those aspects separately. We have shown in this paper that syntactic analysis and SPARQL generation can reach the expressivity of SPARQL, while retaining a high level of naturalness (at the syntactic level). Future work will consist in integrating SQUALL with existing results about other aspects:

- *using domain-specific lexicons instead of the default one.* Such lexicons may be constructed manually, or generated automatically [43]. There is an RDF vocabulary, Lemon [44], to represent and share such lexicons.
- *applying mapping techniques on the intermediate representation.* SQUALL's intermediate representation is compatible with those of OQA systems, only being more complex combinations of triple patterns. In simple cases, mapping simply replaces words by URIs (e.g., "mayor" by `dbo:leader`). In more complex cases, mapping may replace combinations of triples by other combinations of triples (e.g., a count of "entrances" by the property `dbp:entranceCount`).
- *using SQUALL as an intermediate representation.* If full natural language is to be accepted as input, a solution is to use mature NLP tools (e.g., Stanford NLP parser) to parse a spontaneous sentence, and then translate the resulting dependency graph to SQUALL, which is arguably much closer to NL than SPARQL is.
- *using guided input and dialog.* Grammar-based guided input (auto-completion) ensures that only syntactically correct sentences are entered by users [13]. Another possible approach is to use query-based faceted search, which combines the guided exploration of faceted search and the expressivity of query languages [8]. If a lexicon is available, it can also ensures that only known words, and hence known concepts, are used. Finally, if data is available during guided input, it can be queried to rule out invalid interpretations, and the user can be asked to choose among the remaining interpretations. The combination of those mechanisms open the door for a rich dialog between the user and the system, but remains challenging in terms of complexity and scalability.

Acknowledgement. I would like to thank Christina Unger for fruitful interaction during the QALD-3 challenge, which helped to improve the system, and the SQUALL language itself. I also thank Benjamin Sigonneau for his help in the development and deployment of Web forms, and Jean-Marc Vanel for integrating SQUALL in his tool, EulerGUI, and for suggesting useful features.

References

[1] S. Ferré, SQUALL: a controlled natural language as expressive as SPARQL 1.1, in: E. Métais (Ed.), Int. Conf. Application of Natural Language to Information Systems (NLDB), LNCS 7934, Springer, 2013, pp. 114–125.

[2] S. Ferré, SQUALL: a controlled natural language for querying and updating RDF graphs, in: T. Kuhn, N. Fuchs (Eds.), Controlled Natural Languages, LNCS 7427, Springer, 2012, pp. 11–25.

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

[4] J. Pérez, M. Arenas, C. Gutierrez, Semantics and complexity of SPARQL, in: I. F. C. *et al* (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 30–43.

[5] SPARQL 1.1 query language, `http://www.w3.org/TR/sparql11-query/`, W3C Proposed Recommendation (2012).
URL `http://www.w3.org/TR/sparql11-query/`

[6] M. Hildebrand, J. van Ossenbruggen, L. Hardman, /facet: A browser for heterogeneous semantic web repositories, in: I. C. *et al* (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 272–285.

[7] E. Oren, R. Delbru, S. Decker, Extending faceted navigation to RDF data, in: I. C. *et al* (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 559–572.

[8] S. Ferré, A. Hermann, Reconciling faceted search and query languages for the Semantic Web, Int. J. Metadata, Semantics and Ontologies 7 (1) (2012) 37–54.

[9] V. Lopez, V. S. Uren, M. Sabou, E. Motta, Is question answering fit for the semantic web?: A survey, Semantic Web 2 (2) (2011) 125–155.

[10] D. Damljanovic, M. Agatonovic, H. Cunningham, Identification of the question focus: Combining syntactic analysis and ontology-based lookup through the user interaction, in: Language Resources and Evaluation Conference (LREC), ELRA, 2010.

[11] V. Lopez, M. Fernández, E. Motta, N. Stieler, PowerAqua: Supporting users in querying and exploring the semantic web, Semantic Web 3 (3) (2012) 249–265.

[12] E. Kaufmann, A. Bernstein, Evaluating the usability of natural language query languages and interfaces to semantic web knowledge bases, J. Web Semantics 8 (4) (2010) 377–393.

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

[14] P. Cimiano, V. Lopez, C. Unger, E. Cabrio, A.-C. N. Ngomo, S. Walter, Multilingual question answering over linked data (QALD-3): Lab overview, in: P. Forner, H. Müller, R. Paredes, P. Rosso, B. Stein (Eds.), Information Access Evaluation. Multilinguality, Multimodality, and Visualization - Int. Conf. CLEF Initiative, LNCS 8138, Springer, 2013, pp. 321–332.

[15] A. Hermann, S. Ferré, M. Ducassé, An interactive guidance process supporting consistent updates of RDFS graphs, in: A. ten Teije et al. (Ed.), Int. Conf. Knowledge Engineering and Knowledge Management (EKAW), LNAI 7603, Springer, 2012, pp. 185–199.

[16] N. E. Fuchs, R. Schwitter, Web-annotations for humans and machines, in: E. Franconi, M. Kifer, W. May (Eds.), European Semantic Web Conference, LNCS 4519, Springer, 2007, pp. 458–472.

[17] T. Kuhn, A survey and classification of controlled natural languages, Computational Linguistics.

[18] P. Smart, Controlled natural languages and the semantic web, Tech. rep., School of Electronics and Computer Science University of Southampton (2008).
URL `http://eprints.ecs.soton.ac.uk/15735/`

[19] N. E. Fuchs, K. Kaljurand, G. Schneider, Attempto Controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces, in: G. Sutcliffe, R. Goebel (Eds.), FLAIRS Conference, AAAI Press, 2006, pp. 664–669.

[20] R. Schwitter, K. Kaljurand, A. Cregan, C. Dolbear, G. Hart, A comparison of three controlled natural languages for OWL 1.1, in: K. Clark, P. F. Patel-Schneider (Eds.), Workshop on OWL: Experiences and Directions (OWLED), Vol. 258, CEUR-WS, 2008.

[21] P. Haase, J. Broekstra, A. Eberhart, R. Volz, A comparison of RDF query languages, in: S. M. *et al.* (Ed.), Int. Semantic Web Conf., LNCS 3298, Springer, 2004, pp. 502–517.

[22] D. R. Dowty, R. E. Wall, S. Peters, Introduction to Montague Semantics, D. Reidel Publishing Company, 1981.

[23] R. Montague, Universal grammar, Theoria 36 (1970) 373–398.

[24] H. Barendregt, The Lambda Calculus, Its Syntax and Semantics, Vol. 103 of Studies in Logic and the Foundations of Mathematics, Elsevier, 1985.

[25] L. Damas, R. Milner, Principal type-schemes for functional programs, in: ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages, ACM, 1982, pp. 207–212.
[26] D. Biber, S. Johansson, G. Leech, S. Conrad, E. Finegan, Longman grammar of spoken and written English, Pearson Education Limited, 1999.
[27] P.-Y. Hsu, D. S. P. Jr., Improving SQL with generalized quantifiers, in: P. S. Yu, A. L. P. Chen (Eds.), Int. Conf. Data Engineering, IEEE Computer Society, 1995, pp. 298–305.
[28] L. Sterling, E. Shapiro, The Art of Prolog, MIT Press, Cambridge (MA), 1986.
[29] P. Wadler, Monads for functional programming, in: Advanced Functional Programming, Springer, 1995, pp. 24–52.
[30] H. Kamp, U. Reyle, From discourse to logic: Introduction to model theoretic semantics of natural language, formal logic and discourse representation theory, Kluwer, 1993.
[31] R. Muskens, Combining Montague semantics and discourse representation, Linguistics and philosophy 19 (1996) 143–186.
[32] A.-C. N. Ngomo, L. Bühmann, C. Unger, J. Lehmann, D. Gerber, Sorry, I don't speak SPARQL: translating SPARQL queries into natural language, in: WWW, 2013, pp. 977–988.
[33] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, S. Hellmann, DBpedia - a crystallization point for the web of data, Web Semantics: Science, Services and Agents on the World Wide Web 7 (3) (2009) 154–165.
[34] A. Swartz, Musicbrainz: A semantic web service, Intelligent Systems, IEEE 17 (1) (2002) 76–77.
[35] S. He, S. Liu, Y. Chen, G. Zhou, K. Liu, J. Zhao, CASIA@QALD-3: A question answering system over linked data, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL `http://www.clef2013.org`
[36] C. Giannone, V. Bellomaria, R. Basili, A HMM-based approach to question answering against linked data, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL `http://www.clef2013.org`
[37] C. Dima, Intui2: A prototype system for question answering over linked data, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL `http://www.clef2013.org`
[38] C. Pradel, G. Peyet, O. Haemmerlé, N. Hernandez, SWIP at QALD-3: Results, criticisms and lesson learned, in: C. U. et al. (Ed.), Work. Multilingual Question Answering over Linked Data (QALD-3), 2013.
URL `http://www.clef2013.org`
[39] F. Amarger, O. Haemmerlé, N. Hernandez, C. Pradel, Taking SPARQL 1.1 extensions into account in the SWIP system, in: H. D. Pfeiffer, D. I. Ignatov, J. Poelmans, N. Gadiraju (Eds.), Int. Conf. Conceptual Structures, LNCS 7735, Springer, 2013, pp. 75–89.
[40] E. Cabrio, J. Cojan, A. P. Aprosio, B. Magnini, A. Lavelli, F. Gandon, QAKiS: an open domain QA system based on relational patterns, in: B. Glimm, D. Huynh (Eds.), Int. Semantic Web Conf. (Posters & Demos), Vol. 914 of CEUR Workshop Proceedings, 2012.
[41] J. Guyonvarch, S. Ferre, M. Ducassé, Scalable Query-based Faceted Search on top of SPARQL Endpoints for Guided and Expressive Semantic Search, Research report PI-2009, IRISA (2013).
URL `http://hal.inria.fr/hal-00868460`
[42] A. Bernstein, E. Kaufmann, GINO - a guided input natural language ontology editor, in: I. F. C. et al. (Ed.), Int. Semantic Web Conf., LNCS 4273, Springer, 2006, pp. 144–157.
[43] S. Walter, C. Unger, P. Cimiano, A corpus-based approach for the induction of ontology lexica, in: Int. Conf. Applications of Natural Languages to Information Systems (NLDB), LNCS 7934, Springer, 2013, pp. 102–113.
[44] J. McCrae, D. Spohr, P. Cimiano, Linking lexical resources and ontologies on the semantic web with lemon, in: Extended Semantic Web Conference (ESWC), LNCS 6643, Springer, 2011, pp. 245–259.

Abstract. In many domains where information access plays a central role, there is a gap between expert users who can ask complex questions through formal query languages (e.g., SQL), and lay users who either are dependent on expert users, or must restrict themselves to ask simpler questions (e.g., keyword search). Because of the formal nature of those languages, there seems to be an unescapable trade-off between expressivity and usability in information systems. The objective of this thesis is to present a number of results and perspectives that show that the expressivity of formal languages can be reconciled with the usability of widespread information systems (e.g., browsing, Faceted Search (FS)). The final aim of this work is to empower people with the capability to produce, explore, and analyze their data in a powerful way.

We have proposed a number of theories and implementations to better reconcile expressivity and usability, and applied them to a number of contexts going from file systems to the Semantic Web. In this thesis, we introduce an unifying framework inspired by Formal Concept Analysis (FCA) to factor out the main ideas of all those results: Abstract Conceptual Navigation (ACN). The principle of ACN is to guide users by letting them *navigate* in a conceptual space where places are *concepts* connected by navigation links. Concepts are characterized by a formal query, and are made of two parts: an *extension* and an *intension*. The extension is made of query results while the intension is made of the query itself and an index of query increments over results. Finally, navigation links are formally defined as query transformations. The conceptual space is not static but is induced by concrete data, and evolves with it. ACN therefore combines the *expressivity* of formal query languages with the *guidance* of conceptual navigation. The *readability* of queries is improved by verbalizing them to (or parsing them from) a Controlled Natural Language (CNL). Readability and guidance together support usability by speaking user's language, and by providing a systematic assistance.