KRDB Research Centre Technical Report:

Theoretical Foundations of an Ontology-Based Visual Tool for Query Formulation Support [1]

Paolo Guagliardo

[1]Master's thesis for the European Master's Programme in Computational Logic. Vienna University of Technology (VUT), Austria. Free University of Bozen-Bolzano (FUB), Italy. Supervisors: Proff. Thomas Eiter (VUT) and Enrico Franconi (FUB).

Affiliation	Free University of Bozen-Bolzano
Corresponding author	Paolo Guagliardo paolo.guagliardo@stud-inf.unibz.it
Keywords	ontology, query, framework, description logics, automated reasoning
Number	KRDB09-05
Date	07-10-09
URL	http://www.inf.unibz.it/krdb/

Abstract

Recent research showed that adopting formal ontologies as a means for accessing heterogeneous data sources has many benefits, in that not only does it provide a uniform and flexible approach to integrating and describing such sources, but it can also support the final user in querying them, thus improving the usability of the integrated system.

The Query Tool is an experimental software supporting the user in the task of formulating a precise query – which best captures their information needs – even in the case of complete ignorance of the vocabulary of the underlying information system holding the data. The intelligent interface of the Query Tool is driven by means of appropriate automated reasoning techniques over an ontology describing the domain of the data in the information system.

Although an implementation does exist, there is no characterisation of the Query Tool from the formal point of view, that precisely describes how such a system works and on which theoretical foundations it relies. Indeed, the central purpose of this thesis is that of providing a formal framework in which the Query Tool's components and operations are defined in a precise and unambiguous way.

We will describe what a query is and how it is internally represented by the Query Tool, which operations are available to the user in order to modify the query and how the tool provides contextual feedback about it presenting only relevant pieces of information. Moreover, we will also investigate in detail how a query can be represented in a suitable "linear form", so that it can more easily be expressed in natural language.

As a conclusive part of our work, a new implementation of the Query Tool, superseding the existing one and complying with the formal specification, has been devised. At present, the new system includes only the core of the Query Tool and is meant to provide a demonstrative though fully functional implementation based on our framework.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Prof. Enrico Franconi of the KRDB Research Centre at the Free University of Bozen-Bolzano for his invaluable support and contribution to this thesis. A special thank goes also to Dr. Sergio Tessaris for his precious advices and technical insights into the implementation of the Query Tool.

Last but not least, I thank my supervisor Prof. Thomas Eiter for coordinating my work at the Vienna University of Technology, as well as my family, my friends, my colleagues and all the people who supported me during my study career.

CHAPTER 1

Introduction

The main purpose of this thesis is to provide the reader with the understanding of the theoretical foundations at the basis of a visual tool supporting query formulation over an ontology. Such a tool exists in the form of an experimental software, simply called the “Query Tool”, using many of the ideas that have been introduced in previous research work [2, 12, 11]. However, a precise and formal definition of the framework in which the tool operates has not been provided yet, and this thesis is an attempt to fill such gap.

The Query Tool is a software which enables access to heterogeneous data sources by means of the conceptual schema provided by an ontology and supports the users in the task of formulating a precise query over it. In describing a specific domain, the ontology defines a vocabulary which is often richer than the logical schema of the underlying data and usually closer to the user’s own vocabulary. The ontology can thus be effectively exploited by the user in order to formulate a precise query which best captures their information need. The user is constantly guided and assisted in this task by an intuitive visual interface, whose intelligence is dynamically driven by “reasoning” over the ontology. The inferences drawn on the conceptual schema help the user in choosing what is more appropriate with respect to their information need, restricting the possible choices to only those parts of the ontology which are relevant and meaningful in a given context.

The most powerful and innovative feature of the Query Tool lies in the fact that not only do not users need to be aware of the underlying organisation of the data, but they are also not required to have any specific knowledge of the vocabulary used in the ontology. In fact, such knowledge can be acquired, step by step, while using the tool itself, gaining confidence both with it and the ontology. Furthermore, users may decide to just “explore” the ontology without actually querying the information system, with the aim of discovering general information about the modelled domain.

Another very important aspect is that the Query Tool generates only queries that can be actually satisfied, since contradictory or incompatible pieces of information are not presented to the user at all. This makes user’s choices clearer and simpler, by ruling out irrelevant information that might be distracting and even generate confusion. Moreover, it also eliminates the often frustrating and time-consuming process of finding the right combination of parts constituting a meaningful query. For this reason, the user is thus completely free to explore the ontology and express their information need without the worry of making a "wrong" choice at some point.

Queries can be specified through a refinement process consisting in the iteration of few basic operations: the user first specifies an initial request starting with generic terms, then refines or deletes some of the previously added terms or introduces new ones, and iterates the process until the resulting query satisfies their information need. The available operations on the current query include addition, substitution and deletion of pieces of information, and all of them are supported by the reasoning services running over the ontology.

1.1 Motivation

In this section we present a small example showing how, with the support of the Query Tool, an ontology can be explored in order to discover information about it and the domain it models. Our only goal here is to give the reader a first idea of the Query Tool's potential and field of application, hence the chosen example is deliberately simple and it does not exploit the full capabilities of the tool. For the moment, the tone adopted is also quite informal, as opposed to the one that will be generally used throughout this thesis. The same example will then be extended and analysed in more detail in Chapter 3, in order to facilitate a better understanding of the formal definitions that will be introduced there.

Consider a scenario in which we have an ontology and know nothing about it. For this reason, though a bit skeptical, we decide to ask the Query Tool for help. The tool shows us a string of text, representing a query over our ontology, like the following:

"I'm looking for **something**" . (Q1)

True, but not very helpful: we would like to be more precise about our request. Thus, we select the word "something" and ask to perform on it an operation called *substitution*, consisting in the replacement of the selected portion of query (that is, the *selection*) with a different term. We choose such substituting term from a set of suitable ones, which are computed by the Query Tool and depend on the current query and selection. In particular, we are provided with a menu divided into three parts and organised as follows:

- in the upper part are listed terms that are more general than the selection;
- in the middle part are those terms (if any) equivalent to the selection;
- in the lower part are listed terms that are more specific than the selection.

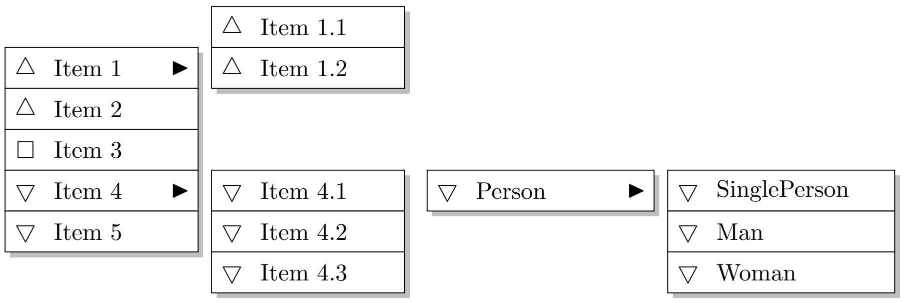
Moreover, from each of the terms in the upper (lower) part we can access nested sub-menus with more and more general (specific) terms, if any. A graphical example of this kind of menu structure is shown in Figure 1.1a, where:

△ indicates that the term is more general than the selection;

□ indicates that the term is equivalent to the selection;

▽ indicates that the term is more specific than the selection.

The symbol "▶" signals the availability of a sub-menu for further generalisation or specialisation. Note that □-terms never have sub-menus, while the sub-menu of a △-term (▽-term) only contains terms that are more general (specific) than it. For instance, every menu item in Figure 1.1a of the form Item 1.x is more general than Item 1 and every menu item of the form Item 4.y is more specific than Item 4.



(a) General structure (b) Concrete example

Figure 1.1: Query Tool's substitution menu

Coming back to our example, Figure 1.1b shows the menu that the Query Tool gives us when asking for the substitution of the term "something" in (Q1). Thus, we find out that a person may be a man, a woman or single, and we begin thinking that our ontology must talk about persons and perhaps relationships among them. We then choose the term "Person" and obtain the query:

"I'm looking for a **person**" . (Q2)

Next, we decide to further refine our query, this time not by means of a substitution of terms, but by *adding a relation* to the term "person". The Query Tool answers that the only suitable relation having a person as subject is "married to", with term "Person" as object. The result of adding this relation to (Q2) is shown below:

"I'm looking for a **person** who is *married to* a **person**" . (Q3)

At this point, an attempt to specialise the leftmost term "person" results in the following options: "Man" and "Woman". Note that the term "SinglePerson" is not available here, even though it was listed as a possible choice in Figure 1.1b for the substitution of the term "something" in (Q1). In fact, the Query Tool opportunely rules out the options that would cause the query to become unsatisfiable, and in this case we discovered that a single person cannot be married. Since "SinglePerson" is also not given as an option for the specialisation of the rightmost term "person", we additionally suspect that the relation "married to" is symmetric, that is, if a person A is married to a person B, then B is married to A.

We decide to go on by substituting the leftmost "person" in (Q3) with the term "Man", obtaining the following query:

"I'm looking for a **man** who is *married to* a **person**" . (Q4)

In order to discover new information we then select the word “person” in (Q4) and ask for a substitution: we get the more specific term “Woman” as the only option, which makes us realise that in our ontology a man can only be married to a woman. The substitution of “person” with “Woman” results in the query:

“I'm looking for a **man** who is *married to* a **woman**” . (Q5)

We now want to check whether a woman can only be married to a man and to do so we first generalise “man” into “person” as follows:

“I'm looking for a **person** who is *married to* a **woman**” . (Q6)

and then again we ask for the substitution of the term “person” itself. As the only option we get “Man”, confirming that in our ontology women cannot be married to other than men.

We are now interested in understanding how a single person relates to men and women. As we know that a single person cannot be married, we first select the term “woman” in (Q6) and ask for its *deletion.* In the resulting query, the relation “married to” disappears along with the selected term, as shown below:

“I'm looking for a **person**” . (Q7)

Then, we specialise “person” into “SinglePerson”, obtaining the following query:

“I'm looking for a **single person**” . (Q8)

Asking the Query Tool which terms are *compatible* with “single person” gives “Man” and “Woman” as answers, meaning that a single person may in addition be a man as well as a woman. If we choose to add the compatible term “Man”, we obtain the following query:

“I'm looking for a **single person** who is a **man**” ; (Q9)

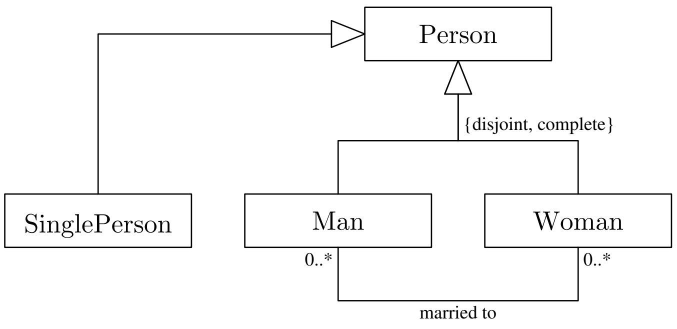
while choosing “Woman” turns (Q8) into:

“I'm looking for a **single person** who is a **woman**” . (Q10)

We now ask for further compatible terms in (Q9) or in (Q10), getting no results. This tells us that a man cannot be at the same time a woman (and vice versa).

The information we have so far discovered and collected about our ontology can be put together and organised into a class diagram in the Unified Modeling Language (UML) enriched with two first-order logic (FOL) formulas, as shown in Figure 1.2. The diagrammatic part expresses the following facts: a person is either a man or a woman; a man can be married to a woman[1] (and vice versa); and some persons may be single. The FOL formulas are needed to impose the additional constraints that “married to” is a symmetric relation and a single person cannot be married.

[1]The only thing we are able to derive using the Query Tool is optional participation into the “married to” relation, but we do not know the exact maximum multiplicity. Thus, we allow for a man (woman) to be possibly married to more than one woman (man), although the ontology could actually be more restrictive about this.



$$\forall x, y \,.\, \text{marriedTo}(x, y) \rightarrow \text{marriedTo}(y, x)$$
$$\forall x \,.\, \text{SinglePerson}(x) \rightarrow \neg\big(\exists y \,.\, \text{marriedTo}(x, y)\big)$$

Figure 1.2: Conceptual schema derived with the support of Query Tool

1.2 Previous Work

The intelligent interface known as Query Tool was devised for the first time in 2004, in the context of the *SEmantic Webs and AgentS in Integrated Economies* (SEWASIE) research project [24], funded by the European Commission. The goal of SEWASIE consisted in the design and implementation of an advanced search engine, which enabled access to heterogeneous data sources on the web via semantic enrichment in order to provide the basis of structured and secure web-based communication. In that early version were already present most of the ingredients that are now at the very core of the actual Query Tool. Here, we summarise the fundamental ideas behind the SEWASIE tool and its main features:

- **focus paradigm**: the manipulation of the query is always restricted to a well defined and visually delimited sub-part of the whole query, called the *focus*;
- **substitution by navigation**: the possibility of substituting the selected portion of the query with more specific or more general terms;
- when operating on the focus, the remaining part of the query expression is not ignored, but fully taken into account *from the focus viewpoint*;
- **refinement by compatible terms**, that are terms in the ontology not in hierarchy with the query expression and whose overlap with the focus is non-empty;
- textual representation of the query in a natural language fashion.

In the course of the SEWASIE project, a usability evaluation of the Query Tool was carried out [4, 3] with the purpose of measuring its complexity of use. In particular, the study aimed at determining how difficult it is for the user to formulate queries using the Query Tool and to understand the results. The experiment involved two distinct groups of users, namely domain experts and non domain experts. The former are users having specific knowledge about the domain to be queried, while the latter have no particular skill in that area. In order to evaluate the level of interplay between the users' domain expertise and the query formulation paradigm used in the Query Tool, different query reading and query writing tasks were devised, along with a number of queries of increasing complexity (according to a model of complexity opportunely designed) and a questionnaire to capture relevant aspects of the interaction with the interface. The experiment had the following outcome:

- the overall functionality and philosophy of the Query Tool interface are well understood by all users;
- the amount of time spent to formulate a query depends only on its complexity and not on the user's domain expertise;
- the questionnaire reveals that the satisfaction of users in completing the specific writing tasks is independent of the domain expertise.

The above results are very important in that they demonstrate that the Query Tool can be easily used also by non-experienced users to formulate queries over a domain in which they have no special expertise.

In order to have a richer and more flexible interface, the web-based version of the Query Tool developed in the context the SEWASIE project was later on converted into a stand-alone Java application [25]. Some important optimisations were then introduced [26], with the purpose of minimising the number of reasoner calls and thus improving the responsiveness of the tool.

1.3 Contribution

The final purpose of the Query Tool is to generate a conjunctive query ready to be executed by some evaluation engine associated with the information system where data are stored. However, in this thesis we only deal with the aspect of query "formulation" rather than "generation", concentrating on the so-called *intensional navigation* of the ontology, an example of which has already been shown in Section 1.1. Thus, in what follows we will describe in great detail the iterative refinement process through which users can formulate a meaningful query, but we will completely disregard how such query is then converted by the system into a form that is executable by a specific evaluation engine.

Although a software implementation does exist, there is no characterisation of the Query Tool from the formal point of view that precisely describes how such a system works and on which theoretical foundations it relies. The central purpose of this thesis is indeed that of providing a formal framework in which the Query Tool's components and operations are defined in a precise and unambiguous way. We will thus explain what a query is and how it is internally represented by the Query Tool, which operations are available to the user in order to modify the query and how the tool provides contextual feedback about it presenting only relevant pieces of information.

It is important to point out that our work does not merely consist in the formalisation of already existing ideas, since some of those have been further refined, as well as new ones introduced. For instance, we extended the portion of the query on which the user operates, by allowing different types of complex selections. Moreover, we investigated in detail how a query can be represented in a suitable "linear form", so that it can more easily be expressed in natural language. In fact, we introduced the notion of query "linearisation", that is, a sequence of "labels" satisfying certain constraints, which facilitates the textual representation of the query by guiding the generation of natural language.

As a conclusive part of our work, a new implementation of the Query Tool, superseding the existing one and complying with the formal specification, has been devised. At present, the new system includes only the core of the Query Tool and is meant to provide a demonstrative though fully functional implementation based on our framework.

1.4 Structure of the Thesis

The rest of this thesis comprises four additional chapters, which are organised as follows. Chapter 2 introduces the reader to the mathematical tools we will use, along with the corresponding notation. Chapter 3 formally defines the Query Tool framework from a theoretical point of view and precisely describes its functional Application Programming Interface (API) by specifying the abstract operations it consists of. It also presents an approach to the representation of a query in "linear form".

Chapter 4 is dedicated to an experimental implementation, that is based on the definitions given in Chapter 3. Finally, in Chapter 5, the achieved results are summarised and considered as the starting point of further development and research.

CHAPTER 2

Preliminaries

To formally define the Query Tool, we will make use of some mathematical notions, that we opportunely present in this chapter. Along with the definitions, we also introduce the notation and conventions used throughout the thesis.

We assume the reader to be familiar with logic and have some background knowledge about query answering. For an introduction to the syntax and semantics of predicate logic we suggest [19], while [1] is an extensive and excellent reference on database theory. In what follows, we will make use of a first-order relational vocabulary, that is, with no function symbols.

2.1 Conjunctive Queries

In first-order logic, a *query* is a formula ϕ with free variables $x_1, \ldots, x_k$, which we write as $\phi(x_1, \ldots, x_k)$. Given an interpretation $\mathcal{I}$ and a variable assignment α, the *answer* to a query $\phi(x_1, \ldots, x_k)$ is defined as follows:

$$\phi(x_1, \ldots, x_k)^{\mathcal{I}} = \{\langle \alpha(x_1), \ldots, \alpha(x_k) \rangle \mid \mathcal{I}, \alpha \models \phi(x_1, \ldots, x_k)\} \ . \tag{2.1}$$

The *conjunctive queries* (CQs) are the fragment of first-order logic consisting of all the formulas that can be constructed from atomic formulas (i.e., atoms and equalities) using only conjunction ($\wedge$) and existential quantification ($\exists$). Each of these formulas can be efficiently rewritten in *prenex normal form*, hence CQs are usually assumed to be of the following general form:

$$Q(x_1, \ldots, x_k) = \exists x_{k+1}, \ldots, x_n \ . \ F_1 \wedge \cdots \wedge F_m \ , \tag{2.2}$$

where $F_1, \ldots, F_m$ are atomic formulas in the variables $x_1, \ldots, x_n$. Using $\vec{x}$ for $x_1, \ldots, x_k$ and $\vec{y}$ for $x_{k+1}, \ldots, x_n$, we can write (2.2) more compactly as

$$Q(\vec{x}\,) = \exists \vec{y} \ . \ \mathrm{conj}(\vec{x}, \vec{y}\,) \ , \tag{2.3}$$

with $\mathrm{conj}(\vec{x}, \vec{y}\,) = F_1 \wedge \cdots \wedge F_m$.

Besides their logical notation, CQs can also be written as *datalog* rules. In datalog notation, (2.3) becomes

$$Q(\vec{x}\,') \leftarrow \mathrm{conj}'(\vec{x}\,', \vec{y}\,') \ , \tag{2.4}$$

where $\mathrm{conj}'(\vec{x}\,', \vec{y}\,') = A_1, \ldots, A_r$ is the list of atoms in $\mathrm{conj}(\vec{x}, \vec{y})$ obtained by equating the variables $x_1, \ldots, x_n$ according to the equalities in the conjunction. Note that, as a result of such equality elimination, we have that $\vec{x}\,'$ and $\vec{y}\,'$ may actually contain constants and multiple occurrences of the same variable. The variables in $\vec{x}\,'$ are called the *distinguished* variables of Q, while the ones in $\vec{y}\,'$ are *non-distinguished*. CQs without distinguished variables are called *Boolean*. Moreover, we call $Q(\vec{x}\,')$ the *head* of Q and $\mathrm{conj}'(\vec{x}\,', \vec{y}\,')$ its *body*. Note also that, although there are no quantifiers in datalog notation, variables appearing only in the body of the rule are implicitly considered to be existentially quantified, while those in the head are free.

The vast majority of queries that are most frequently issued on relational databases are CQs, corresponding to the *select-project-join* queries (i.e., not using the union or difference operations) in relational algebra. Conjunctive queries also correspond to the *select-from-where* SQL queries in which the "where" condition uses exclusively conjunctions of atomic equalities (that is, conditions constructed from column names and constants, using no comparison operators other than "=" and combined using "and").

Many interesting problems that are computationally hard or undecidable for larger classes of queries are instead feasible in the case of conjunctive queries, making them one of the great success stories of database theory. For instance, consider the problem of *query containment* that given two FOL queries ϕ and ψ consists in checking whether, for all interpretations $\mathcal{I}$ and all assignments α, we have that $\mathcal{I}, \alpha \models \phi$ implies $\mathcal{I}, \alpha \models \psi$. This problem, of special interest in query optimisation, is undecidable for FOL (and for SQL and relational algebra), but it is decidable and NP-complete for conjunctive queries [5].

2.2 Binary Relations, Orders, and Functions

In general, a *binary relation* R is defined as an ordered triple (A, B, G), where A and B are (arbitrary) sets and G is a subset of the Cartesian product $A \times B$. The sets A and B are respectively called the *domain* and *codomain* of R, and G is its *graph*. In order to simplify the notation, we identify a binary relation with its graph, by considering a binary relation R with domain A and codomain B simply as a subset of $A \times B$. We say that an element $x \in A$ is *R-related* to an element $y \in B$ if $\langle x, y \rangle \in R$, written indifferently as xRy or $R(x, y)$. For two binary relations R and S with the same domain and codomain, their *union* $R \cup S$ and *intersection* $R \cap S$ are given by the corresponding general operations on sets. An important operation between two binary relations $R \subseteq A \times B$ and $S \subseteq B \times C$ is the *composition* of S with R, defined as follows:

$$S \circ R := \{\langle x, z \rangle \mid \exists y \in B \text{ such that } \langle x, y \rangle \in R \text{ and } \langle y, z \rangle \in S\} \quad . \tag{2.5}$$

Note that, differently from union and intersection, the composition of two binary relations is not commutative, that is, in general $S \circ R \neq R \circ S$.

A binary relation over a set A is a subset of $A \times A$, that is, a set of ordered pairs of elements of A. Some important classes of binary relations over a set A are:

- *reflexive*: every element of A is R-related to itself;
- *irreflexive* (or *strict*): every element of A is not R-related to itself;

Property	**Condition** (in FOL)
reflexivity	$\forall x \in A \,.\, R(x,x)$
irreflexivity/strictness	$\forall x \in A \,.\, \neg R(x,x)$
symmetry	$\forall x,y \in A \,.\, R(x,y) \rightarrow R(y,x)$
antisymmetry	$\forall x,y \in A \,.\, R(x,y) \wedge R(y,x) \rightarrow x = y$
asymmetry	$\forall x,y \in A \,.\, R(x,y) \rightarrow \neg R(y,x)$
transitivity	$\forall x,y,z \in A \,.\, R(x,y) \wedge R(y,z) \rightarrow R(x,z)$
totality	$\forall x,y \in A \,.\, R(x,y) \vee R(y,z)$
trichotomy	$\forall x,y \in A \,.\, R(x,y) \oplus R(y,x) \oplus x = y$ [1]

Table 2.1: Most common properties of binary relations

- *symmetric*: if x is R-related to y, then also y is R-related to x;
- *antisymmetric*: if x and y are R-related to each other, then $x = y$;
- *asymmetric*: if x is R-related to y, then y is not R-related to x;
- *transitive*: if x is R-related to y and y is R-related to z, then x is R-related to z;
- *total*: for each pair of elements x and y, we have that x is R-related to y or vice versa (or both);
- *trichotomous*: for each pair of elements x and y, exactly one of xRy, yRx or $x = y$ holds.

The properties corresponding to the above classes of binary relations are summarised in Table 2.1, along with the associated condition (expressed by a FOL formula) that must be satisfied. Given a binary relation

Given a binary relation R over a set A, we call the smallest reflexive relation over A containing R the *reflexive closure* of R, denoted by $R^=$, while the largest irreflexive relation over A contained in R is its *reflexive reduction*, written as $R^{\neq}$. The smallest transitive relation over A containing R is called the *transitive closure* of R, denoted by R^+, and the minimal relation over A having the same transitive closure as R is its *transitive reduction* R^-. Lastly, the relation $(R^+)^=$ is called the *reflexive transitive closure* of R and is denoted by R^*.

A reflexive, symmetric and transitive binary relation is called an *equivalence relation*, while one that is reflexive, antisymmetric and transitive is a *non-strict* (or *weak*, or *reflexive*) *partial order*, as opposed to a *strict* (or *irreflexive*) partial order, which is irreflexive and transitive (thus also asymmetric). Strict partial orders are of special interest because, as we shall see in the next section, they are closely related to directed acyclic graphs. A non-/strict partial order that is also total is called a non-/strict *total* (or *linear*) *order*. Table 2.2 shows which properties are satisfied by partial and total orders in both of their strict and non-strict flavours.

Given a binary relation R over a set A, the *restriction* of R to a set $X \subseteq A$ is defined as the set $R_{|X}$ of all pairs $\langle x,y \rangle \in R$ for which $x, y \in S$. If a binary relation satisfies one (or more) of the properties in Table 2.1, then its restriction

[1]For two FOL formulas ϕ and ψ, we use $\phi \oplus \psi$ as an abbreviation for $(\phi \vee \psi) \wedge \neg(\phi \wedge \psi)$.

	partial order		total order	
	non-strict	strict	non-strict	strict
reflexive	✓		✓	
irreflexive		✓		✓
antisymmetric	✓		✓	
asymmetric		✓		✓
transitive	✓	✓	✓	✓
total			✓	✓
trichotomous				✓

Table 2.2: Strict and non-strict partial/total orders

does too. Thus, the restriction of a partial (total) order is a partial (total) order as well.

Note that for any strict binary relation there is a corresponding non-strict one given by its reflexive closure and, similarly, for any non-strict binary relation there is a corresponding strict one given by its reflexive reduction. Hence, it is irrelevant whether one works with strict or non-strict partial/total orders, on condition of not mixing them. In what follows, the symbol "$\preccurlyeq$" will usually indicate a weak partial order, while we will use "$\prec$" if the partial order is strict. Similarly, we will denote a strict total order using the symbol "$<$" and a non-strict one using "$\leqslant$".

Proposition 2.1 *Let $\leqslant_A$ and $\leqslant_B$ be (possibly strict) total orders on sets A and B, respectively, such that $\leqslant_A \subseteq \leqslant_B$. Then, $x \leqslant_A y$ if and only if $x \leqslant_B y$, for all $x, y \in A \cap B$.*

Proof. The "only if" part holds trivially, since $x \leqslant_A y$ implies $x \leqslant_B y$ for every $x, y \in A$, as $\leqslant_A \subseteq \leqslant_B$.

For the opposite direction, let $x, y \in A \cap B$, assume $x \leqslant_B y$ and suppose $x \nleqslant_A y$. Then, $y \leqslant_A x$ must hold, as $\leqslant_A$ is a total order on A. Since $\leqslant_A \subseteq \leqslant_B$, we also have $y \leqslant_B x$, hence $x = y$ by the antisymmetry of $\leqslant_B$. Therefore, as $\leqslant_A$ is reflexive, we obtain $x \leqslant_A y$, in contradiction with our initial assumption. $\square$

The informal meaning of the following lemma is that there is exactly one way to extend a totally ordered set by inserting a new element "immediately after" any of the existing ones. Before stating the lemma, we define the relation of *immediate precedence*, associated with a strict total order $<$, as follows:

$$\lessdot := \{\langle x, z\rangle \in A \mid x < z \text{ and } \nexists y \in A \text{ such that } x < y < z\} \quad . \tag{2.6}$$

Lemma 2.1 *Let $<_A$ be a strict total order on a set A, and let $B = A \cup \{b\}$ for some $b \notin A$. Then, for each $a \in A$, there exists one and only one strict total order $<_B$ on B such that $a \lessdot_B b$ and $<_A \subseteq <_B$.*

Proof. Take $a \in A$ and let

$$<_B := <_A \cup \{\langle x, b\rangle \mid x = a \text{ or } x <_A a\} \cup \{\langle b, x\rangle \mid a <_A x\} \quad . \tag{2.7}$$

By construction, $<_B$ is a binary relation on B containing $<_A$. Moreover, it is trivial to see that we have the following:

$$x <_B b \iff x = a \text{ or } x <_A a \ ; \tag{2.8a}$$

$$b <_B x \iff a <_A x \ ; \tag{2.8b}$$

$$x <_B y \iff x <_A y \ ; \tag{2.8c}$$

for all $x, y \in A$. To prove that $<_B$ is a strict total order on B, we must show that $<_B$ as in (2.7) is irreflexive, transitive and total.

Proving that $<_B$ is irreflexive means showing that for all $x \in B$ it is never the case that $x <_B x$. This follows immediately from the irreflexivity of $<_A$ and (2.8c), for the elements of A, and from the fact that $\langle b, b\rangle \notin <_B$ by construction.

Proving that $<_B$ is total means showing that any two elements of B are comparable under $<_B$. We distinguish the following cases.

Case 1. Let $x, y \in A$. Then, x and y are comparable under $<_A$, which is total on A. Hence, by using (2.8c), we get that x and y are also comparable under $<_B$.

Case 2. Let $x \in A$ and $y = b$. If $x = a$, we have $a <_B b$ directly by (2.8a). If $x \neq a$, by the totality of $<_A$, it has to be comparable to a under $<_A$. Therefore, either by (2.8a) or (2.8b), we get that x and b are comparable under $<_B$.

Proving that $<_B$ is transitive means showing that for every $x, y, z \in B$ whenever $x <_B y$ and $y <_B z$ we also have $x <_B z$. Since $<_B$ is irreflexive, we can safely assume w.l.o.g. that $x \neq y$ and $y \neq z$. We distinguish the following cases.

Case 1. Let $x = b$ and $y, z \in A$. Then, from $b <_B y$ we have $a <_A y$ by (2.8b) and from $y <_B z$ we get $y <_A z$ by (2.8c). As $<_A$ is transitive, we obtain $a <_A z$ and in turn $b <_B z$, again by (2.8b).

Case 2. Let $y = b$ and $x, z \in A$. Then, from $b <_B z$ we have $a <_A z$ by (2.8b) and from $x <_B b$ we get that $x = a$ or $x <_A a$ by (2.8a). Either way, directly or by using the transitivity of $<_A$, we obtain $x <_A z$ and in turn $x <_B z$ by (2.8c).

Case 3. Let $z = b$ and $x, y \in A$. Then, from $x <_B y$ we have $x <_A y$ by (2.8c) and from $y <_B b$ we get that $y = a$ or $y <_A a$ by (2.8a). In both cases, either directly or by using the transitivity of $<_A$, it follows that $x <_A a$ and in turn $x <_B b$, again by (2.8a).

Case 4. Let $x = z = b$ and $y \in A$. Then, from $b <_B y$ we have $a <_A y$ by (2.8b) and from $y <_B b$ we get that $y = a$ or $y <_A a$ by (2.8a). Either case, directly or by using the transitivity of $<_A$, we obtain $a <_A a$, which is a contradiction of the irreflexivity of $<_A$. Therefore, $b <_B y$ and $y <_B b$ cannot both be true at the same time.

This concludes the proof that $<_B$ is a strict total order.

We will now show that $<_B$ is such that $a \lessdot_B b$. Suppose this is not the case, i.e., there is some $x \in B$ such that $a <_B x <_B b$. Then, from $a <_B x$ we have $a <_A x$ by (2.8c) and from $x <_B b$ we get $x = a$ or $x <_A a$ by (2.8a). In both cases, either directly or by the transitivity of $<_A$, it follows that $a <_A a$, which is a contradiction of the irreflexivity of $<_A$.

So far, we have only proved that there exists a strict total order, namely $<_B$ as in (2.7), including $<_A$ and such that $a \lessdot_B b$. To conclude our proof, we still have to show that such order is unique. Suppose this is not the case, i.e., there exists a strict total order $<'_B \neq <_B$ such that $<_A \subseteq <'_B$ and $a \lessdot'_B b$. This means that there are $x, y \in B$ such that $x <_B y$ and $y <'_B x$. We distinguish the following cases.

Case 1. Let $x, y \in A$. Then, from $x <_B y$ we get $x <_A y$ by (2.8c) and in turn, as $<_A \subseteq <'_B$, it follows that $x <'_B y$, in contradiction of the assumption that $<'_B$ is a strict total order.

Case 2. Let $x = b$ and $y \in A$. Then, from $b <_B y$ we have $a <_A y$ by (2.8b) and in turn it follows that $a <'_B y$, as $<_A \subseteq <'_B$. Thus, we have $a <'_B x <'_B b$, in contradiction of the assumption that $a \lessdot'_B b$.

Case 3. Let $x \in A$ and $y = b$. Then, from $x <_B b$ we get that $x = a$ or $x <_A a$ by (2.8b). If $x = a$, we directly obtain $b <'_B a$, which is a contradiction of the assumption that $a <'_B b$. If $x <_A a$ instead, we have $x <'_B a$ as $<_A \subseteq <'_B$, and by the transitivity of $<'_B$ we reach the same contradiction as before.

This concludes the proof of the lemma. □

The previous lemma deals with the insertion of a new element into a strictly and totally ordered set so that it "immediately follows" one of the elements in the set. An analogous result can be proved in the case of adding a new element "immediately before" one of the elements of a strictly and totally ordered set. We will only state this lemma and omit its proof, as it is very similar to the one already given for Lemma 2.1.

Lemma 2.2 *Let $<_A$ be a strict total order on a set A, and let $B = A \cup \{b\}$ for some $b \notin A$. Then, for each $a \in A$, there exists one and only one strict total order $<_B$ on B such that $b \lessdot_B a$ and $<_A \subseteq <_B$.*

A *function* is a binary relation associating each element in the domain with exactly one element of the codomain. The following two-part notation is used specifically for functions:

$$f \colon A \to B \qquad , \qquad \{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\} \tag{2.9}$$

where f is the function's name, $A = \{x_1, \ldots, x_k\}$ its domain, B the codomain and $\{x_1 \mapsto y_1, \ldots, x_k \mapsto y_k\}$ is a set of *mappings* associating each element x_i of A with a corresponding $y_i \in B$. In (2.9), the expression on the left is read "f is a function from A to B", while each mapping $x_i \mapsto y_i$ is read "x_i maps to y_i". A mapping can also be written as $y_i = f(x_i)$ and y_i is called the *image* of x_i under f. The concept of image can be extended to subsets of the domain by defining the image of $X \subseteq A$ as the following subset of the codomain:

$$f(X) := \{y \in B \mid y = f(x) \text{ and } x \in X\} \tag{2.10}$$

If $X = A$, the set $f(A)$ is called the *image of f* and denoted by $\mathsf{im}(f)$.

The notation in (2.9), where the mapping of each element of the domain is given explicitly, cannot be used in the case in which the domain is not finite. Thus, we introduce the following more general notation:

$$f\colon A \to B \qquad , \qquad x \mapsto \begin{cases} \text{expression in } x & \text{if condition on } x \\ \vdots & \vdots \\ \text{expression in } x & \text{if condition on } x \end{cases} \tag{2.11}$$

where the conditions on x are mutually exclusive (i.e., not conflicting with each other) and such that every element of the domain satisfies exactly one of them. Moreover, each expression in x results in an element of the codomain. The intuitive meaning is that an element of the domain is mapped to an element of the codomain resulting from the evaluation of the expression corresponding to the satisfied condition. Clearly, in the case in which the domain is finite, (2.9) can be equivalently expressed using (2.11) as follows:

$$f\colon A \to B \qquad , \qquad x \mapsto \begin{cases} y_1 & \text{if } x = x_1 \\ \vdots & \vdots \\ y_k & \text{if } x = x_k \end{cases}$$

The composition of functions and the restriction of a function are defined as for binary relations in general. The unique function over a set A that maps each element to itself is called the *identity function* for A and it is denoted by id_A. As each set has its own identity function, the subscript cannot be omitted unless the set can be inferred from the context.

We conclude the section by giving some other useful definitions that will be used later on and which also provide an example of the notation we introduced for functions in (2.11).

Definition 2.1 Let C be a set, op a binary operation between sets, and let f and g be functions such that $\mathsf{codom}(f) = \mathsf{codom}(g) = 2^C$. Then, we define:

$$\begin{aligned} & f \mathbin{\widehat{\mathsf{op}}} g : \mathsf{dom}(f) \cup \mathsf{dom}(g) \to 2^C \\ & \qquad x \mapsto \begin{cases} f(x) & x \in \mathsf{dom}(f) \setminus \mathsf{dom}(g) \ , \\ f(x) \mathbin{\mathsf{op}} g(x) & x \in \mathsf{dom}(f) \cap \mathsf{dom}(g) \ , \\ g(x) & \text{otherwise} \ . \end{cases} \end{aligned} \tag{2.12}$$

Definition 2.2 Let A, B and C be sets such that $A \cap B = \varnothing$. The *disjoint union* of functions $f : A \to C$ and $g : B \to C$ is defined as follows:

$$\begin{aligned} & f \mathbin{\dot{\cup}} g : A \cup B \to C \\ & \qquad x \mapsto \begin{cases} f(x) & x \in A \ , \\ g(x) & \text{otherwise} \ . \end{cases} \end{aligned} \tag{2.13}$$

2.3 Graphs and Trees

As we shall see in the next chapter, trees are the fundamental structure used for representing queries in the Query Tool framework. Here, we first give some basic definitions about graphs in general, following mainly [8]. Then, we will introduce a specific notation for trees, adapted to our needs.

A *directed graph* is a pair $G = (V, E)$ of disjoint sets satisfying $E \subseteq V \times V$, thus the elements of E, called *edges* (or *arcs*), are ordered pairs of the elements of V, which are the *nodes* (or *vertices*) of G. An edge $e = \langle u, v\rangle$ is said to be *directed* from the *initial node* u to the *terminal node* v, denoted by $\mathsf{init}(e)$ and $\mathsf{ter}(e)$, respectively. If $\mathsf{init}(e) = \mathsf{ter}(e)$, the edge e is called a *loop*.

The number of nodes in a graph G is its *order*, written as $|G|$, and graphs are *finite* or *infinite* according to their order. The graph with no nodes (and, therefore, no edges) is the *empty graph*, and a graph of order 0 or 1 is called *trivial*.

A node n is *incident* with an edge e if $n = \mathsf{init}(e)$ or $n = \mathsf{ter}(e)$; then e is an edge *at* n. The initial node of an edge is a *parent* of the terminal one, which is its *child*, and an edge is *incoming* (*outgoing*) w.r.t. its terminal (initial) node. The sets of incoming and outgoing edges of a node n in a graph G are respectively defined as follows:

$$E_{\mathrm{in,G}}(n) := \{e \in E(G) \mid \mathsf{ter}(e) = n\} \quad ; \tag{2.14}$$

$$E_{\mathrm{out,G}}(n) := \{e \in E(G) \mid \mathsf{init}(e) = n\} \quad . \tag{2.15}$$

The set of all the edges in $E(G)$ at a node n is given by $E_{\mathrm{in,G}}(n) \cup E_{\mathrm{out,G}}(n)$ and denoted by $E_G(n)$, or briefly by $E(n)$.[2]

Given graphs $G = (V, E)$ and $H = (U, F)$, we define their *union* and *intersection*, respectively, as follows:

$$G \cup H := (V \cup U, E \cup F) \quad ; \qquad G \cap H := (V \cap U, E \cap F) \quad .$$

If $U \subseteq V$ and $F \subseteq E$, then H is a *subgraph* of G, written as $H \subseteq G$. When $G \cap H$ is the empty graph, G and H are *disjoint*. The graph resulting from the union of disjoint graphs is called their *disjoint union*.

A *path* is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_0, x_1, \dots, x_k\} \quad , \qquad E = \{\langle x_0, x_1\rangle, \langle x_1, x_2\rangle, \dots, \langle x_{k-1}, x_k\rangle\} \quad ;$$

where $x_0, \dots, x_{k-1}$ and $x_1, \dots, x_k$ are sequences of distinct elements (i.e., only x_0 and x_k are allowed to be the same node). We say that P is a path from x_0 to x_k and refer to it by the natural sequence of its nodes, writing $P = x_0 x_1 \dots x_k$. The nodes x_0 and x_k are respectively called the *start* and the *end* of P, while $x_1, \dots, x_{k-1}$ are its *inner* nodes. In the particular case when x_0 and x_k coincide, we call P a *cycle* (or *cyclic path*). The number of edges in a path is its *length*: note that a node is a path of length 0 and a loop is a cycle of length 1. In what follows, when we speak of a path, we always refer to a "proper" path, i.e. a path of length greater than or equal to 1.

A node n is an *ancestor* of order k (or k*-ancestor* for short) of a node m, which is called a k*-descendant* of n, if there is a path of length k from n to m. Note that the 1-ancestors of a node are its parents and the 1-descendants are its children. We say that n is an ancestor (descendant) of m if n is a k-ancestor (k-descendant) of m, for some $k \in \mathbb{N}_1$. The sets of k-ancestors and

[2]Here, as elsewhere, we omit the subscript referring to the underlying graph if the reference is clear.

k-descendants of a node n in a graph G are denoted by $V^k_{\mathrm{anc,G}}(n)$ and $V^k_{\mathrm{des,G}}(n)$, respectively. Then, the sets of all ancestors and descendants of a node n in a graph G are given by

$$V_{\mathrm{anc,G}} := \bigcup_{k\in\mathbb{N}_1} V^k_{\mathrm{anc,G}} \quad ; \tag{2.16}$$

$$V_{\mathrm{des,G}} := \bigcup_{k\in\mathbb{N}_1} V^k_{\mathrm{des,G}} \quad . \tag{2.17}$$

A *directed acyclic graph* (DAG) is directed graph without cycles. Each DAG induces a strict partial order $\prec$ on its nodes, where $u \prec v$ exactly when there exists a path from u to v. Such strict partial order is the *reachability relation* of the DAG, telling whether (and how) it is possible to get from one node to another, and corresponds to the transitive closure of its edge set. Note that different DAGs may give rise to the same strict partial order: for example, $G = (\{a, b, c\}, \{\langle a, b\rangle, \langle b, c\rangle\})$ and $G' = G + \langle a, c\rangle$ have the same reachability, but G' has an additional edge. Among all such DAGs, the one with the fewest edges is the transitive reduction of each of them, while the one with the most is their transitive closure. In particular, the transitive closure has an edge $\langle u, v\rangle$ for every related pair $u \prec v$ in the reachability relation $\prec$, which it may therefore be identified with.

A *tree* is DAG in which a special node, called the *root*, has no incoming edge and every other node has exactly one parent. In a tree, a node with no children is called a *leaf.* It is easy to see that, in a tree T with root r, for each node $n \neq r$ there exists a unique path from the r to n, hence the root is an ancestor of every other node in the tree. The unique path from the root to a node n is denoted by $P_T(n)$ and its length is the *depth* of n, denoted by $d(n)$. The depth of a tree T is the maximum among the depths of its nodes, that is, $d(T) := \max(\{d(n) \mid n \in V(T)\}$. Clearly, the depth of the root is 0 and each node n is a $d(n)$-descendant of the root, which is an ancestor of every other node in the tree.

The *tree-order* of a tree T rooted in r is the partial order induced on $V(T)$ associated with T and r. Note that r is the least element in this partial order, every leaf $x \neq r$ of T is a maximal element, the ends of any edge of T are comparable, and every set of the form $\{x \mid x \prec y\}$ (where y is any fixed node) is a *chain*, a set of pairwise comparable elements.

PROPOSITION 2.2 *Let $T = (V, E)$ and $S = (U, F)$ be directed trees rooted in t and s, respectively. If $V \cap U = \{s\}$, then the graph $T \cup S = (V \cup U, E \cup F)$ is a tree rooted in t.*

Proof. In order to prove that $G = T \cup S$ is a tree rooted in t, it suffices to show that for every node $n \in V(G) \setminus \{t\}$ there exists a unique path leading from t to n. If $n \in V$, then such unique path exists by assumption, as T is a tree. Let us now consider the case in which $n \in U$. If $n = s$ we are again in the previous case, so let $n \neq s$. By assumption, there is a unique path P_T from t to s in T and a unique path P_S from s to n in S. Since both P_T and P_S are also paths in G, by joining them we obtain a path P from t to n. As the only common node between T and S is s, there can be no path from t to n in G not going through s, therefore P is unique in G. □

2.4 Description Logics

Description Logics (DL) are a well-known family of knowledge representation languages that can be used to represent an application domain in a structured and formally well-understood way.

The DL syntax can be given using an abstract language, similar to other logical formalisms, where two disjoint alphabets of symbols are used to denote *atomic concepts* and *atomic roles*, designated by unary and binary predicate symbols, respectively. In particular, the latter are used to express relationships between concepts and terms are then built from the basic symbols using several kinds of constructors. Some common concept constructors include intersection (or conjunction) of concepts, union (or disjunction) of concepts, negation (or complement) of concepts, value restriction (universal quantification), existential quantification, to name the most commonly used ones. Other constructors may also include restrictions on roles which are usual for binary relations, such as inverse, transitivity, functionality, among others.

As for the DL semantics, this is given using a set-theoretic interpretation: a concept is interpreted as a set of individuals and a role is interpreted as a set of pairs of individuals. The domain of interpretation can be chosen arbitrarily and can be infinite. Atomic concepts are interpreted as subsets of the interpretation domain, while the semantics of the other constructs is then specified by defining the set of individuals denoted by each construct.

The basic DL language is the *Attributive Language* $\mathcal{AL}$, whose syntax is defined by the following grammar:

$$C, D \rightarrow \neg A \mid C \sqcap D \mid \forall R.C \mid \exists R \enspace ;$$

in which C and D are (general) concepts, R is an atomic role and A denotes an atomic concept. Thus, $\mathcal{AL}$ is the language allowing for atomic negation (i.e., negation only in front of atomic concepts), concept intersection, value restriction and limited existential quantification, in which concepts are constructed according to the following:

$$\begin{aligned} \langle\text{concept}\rangle ::= \ & \neg\langle\text{atomic-concept}\rangle \mid \langle\text{concept}\rangle \sqcap \langle\text{concept}\rangle \mid \\ & \forall\langle\text{atomic-role}\rangle.\langle\text{concept}\rangle \mid \exists\langle\text{atomic-role}\rangle \enspace . \end{aligned}$$

There is a naming convention that indicates which operators are allowed in each of the many different varieties of DL. In particular, the expressivity of a DL language[3] is encoded in its name using the following letters:

- $\mathcal{F}$ functional properties;
- $\mathcal{E}$ full existential qualification (i.e., existential restrictions with fillers other than $\top$);
- $\mathcal{U}$ concept union;
- $\mathcal{C}$ complex concept negation;
- $\mathcal{S}$ an abbreviation for $\mathcal{ALC}$ with transitive roles;

[3]Apart from a few exceptions like $\mathcal{AL}$, $\mathcal{FL}^-$ and $\mathcal{EL}$, which do not fit in this naming scheme.

- $\mathcal{H}$ role hierarchy (i.e., subproperties);
- $\mathcal{R}$ limited complex role inclusion axioms; reflexivity and irreflexivity; role disjointness.
- $\mathcal{O}$ nominals (i.e., enumerated classes of object value restrictions);
- $\mathcal{I}$ inverse properties;
- $\mathcal{N}$ cardinality restrictions;
- $\mathcal{Q}$ qualified cardinality restrictions (i.e., cardinality restrictions with fillers other $\top$);
- $^{(\mathcal{D})}$ use of datatype properties, data values or data types.

As an example of the above naming scheme, $\mathcal{ALC}$ is the DL language obtained from $\mathcal{AL}$ by allowing negation in front of any concept (not just atomic ones).

The ontology language adopted by the Query Tool is OWL 2, which is expressive enough for our purposes, and for which there are state of the art reasoners. OWL 2 has the expressiveness of the DL language $\mathcal{SROIQ}^{\mathcal{D}}$, though we did not fully exploit it. In the rest of this section, we will describe the syntax and semantics of $\mathcal{SROIQ}$. Our presentation is based on [17], where this language was first introduced. A $\mathcal{SROIQ}$ knowledge base is a triple $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$, where $\mathcal{T}$ is a *terminological box* (*Tbox*) of statements about concepts, $\mathcal{R}$ is a *role box* (*Rbox*) of statements about roles and $\mathcal{A}$ is an *assertion box* (*Abox*) of assertions about individuals. In what follows, we analyse each component in detail.

Let C be a set of *concept names* including a subset N of *nominals*, R a set of *role names* and $\mathsf{I} = \{a, b, c, \dots\}$ a set of *individual names*. A *role* is an element of the set $\mathsf{R} \cup \{R^- \mid R \in \mathsf{R}\}$, where a role R^- is called the *inverse role* of R. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty (possibly infinite) set $\Delta^{\mathcal{I}}$, called the *domain* of $\mathcal{I}$, and a *valuation* $\cdot^{\mathcal{I}}$ associating each role name $R \in \mathsf{R}$ with a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, each concept name $C \in \mathsf{C}$ with a subset $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, where $C^{\mathcal{I}}$ contains exactly one element if $C \in \mathsf{N}$, and each individual name $a \in \mathsf{I}$ with an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. Inverse roles are interpreted as follows:

$$(R^-)^{\mathcal{I}} = \left\{\langle x, y\rangle \mid \langle y, x\rangle \in R^{\mathcal{I}}\right\} \quad .$$

In order to avoid considering roles of the form R^{--}, we define a function Inv on roles such that $\mathsf{Inv}(R) = R^-$ if R is a role name, and $\mathsf{Inv}(R) = S \in \mathsf{R}$ if $R = S^-$. An expression of the form $w = R_1 \dots R_n$ is a string of roles, for which we set $\mathsf{Inv}(w) = \mathsf{Inv}(R_1) \dots \mathsf{Inv}(R_n)$ and $w^{\mathcal{I}} = R_1^{\mathcal{I}} \circ \cdots \circ R_n^{\mathcal{I}}$, where $\circ$ denotes composition of binary relations.

A (generalised) *role inclusion axiom* (RIA) is an expression of the form $w \sqsubseteq R$, where w is a finite string of roles and $R \in \mathsf{R}$. A *role hierarchy* is a finite set of RIAs. An interpretation $\mathcal{I}$ satisfies a RIA $w \sqsubseteq R$ (written $\mathcal{I} \models w \sqsubseteq R$) if $w^{\mathcal{I}} \subseteq R^{\mathcal{I}}$, and it is a *model* of a role hierarchy $\mathcal{R}_h$ (in symbols $\mathcal{I} \models \mathcal{R}_h$) if it satisfies all RIAs in it. Given a role hierarchy $\mathcal{R}_h$, the relation $\sqsubseteq^*$ denotes the reflexive-transitive closure of $\sqsubseteq$ over the set $\{R \sqsubseteq S,\ \mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}_h\}$.

For role names R and S, a *role assertion* is one of the following expressions:

$$\mathsf{Ref}(R), \quad \mathsf{Irr}(R), \quad \mathsf{Sym}(R), \quad \mathsf{Asy}(R), \quad \mathsf{Tra}(R), \quad \mathsf{Dis}(R, S) \ ,$$

stating that R has to be interpreted, respectively, as reflexive, irreflexive, symmetric, asymmetric and transitive, and that R and S are disjoint. For each interpretation $\mathcal{I}$ and all $x, y, z \in \Delta^{\mathcal{I}}$, we have:

$$\begin{array}{ll}
\mathcal{I} \models \mathsf{Ref}(R) & \text{if } \{\langle x,x\rangle \mid x \in \Delta^{\mathcal{I}}\} \subseteq R^{\mathcal{I}} \ ; \\
\mathcal{I} \models \mathsf{Irr}(R) & \text{if } \{\langle x,x\rangle \mid x \in \Delta^{\mathcal{I}}\} \cap R^{\mathcal{I}} = \varnothing \ ; \\
\mathcal{I} \models \mathsf{Sym}(R) & \text{if } \langle x,y\rangle \in R^{\mathcal{I}} \text{ implies } \langle y,x\rangle \in R^{\mathcal{I}} \ ; \\
\mathcal{I} \models \mathsf{Asy}(R) & \text{if } \langle x,y\rangle \in R^{\mathcal{I}} \text{ implies } \langle y,x\rangle \notin R^{\mathcal{I}} \ ; \\
\mathcal{I} \models \mathsf{Tra}(R) & \text{if } \langle x,y\rangle \in R^{\mathcal{I}} \text{ and } \langle y,x\rangle \in R^{\mathcal{I}} \text{ implies } \langle x,z\rangle \in R^{\mathcal{I}} \ ; \\
\mathcal{I} \models \mathsf{Dis}(R,S) & \text{if } R^{\mathcal{I}} \cap S^{\mathcal{I}} = \varnothing \ .
\end{array}$$

Transitive and symmetric role assertions can be replaced by complex role inclusion axioms. In particular, $\mathsf{Tra(R)}$ is equivalent to $RR \sqsubseteq R$ and $\mathsf{Sym}(R)$ is equivalent to $R^{-} \sqsubseteq R$.

A role is *simple* if it is the inverse of a simple role or it occurs only in the RHS of RIAs whose LHS is a simple role (thus, role names not occurring in the RHS of any RIA are simple). Intuitively, non-simple roles are those that are implied by the composition of roles.

A $\mathcal{SROIQ}$-Rbox is a set $\mathcal{R} = \mathcal{R}_h \cup \mathcal{R}_a$, where $\mathcal{R}_h$ is a *regular*[4] role hierarchy and $\mathcal{R}_a$ is a finite set of role assertions about simple roles only. An interpretation $\mathcal{I}$ is a model of $\mathcal{R}$ (written $\mathcal{I} \models \mathcal{R}$) if $\mathcal{I} \models \mathcal{R}_h$ and $\mathcal{I} \models \phi$ for all role assertions $\phi \in \mathcal{R}_a$.

The *$\mathcal{SROIQ}$-concepts* are constructed according to the following grammar:

$$\begin{aligned} C, D \to \top \mid \bot \mid A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \forall R.C \mid \exists R.C \mid \\ \exists S.\,\mathsf{Self} \mid (\geqslant n\ S.C) \mid (\leqslant n\ S.C) \end{aligned} \tag{2.18}$$

where C and D are concepts, A is a concept name, R is a role, S is a simple role and n is a non-negative integer. As for the semantics, for each interpretation $\mathcal{I}$, the extension of complex concepts is defined as follows:

$$\begin{aligned}
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \ ; \\
\bot^{\mathcal{I}} &= \varnothing \ ; \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \ ; \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \ ; \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \ ; \\
(\exists R.C)^{\mathcal{I}} &= \{x \mid \text{there is } y \in C^{\mathcal{I}} \text{ such that } \langle x,y\rangle \in R^{\mathcal{I}}\} \ ; \\
(\exists R.\,\mathsf{Self})^{\mathcal{I}} &= \{x \mid \langle x,x\rangle \in R^{\mathcal{I}}\} \ ; \\
(\forall R.C)^{\mathcal{I}} &= \{x \mid y \in C^{\mathcal{I}} \text{ for all } \langle x,y\rangle \in R^{\mathcal{I}}\} \ ; \\
(\geqslant n\ R.C)^{\mathcal{I}} &= \{x \mid \#\{y \in C^{\mathcal{I}} \mid \langle x,y\rangle \in R^{\mathcal{I}}\} \geq n\} \ ; \\
(\leqslant n\ R.C)^{\mathcal{I}} &= \{x \mid \#\{y \in C^{\mathcal{I}} \mid \langle x,y\rangle \in R^{\mathcal{I}}\} \leq n\} \ ;
\end{aligned}$$

[4]*Regularity* is a requirement that prevents a role hierarchy from containing cyclic dependencies, which are known to cause undecidability, and whose definition involves a certain ordering on roles. See [17] for the details.

where $\#M$ denotes the number of elements in a set M.

A *general concept inclusion axiom* (GCI) is an expression of the form $C \sqsubseteq D$ for two $\mathcal{SROIQ}$-concepts C and D; a $\mathcal{SROIQ}$-Tbox is a finite set $\mathcal{T}$ of GCIs. An interpretation $\mathcal{I}$ is a model of a Tbox $\mathcal{T}$ (written ($\mathcal{I} \models \mathcal{T}$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for each GCI $C \sqsubseteq D$ in $\mathcal{T}$. A concept C is *satisfiable* if there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \varnothing$. For an interpretation $\mathcal{I}$, an element $x \in \Delta^{\mathcal{I}}$ is an *instance* of a concept C if $x \in C^{\mathcal{I}}$.

An *individual assertion* is one of the following expressions:

$$a : C, \quad (a, b) : R, \quad (a, b) : \neg R, \quad a \neq b \ ;$$

where $a, b \in \mathsf{I}$ are individuals, R is a role and C is a concept. A $\mathcal{SROIQ}$-Abox is a finite set $\mathcal{A}$ of individual assertions. An interpretation $\mathcal{I}$ is a model of an Abox $\mathcal{A}$ (written $\mathcal{I} \models \mathcal{A}$) if $\mathcal{I} \models \phi$ for all individual assertions $\phi \in \mathcal{A}$, where we have:

$$\begin{array}{ll}
\mathcal{I} \models a : C & \text{if } a^{\mathcal{I}} \subseteq C^{\mathcal{I}} \ ; \\
\mathcal{I} \models (a, b) : R & \text{if } \langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in R^{\mathcal{I}} \ ; \\
\mathcal{I} \models (a, b) : \neg R & \text{if } \langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \notin R^{\mathcal{I}} \ ; \\
\mathcal{I} \models a \neq b & \text{if } a^{\mathcal{I}} \neq b^{\mathcal{I}} \ .
\end{array}$$

An Abox $\mathcal{A}$ is *consistent* w.r.t. a Tbox $\mathcal{T}$ and an Rbox $\mathcal{R}$ if there is a model $\mathcal{I}$ for $\mathcal{T}$ and $\mathcal{R}$ such that $\mathcal{I} \models \mathcal{A}$.

In a knowledge base $\mathcal{K}$, a concept D *subsumes* a concept C (written $\mathcal{K} \models C \sqsubseteq D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every interpretation $\mathcal{I}$. Two concepts C and D are *equivalent* in $\mathcal{K}$ (in symbols $\mathcal{K} \models C \equiv D$) if $\mathcal{K} \models C \sqsubseteq D$ and $\mathcal{K} \models D \sqsubseteq C$. For atomic concepts A and B, we say that B is a *direct super-concept* of A, which is then a *direct sub-concept* of B, if $\mathcal{K} \not\models A \equiv B$, $\mathcal{K} \models A \sqsubseteq B$ and there is no $C \in \mathsf{C}$ subsuming A and subsumed by B.

CHAPTER 3

Theoretical Foundations

In this chapter, we formally define the Query Tool and its functional API, using the mathematical tools we previously introduced in Chapter 2. Since the most direct and perhaps most effective way to understand the overall meaning of a new notion is often by means of examples, the vast majority of the definitions we will give is accompanied by a supporting example. Therefore, before presenting any formal definition, we first introduce a simple ontology on which all of the examples in this chapter are based.

We consider a $\mathcal{SROIQ}$ ontology $\mathcal{K} = \big(\mathcal{T}, \mathcal{R}, \varnothing\big)$, with an empty Abox (i.e., no individual assertions), over the following sets C of concept names, R of role names and I of individual names:

$$\begin{aligned}
\mathsf{C} &= \{\mathsf{Beautiful}, \mathsf{Ugly}, \mathsf{House}, \mathsf{Person}, \mathsf{Man}, \mathsf{Woman}, \mathsf{RichPerson}, \mathsf{SinglePerson}\}\\
\mathsf{R} &= \{\mathsf{inhabitedBy}, \mathsf{ownedBy}, \mathsf{owns}, \mathsf{livesIn}, \mathsf{marriedTo}\} \;\; ; \text{ and}\\
\mathsf{I} &= \varnothing \;\; ;
\end{aligned}$$

where $\mathcal{T}$ consists of the following GCIs:

$$\begin{aligned}
\mathsf{Ugly} \sqcap \mathsf{Beautiful} &\sqsubseteq \bot & \mathsf{House} \sqcap \mathsf{Person} &\sqsubseteq \bot\\
\mathsf{Man} &\sqsubseteq \mathsf{Person} & \mathsf{Woman} &\sqsubseteq \mathsf{Person}\\
\mathsf{RichPerson} &\sqsubseteq \mathsf{Person} & \mathsf{SinglePerson} &\sqsubseteq \mathsf{Person}\\
\exists\, \mathsf{marriedTo}\,.\top &\sqsubseteq \mathsf{Person} & \exists\, \mathsf{marriedTo}^-\,.\top &\sqsubseteq \mathsf{Person}\\
\exists\, \mathsf{owns}\,.\top &\sqsubseteq \mathsf{Person} & \exists\, \mathsf{owns}^-\,.\top &\sqsubseteq \mathsf{House}\\
\exists\, \mathsf{livesIn}\,.\top &\sqsubseteq \mathsf{Person} & \exists\, \mathsf{livesIn}^-\,.\top &\sqsubseteq \mathsf{House}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Man} &\sqsubseteq \forall\, \mathsf{marriedTo}\,.\,\mathsf{Woman} & \mathsf{Woman} &\sqsubseteq \forall\, \mathsf{marriedTo}\,.\,\mathsf{Man}\\
\mathsf{Person} &\sqsubseteq (\leqslant 1\, \mathsf{marriedTo}\,.\,\mathsf{Person}) & \mathsf{Person} &\sqsubseteq (\leqslant 1\, \mathsf{livesIn}\,.\,\mathsf{House})\\
\mathsf{Man} &\equiv \mathsf{Person} \sqcap \neg\, \mathsf{Woman}\,^{1} & \mathsf{RichPerson} &\sqsubseteq \exists\, \mathsf{owns}\,.\,\mathsf{House}
\end{aligned}$$

$$\begin{aligned}
\mathsf{SinglePerson} &\equiv \mathsf{Person} \sqcap \neg(\exists\, \mathsf{marriedTo}\,.\,\mathsf{Person})\\
\mathsf{RichPerson} &\sqsubseteq \forall\, \mathsf{livesIn}\,.(\mathsf{House} \sqcap \mathsf{Beautiful}) \;\; ;
\end{aligned}$$

and $\mathcal{R} = (\mathcal{R}_h, \mathcal{R}_a)$, with:

$$\begin{aligned}\mathcal{R}_h &= \{\mathsf{inhabitedBy} \equiv \mathsf{livesIn}^-, \mathsf{ownedBy} \equiv \mathsf{owns}^-\} \ ;\\ \mathcal{R}_a &= \{\mathsf{Sym}\ (\mathsf{marriedTo})\} \ .\end{aligned}$$

Informally, the given ontology expresses the following information: nothing can be both a **house** and a **person**, nor can it be **ugly** and **beautiful** at the same time, although there might be, for example, beautiful persons and ugly houses, as well as beautiful things that are neither houses nor persons, and so on. A **man** is a person and so is a **woman**, and a person is either a man or a woman.[2] Persons (and only persons) can be married and, in that case, they are **married to** at most another person. However, men can only be married to women and vice versa. Furthermore, as the role marriedTo is asserted to be symmetric, if person A is married to person B, then B is married to A. Some persons are **rich** and some are **single**, and the latter are those who are not married to anyone. Persons **live in** houses they may **own**, but a person can live in at most one house. However, a house might be **inhabited by** more than one person. If a person owns a house, then the latter is **owned by** that person. Rich persons always own (at least) one house and they live in a beautiful house (which may be different from the one they own).

Suppose we want to formulate the following conjunctive query over the set of unary (concept names) and binary predicates (role names) of our sample ontology:

$$\begin{aligned}Q(x) \leftarrow &\,\mathrm{SinglePerson}(x), \mathrm{Man}(x), \mathrm{marriedTo}(x, y), \mathrm{Woman}(y)\\ &\mathrm{owns}(x, w), \mathrm{Beautiful}(w), \mathrm{House}(w)\\ &\quad\mathrm{inhabitedBy}(w, z), \mathrm{RichPerson}(z) \ .\end{aligned} \tag{3.1}$$

It should be immediately clear, from the constraints in the ontology, that the above query is not satisfiable, since a single person is one who is not married. In what follows, we will show how such a query might in principle be represented in the framework we are about to introduce, but it cannot indeed be formulated using the operations in the Query Tool's functional API.

3.1 Formal Definition of the Framework

We represent a conjunctive query without co-references as a directed labelled tree, where each node is associated with a non-empty set of concept names and each edge is associated with exactly one role name.

Definition 3.1 (Query) Let $\mathbf{N}$ be a countable set of node names, C a finite set of concept names and R a finite set of role names, and let $\mathbf{N}$, C and R be pairwise disjoint. A *query* Q is a quintuple $\langle V, E, o, \mathcal{V}, \mathcal{E}\rangle$ where:

- (V, E) is a directed tree rooted in $o \in V$, in which $V \subseteq \mathbf{N}$ is the set of nodes and $E \subseteq V \times V$ is the set of (directed) edges;

[1] With some abuse of notation we write $C \equiv D$ as an abbreviation for the two GCIs $C \sqsubseteq D$ and $D \sqsubseteq C$. We use a similar shortcut also for RIAs.

[2] This follows from the fact that a man is exactly a person who is not a woman.

- $\mathcal{V} : V \longrightarrow \big(2^{\mathsf{C}} \setminus \{\varnothing\}\big) \cup \{\{\top\}\}$ is a total function, called *node-labelling function*, which associates each node with a non-empty set of concept names (including the singleton $\{\top\}$); and
- $\mathcal{E} : E \longrightarrow \mathsf{R}$ is a total function, called *edge-labelling function*, associating each edge with a role name.

A query is *atomic query* if $V = \{o\}$, $E = \varnothing$ and $|\mathcal{V}(o)| = 1$.

As with graphs and trees in general, we refer to the node set of a query Q as $V(Q)$ and to its edge set as $E(Q)$, independently of the actual name of the two sets. We also assume that, unless explicitly stated otherwise, the remaining components of a query (i.e., its root, its node-labelling function and its edge-labelling function) are identified using the name of the query as subscript. For instance, for queries Q and R, we refer to the roots of Q and R as o_Q and o_R, respectively; similarly, their respective edge-labelling functions are indicated with $\mathcal{E}_Q$ and $\mathcal{E}_R$, and so on.

Not surprisingly, since a query is a directed labelled tree, a subquery is a directed labelled subtree, where the set of labels associated with each node is a subset of the labels associated with that node in the original tree. As for the edge labels, each edge in the subtree is obviously labelled by the only label it is associated with in the original tree. The query in (3.1) can be represented as in Figure 3.1a, while Figure 3.1b shows an example of subquery.

Definition 3.2 (Subquery) Given queries S and Q, we say that S is a *subquery* of Q, and write $S \subseteq Q$, if all of the following conditions hold:

$$V(S) \subseteq V(Q) \ ; \tag{3.2a}$$

$$E(S) \subseteq E(Q) \ ; \tag{3.2b}$$

$$\forall n \in V(S), \quad \mathcal{V}_S(n) \subseteq \mathcal{V}_Q(n) \ ; \tag{3.2c}$$

$$\forall e \in E(S), \quad \mathcal{E}_S(e) = \mathcal{E}_Q(e) \ . \tag{3.2d}$$

We say that S is a *complete subquery* of Q (in symbols $S \subseteqq Q$) if in addition it also holds that:

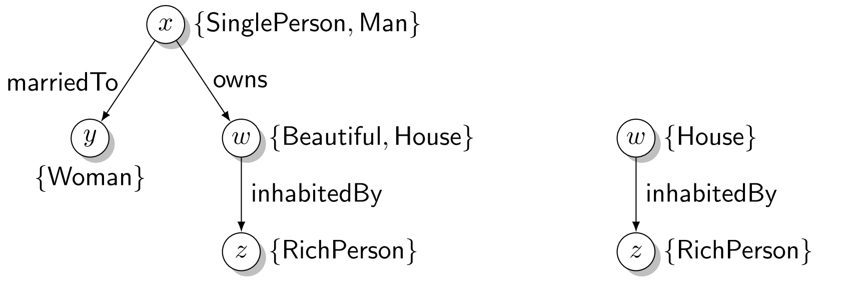
$$\forall n \in V(S), \quad \mathcal{V}_S(n) \supseteq \mathcal{V}_Q(n) \ ; \tag{3.2e}$$

and

$$\forall n \in V(Q), \quad n \in V_{\mathrm{des,Q}}(o_S) \implies n \in V(S) \ . \tag{3.2f}$$

Figure 3.1a shows how the conjunctive query expressed by (3.1) is represented as a directed labelled tree according to Definition 3.1. However, let us stress once more the fact that such a query, being unsatisfiable, cannot be formulated using the operations in the Query Tool's functional API, which will be introduced in the next section. An example of subquery for the above query is shown in Figure 3.1b.

A selection within a query Q is a subquery of Q, which is called *simple* if it is complete or consists of a single node labelled by either one, in which case we speak of an *atomic selection*, or all the labels associated with that node in Q.



(a) A query Q (b) A subquery S of Q

Figure 3.1: Examples of query and subquery

Definition 3.3 (Selection) A *selection* within a query Q is a subquery S of Q. A selection S within a query Q is *simple* if one of the following holds:

$$E(S) = \varnothing \text{ and } |\mathcal{V}_S(o_S)| = 1 < |\mathcal{V}_Q(o_S)| \ ; \tag{3.3a}$$

$$E(S) = \varnothing \text{ and } \mathcal{V}_S(o_S) = \mathcal{V}_Q(o_S) \ ; \tag{3.3b}$$

$$S \subseteqq Q \ . \tag{3.3c}$$

A (simple) selection satisfying (3.3a) is called *atomic*.

Every selection S within a query Q partitions the nodes of Q into *selected*, which belong to $V(S)$, and *unselected*, belonging to $V(Q) \setminus V(S)$. The selected nodes can be further partitioned into *totally selected*, having all of their labels selected, and *partially selected*, which have some, but not all, of their labels selected. In symbols, the former are the elements of

$$V_{\text{ts}} = \big\{ n \in V(S) \mid \big|\mathcal{V}_S(n)\big| = \big|\mathcal{V}_Q(n)\big| \big\} \ , \tag{3.4}$$

while the latter constitute the set

$$V_{\text{ps}} = \big\{ n \in V(S) \mid \big|\mathcal{V}_S(n)\big| < \big|\mathcal{V}_Q(n)\big| \big\} \ . \tag{3.5}$$

From the graphical point of view, we will use special notation for specifying a selection, consisting in underlining the selected concept names directly within the query. As an example, the selection of Figure 3.1b is represented within the query of Figure 3.1a as shown in Figure 3.2. For the sake of clarity, the selected nodes are drawn using a double circle, although this information is redundant and not strictly necessary, because selected nodes are already identified as those having some of their associated concept names underlined .

Figure 3.3 shows the different types of simple selection within the sample query of Figure 3.1a: atomic when the selection is an atomic query, single-node when it consists of a single node along with all of the concept names associated with that node in the original query, and complete when it consists of a node and all of its descendants along with all of the concept names associated with each of those nodes in the original query. Any combination of the three types of simple selection is possible and, in particular, the selection consisting of a leaf node associated with a single concept name in the original query is atomic, single-node and complete at the same time. Note that, in general, a single-node

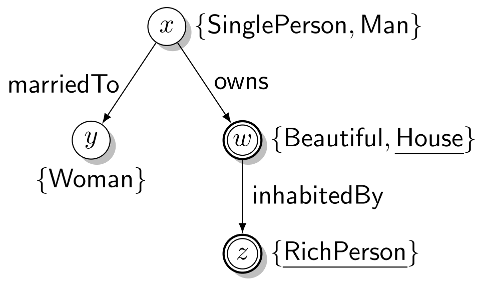
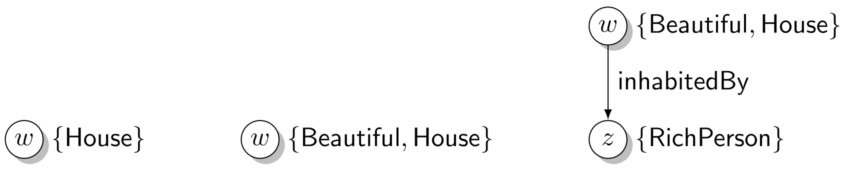


Figure 3.2: Graphical notation for selections

selection is also complete if the node is a leaf[1] and it is atomic if the node is associated[1] with one concept name only.



(a) Atomic selection (b) Single-node selection (c) Complete selection

Figure 3.3: Different types of simple selection

So far the word "label" has been generically used for indicating concept and role names associated with nodes and edges in a query. However, as a concept (role) name can be associated with more than one node (edge) in the same query, we need to be more precise and always refer to a label together with the node or edge it is associated with. This leads us to the following definition.

Definition 3.4 (Label) Given a query Q, a *label* is a pair $\langle x, y\rangle$ in which either $x \in V(Q)$ and $y \in \mathcal{V}(x)$, or $x \in E(Q)$ and $y = \mathcal{E}(x)$. In the former case we speak of a *node label*, in the latter of an *edge label*. The set of node and edge labels occurring in Q is defined as follows:

$$\begin{aligned}\mathsf{labels}(Q) := &\{\langle n, c\rangle \mid n \in V(Q),\ c \in \mathcal{V}(n)\} \cup \\ &\{\langle e, r\rangle \mid e \in E(Q),\ r = \mathcal{E}(e)\} \quad .\end{aligned} \tag{3.6}$$

Moreover, the set of labels of a node n is given by

$$\mathsf{labels}(n) := \{\langle n, c\rangle \mid c \in \mathcal{V}(n)\} \quad . \tag{3.7}$$

Occasionally, we might still refer to a concept (role) name associated with a node (edge) simply as a "label", but only when there is no risk of ambiguity.

The result of deleting the selected node labels from a query is clearly something "weaker" than the original, but it might not be a query. In particular, special care is needed for totally selected nodes, because if we simply deleted all of their labels they would end up having an empty set of labels, which is

[1]In the original query.

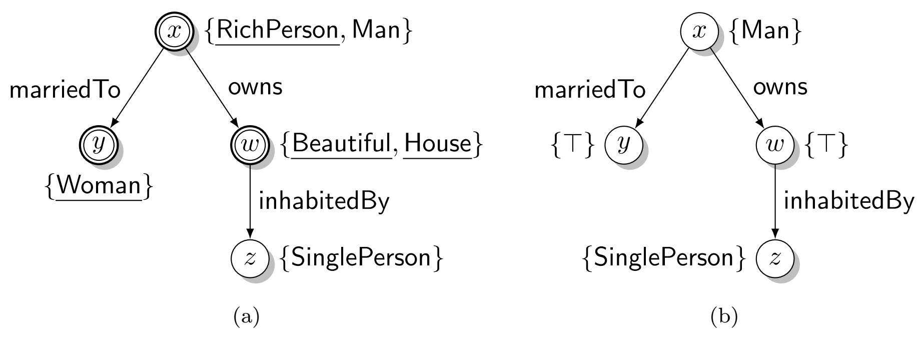


Figure 3.4: (a) A selection within a query, and (b) the result of weakening.

not allowed according to the definition of query. In order to ensure that the result will be a query, each totally selected node is then associated with $\top$ as its only label. The deletion of the selected node labels from a query, opportunely replacing the set of labels of totally selected nodes with the singleton $\{\top\}$, is called "weakening" w.r.t. a selection. A graphical example is shown in Figure 3.4, while the formal definition is given below.

Definition 3.5 (Weakening) Given a selection S within a query Q, the *weakening* of Q w.r.t. S is the query $Q \ominus S$ obtained from Q by replacing its node-labelling function by:

$$\mathcal{V} := \Big(\mathrm{id} \circ \big(\mathcal{V}_Q \mathbin{\widehat{\setminus}} \mathcal{V}_S\big)\Big)\,[\varnothing \;/\; \{\top\}] \quad . \tag{3.8}$$

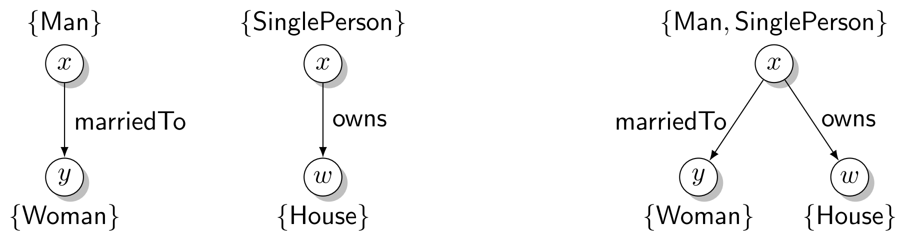
Nodes having only $\top$ as their label are called $\top$-*nodes*.

Another useful operation is that of putting together two queries to create a new one. In general, the naive union of two queries is not a query, as their node sets may be disjoint or the interaction between their edges may give rise to undesired cycles. Therefore, we restrict the way of combining two queries to the simpler case of "appending" one query to a node of the other, in which the only requirement is that the two queries in question must have only one node in common, namely the root of the appended query. By Proposition 2.2, this condition is sufficient to ensure that the result of their union is a query. For the sake of continuing our botanic metaphor about trees, we will call the described operation "grafting" rather than "appending". An example is shown in Figure 3.5.

Definition 3.6 (Grafting) Given queries Q and R s.t. $V(Q) \cap V(R) = \{o_R\}$, where o_R is the root of R, the *graft* of R onto Q is the query $Q \Cup R$ defined as follows:

$$\langle V(Q) \cup V(R),\ E(Q) \cup E(R),\ o_Q,\ \mathcal{V}_Q \mathbin{\widehat{\cup}} \mathcal{V}_R,\ \mathcal{E}_Q \mathbin{\dot{\cup}} \mathcal{E}_R \rangle \quad . \tag{3.9}$$

A query can be represented in *linear form* as a sequence of labels, which is meant to provide the basis for a representation in natural language. In this respect, however, there are some constraints that a linearised query has to satisfy, thus only certain sequences of labels are admissible as a linear form of a query. In particular, in a linearised query, every edge label (that is, a property)



(a) A query Q (b) A query R (c) The result of $Q \uplus R$

Figure 3.5: Example of grafting

must always be preceded by all the labels of its start node and "immediately" followed by at least one of the labels of its end node. Roughly, this means that a property cannot be mentioned before mentioning its subject and after mentioning its object. There is an additional requirement that the linear form of a query must satisfy, stating that all the labels describing a node cannot be mixed up with labels not referring to the same node. We may call this last constraint the "no change of subject" condition. We will now formally define the notion of "linearisation" of a query and then further clarify the meaning of each constraint we impose.

Definition 3.7 (Linearisation) A *linearisation* of a query Q is a strict total order $\lhd$ on $\mathsf{labels}(Q)$ s.t., for each edge $e \in E(Q)$, both of the following conditions hold:

$$\forall l \in \mathsf{labels}(\mathsf{init}(e)), \qquad l \lhd \langle e, \mathcal{E}(e)\rangle \qquad ; \tag{3.10a}$$
$$\exists l \in \mathsf{labels}(\mathsf{ter}(e)), \qquad \langle e, \mathcal{E}(e)\rangle \blacktriangleleft l \qquad ; \tag{3.10b}$$

and, for each node $n \in V(Q)$, it is the case that:

$$\forall l_1, l_2 \in \mathsf{labels}(n), \qquad l_1 \lhd l \lhd l_2 \implies l \in \mathsf{labels}(n) \ . \tag{3.10c}$$

Moreover, we say that $l_1 \in \mathsf{labels}(Q)$ *immediately precedes* $l_2 \in \mathsf{labels}(Q)$, and write $l_1 \blacktriangleleft l_2$, if $l_1 \lhd l_2$ and there is no $l \in \mathsf{labels}(Q)$ such that $l_1 \lhd l \lhd l_2$. In this case we also say that l_2 *immediately follows* l_1. Note that $\lhd = \blacktriangleleft^+$, where the symbol "+" denotes transitive closure.

The first two conditions in Definition 3.7, that a strict total order must satisfy to fully qualify as a linearisation, informally say that "the label of every edge is preceded by all the labels of its start node and followed by at least one of the labels of its end node". The third condition states that "between any two labels of a node there can only be (distinct) labels of the same node". Note that in general, without further restrictions, a query admits more than one linearisation. For example, two of the several possible linearisations of the query in Figure 3.1a are given by:

$$\begin{aligned}&\langle x, \mathsf{SinglePerson}\rangle \lhd_1 \langle x, \mathsf{Man}\rangle \lhd_1 \langle\langle x, y\rangle, \mathsf{marriedTo}\rangle \lhd_1 \langle y, \mathsf{Woman}\rangle \\ &\lhd_1 \langle\langle x, w\rangle, \mathsf{owns}\rangle \lhd_1 \langle x, \mathsf{Beautiful}\rangle \lhd_1 \langle x, \mathsf{House}\rangle \\ &\qquad\qquad\qquad \lhd_1 \langle\langle w, z\rangle, \mathsf{inhabitedBy}\rangle \lhd_1 \langle z, \mathsf{RichPerson}\rangle\end{aligned}$$

and

$$\begin{aligned}
&\langle x, \mathsf{SinglePerson}\rangle \lhd_2 \langle x, \mathsf{Man}\rangle \lhd_2 \langle\langle x, y\rangle, \mathsf{marriedTo}\rangle \lhd_2 \langle y, \mathsf{Woman}\rangle \\
&\quad \lhd_2 \langle\langle w, z\rangle, \mathsf{inhabitedBy}\rangle \lhd_2 \langle z, \mathsf{RichPerson}\rangle \\
&\qquad\qquad\qquad \lhd_2 \langle\langle x, w\rangle, \mathsf{owns}\rangle \lhd_2 \langle x, \mathsf{Beautiful}\rangle \lhd_2 \langle x, \mathsf{House}\rangle \;.
\end{aligned}$$

A query can be modified in a number of (predefined) ways, some of which involve the presence of a selection. With respect to query manipulation, another important notion is that of *sticky edges*, which are edges that can only be deleted explicitly (that is, when performing a delete operation), but never implicitly (e.g., as the consequence of a substitute operation). The meaning and importance of sticky edges will become more clear in the next section, where we introduce and describe the two operations delete and substitute. For the moment, sticky edges can be simply understood as immutable (to some extent) pieces of information within a query, which are not modified as a "side effect" of an operation not directly intended to do so.

In order to keep track of which node and edge labels are selected within a query and which should be prevented from being modified, we give the following definition:

Definition 3.8 (State) A *state* is a triple $\langle Q, S, \widetilde{E}\rangle$ in which Q is a query, S is either ϵ or a selection within Q, and $\widetilde{E} \subseteq E(Q)$. The symbol "$\epsilon$" denotes *absence* of selection.

Absence of selection has not be interpreted as "emptiness". In fact, since a selection is a (sub)query, by definition it always contains at least one node, namely the root, therefore the notion of empty selection does not exist in our framework.

3.2 Functional API

In this section, we will present the Query Tool's functional API, describing the available operations for the formulation of a query and its refinement. These operations are backed up by reasoning services running over the ontology, that are responsible of ruling out all the redundant and contradictory information with respect to the current query, in order to provide the user with relevant and contextual choices only.

To draw the inferences that are at the basis of the query formulation tasks, we express a query into a concept of some DL language $\mathcal{L}$, for which a reasoner is available. In the following, we assume the existence of an underlying knowledge base $\mathcal{K}$ in such a DL language $\mathcal{L}$ over C and R, and we define a function roll-up that, given a query as input, returns its translation into a concept in $\mathcal{L}$.

Definition 3.9 (roll-up) Given a query Q and a node $n \in V(Q)$, the operation roll-up(Q, n) encodes Q into a DL concept in $\mathcal{L}$ w.r.t. n. The operation roll-up(Q, n) is defined as encode(Q, n, n), where encode is the recursive procedure described in Algorithm 3.1. We use roll-up(Q) as an abbreviation for roll-up(Q, o), where o is the root of Q.

The roll-up operation performs a complete visit of the underlying tree of a query Q, starting from a node n and using the encode procedure as follows:

Algorithm 3.1 Calculate $\mathsf{encode}(Q, n, m)$

Input: a query Q and two nodes $n, m \in V(Q)$
Output: a concept C expressing Q in the DL language $\mathcal{L}$

```
1:  $C \leftarrow c$, for some $c \in \mathcal{V}(n)$
2:  for all $x \in \mathcal{V}(n)$ such that $x \neq c$ do
3:      $C \leftarrow C \sqcap x$
4:  end for
5:  for all $x \in V^1_{\text{des,Q}}(n)$ such that $x \neq m$ do
6:      $R \leftarrow \mathcal{E}(\langle n, x \rangle)$
7:      $C \leftarrow C \sqcap \exists R\,.\,\mathsf{encode}(Q, x, n)$
8:  end for
9:  if $n \neq o$ then
10:     Let $p \in V^1_{\text{anc,Q}}(n)$
11:     if $p \neq m$ then
12:         $R \leftarrow \mathcal{E}(\langle p, n \rangle)$
13:         $C \leftarrow C \sqcap \exists R^-\,.\,\mathsf{encode}(Q, p, n)$
14:     end if
15: end if
16: return $C$
```

1. process each node in the complete subquery of Q rooted in n;
2. process the subqueries rooted in each of the nodes in the inverse path from n to the root of Q, skipping those which have already been processed.

We will now comment and explain in detail the pseudocode of Algorithm 3.1. The first line of the procedure simply initialises the return value C to a concept name c chosen among those associated with node n. In lines 2–4, all of the other concept names in $\mathcal{V}(n)$ are then intersected with C, resulting in the following conjunction:

$$C = \bigsqcap_{c \in \mathcal{V}(n)} c \ . \tag{3.11}$$

For each child x of n, the second loop in lines 5–8 adds to the concept expression in (3.11) an additional conjunct of the form $\exists R.D$, where R is the role name associated with the edge going from n to x and D is the recursive encoding of the subquery of Q rooted in x. The reason why only the children of n which are distinct from m are taken into consideration at this point will become clear soon. In lines 9–15, if n is non-root and its parent p is distinct from m, we add to the concept expression C so far obtained a new conjunct of the form $\exists R^-.D$, where R is the role name associated with the edge from p to n and D is the recursive encoding of the subquery of Q rooted in p. The third argument to encode represents the previously processed node and it used to control the encoding in two ways:

- when the active call originated from within a call having as focus one of the children of the current focus, it avoids a recursive call with that child as focus (see lines 5–8);

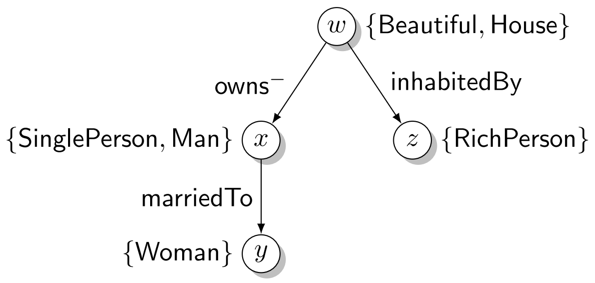


Figure 3.6: The context of the query in Figure 3.1a with respect to node w, represented as a tree. The edges in the path from the (original) root x to the focus w are inverted in the context using the operator "$^-$".

- when the active call was initiated from within a call having as focus the parent node of the current focus, it inhibits the execution of lines 12–13.

The DL concept resulting from $\textsf{roll-up}(Q, n)$ is called the *context* of Q with respect to n. The context of a query Q contains essentially the same information as Q, but relative to a specific node n, which we call the *focus*. In other words, the informative content of the query is expressed from the point of view of the focus node. Although not formally correct, we can visualise the context of a query Q w.r.t. a node n as a query C rooted in n, having the same nodes as Q along with their labels, and such that all the edges in the path from the root of Q to n in Q are "inverted" in C, while all other edges in Q are left untouched in C. For example, the context of the query in Figure 3.1a w.r.t. node w is given by DL concept

$$\begin{aligned}\textsf{Beautiful} \sqcap \textsf{House} \sqcap \exists\,\textsf{owns}^- . \big(\textsf{SinglePerson} \sqcap \textsf{Man} \sqcap \\ \exists\,\textsf{marriedTo} . \textsf{Woman}\big) \sqcap \exists\,\textsf{inhabitedBy} . \textsf{RichPerson}\end{aligned}$$

and it can be represented in the shape of a tree as shown in Figure 3.6.

Once we have defined the roll-up of a query, we can say that two queries are equivalent if their corresponding encoding (i.e., the concepts into which they are respectively encoded) are such.

Definition 3.10 (Query equivalence) Two queries Q_1 and Q_2 are *equivalent*, in symbols $Q_1 \equiv Q_2$, if $\textsf{roll-up}(Q_1) \equiv_{\mathcal{K}} \textsf{roll-up}(Q_2)$.

Note that, from a formal point of view, the tree in Figure 3.6 is not a query, as associating a role of the form R^- with an edge is not allowed. However, if it were such, then its roll-up (w.r.t. its root) would be equivalent to $\textsf{roll-up}(Q, w)$, where Q is the query of our running example, shown in Figure 3.1a.

The following definition introduces the important notion of *query satisfiability* with respect to a knowledge base, which is assumed to be consistent.

Definition 3.11 We say that a query Q over a consistent knowledge base $\mathcal{K}$ is *satisfiable* if its roll-up is such in $\mathcal{K}$, that is, if $\mathcal{K} \not\models \textsf{roll-up}(Q) \sqsubseteq \bot$.

The functional API of the Query Tool consists of three main parts: the underlying reasoning services, the operations for query manipulation and, lastly, the constraints that such operations impose on the resulting query in order to opportunely restrict the number of possible linearisations.

3.2.1 Reasoning services

We start the specification of the Query Tool's functional API by defining the reasoning services that are needed and used by some of the other operations in order to modify a query in a meaningful way.

Definition 3.12 (getCompatibles) The operation getCompatibles takes as input a query Q and a focus node $n \in V(Q)$, and returns a DAG $G = (V, E)$, in which $V \subseteq \mathsf{C}$. Let $C = \mathsf{roll\text{-}up}(Q, n)$ be the context of Q w.r.t. n. Then, a concept name $c \in \mathsf{C}$ belongs to V if and only if all of the following conditions are satisfied:

$$\mathcal{K} \not\models c \sqcap C \sqsubseteq \bot \ ; \tag{3.12a}$$

$$\mathcal{K} \not\models c \sqsubseteq C \ ; \tag{3.12b}$$

$$\mathcal{K} \not\models C \sqsubseteq c \ . \tag{3.12c}$$

A pair $\langle c_1, c_2 \rangle \in V \times V$ belongs to E if and only if c_2 is a direct sub-concept of c_1 in $\mathcal{K}$.

Definition 3.13 (getRelations) The operation getRelations takes as input a query Q and a focus node $n \in V(Q)$, and returns a DAG $G = (V, E)$, in which $V \subseteq \mathsf{R} \times \mathsf{C}$. Let $C = \mathsf{roll\text{-}up}(Q, n)$ be the context of Q w.r.t. n. Then, a pair $\langle r, c \rangle \in \mathsf{R} \times \mathsf{C}$ belongs to V if and only if all of the following conditions are satisfied:

$$\mathcal{K} \not\models \exists r^{-}.C \sqsubseteq \bot \ ; \tag{3.13a}$$

$$\mathcal{K} \models c \sqsubseteq \exists r^{-}.C \text{ or } \mathcal{K} \models \exists r^{-}.C \sqsubseteq c \ . \tag{3.13b}$$

A pair $\langle \langle r, c_1 \rangle, \langle r, c_2 \rangle \rangle \in V \times V$ belongs to E if and only if c_2 is a direct super-concept of c_1 in $\mathcal{K}$.

Definition 3.14 (getSupers) The operation getSupers takes as input a query Q and a selection S within Q, and returns a DAG $G = (V, E)$, with $V \subseteq \mathsf{C}$. A concept name $c \in \mathsf{C}$ belongs to V if and only if all of the following conditions are satisfied:

$$\mathcal{K} \not\models c \sqsubseteq \mathsf{roll\text{-}up}(S) \ ; \tag{3.14a}$$

$$\mathcal{K} \models \mathsf{roll\text{-}up}(S) \sqsubseteq c \ ; \tag{3.14b}$$

A pair $\langle c_1, c_2 \rangle \in V \times V$ belongs to E if and only if c_2 is a direct super-concept of c_1 in $\mathcal{K}$.

Definition 3.15 (getEquivalents) The operation getEquivalents takes as input a query Q and a selection S within Q, and returns a DAG $G = (V, \varnothing)$, with $V \subseteq \mathsf{C}$. A concept name $c \in \mathsf{C}$ belongs to V if and only if all of the following conditions are satisfied:

$$\mathcal{K} \models c \sqsubseteq \mathsf{roll\text{-}up}(S) \ ; \tag{3.15a}$$

$$\mathcal{K} \models \mathsf{roll\text{-}up}(S) \sqsubseteq c \ . \tag{3.15b}$$

Definition 3.16 (getSubs) The operation getSubs takes as input a query Q and a selection S within Q, and returns a DAG $G = (V, E)$, with $V \subseteq \mathsf{C}$. A concept name $c \in \mathsf{C}$ belongs to V if and only if all of the following conditions are satisfied:

$$\mathcal{K} \models c \sqsubseteq \mathsf{roll\text{-}up}(S) \ ; \tag{3.16a}$$

$$\mathcal{K} \not\models \mathsf{roll\text{-}up}(S) \sqsubseteq c \ ; \tag{3.16b}$$

$$\mathcal{K} \not\models c \sqcap \mathsf{roll\text{-}up}(Q, o_S) \sqsubseteq \bot \ ; \tag{3.16c}$$

where o_S denotes the root of S. A pair $\langle c_1, c_2\rangle \in V \times V$ belongs to E if and only if c_2 is a direct sub-concept of c_1 in $\mathcal{K}$.

3.2.2 Operations on queries

We will now introduce the operations that are available to the user in order to refine the query. We start with two operations for adding further constraints to the query, namely new concepts and relations. As the query becomes more restrictive, we need to make sure that what the new constraints are compatible with it, in the sense of Definition 3.12 for concepts and of Definition 3.13 for relations.

The operation addCompatible takes as input a focused query and a concept name, obtained by means of getCompatibles (thus compatible with the current query), and adds it to the set of concept names associated with the focus. The operation is defined as the grafting of an atomic query consisting only of the focus associated with the compatible concept name onto the input query. An example of application of the addCompatible operation is shown in Figure 3.7.

Definition 3.17 Let Q be a query and let $n \in V(Q)$ be a focus node. Then, for $c \in V\big(\mathsf{getCompatibles}(Q, n)\big)$ we define:

$$\mathsf{addCompatible}(Q, n, c) := Q \uplus R \ ,$$

where R is the atomic query $\langle n, \varnothing, \{n \mapsto \{c\}\}, \varnothing\rangle$.

The operation addRelation takes as input a query Q, a focus node n and a pair consisting of a role name r and a concept name c, which are obtained by means of getRelations, hence compatible with the current query. The operation is defined as the grafting onto Q of a query R consisting of the focus node n, a new node n' (not in Q) associated with the singleton $\{c\}$ and an edge from n to n' associated with r. An example of application of the addRelation operation is shown in Figure 3.7.

Definition 3.18 Let Q be a query and let $n \in V(Q)$ be a focus node. Then, for $\langle r, c\rangle \in V\big(\mathsf{getRelations}(Q, n)\big)$ we define:

$$\mathsf{addRelation}(Q, n, \langle r, c\rangle) := Q \uplus R \ ,$$

with $R = \langle n, \{\langle n, n'\rangle\}, \{n \mapsto \{\mathcal{V}_Q(n)\}, n' \mapsto \{c\}\}, \{\langle n, n'\rangle \mapsto r\}\rangle$, where $n' \in \mathbf{N}$ and $n' \notin V(Q)$.

We now introduce an operation called prune that, though not directly available to the user, is used by the other operations for query modification, which will be defined later on. Given two queries Q and S, the operation prune deletes from Q the maximal number of non-root nodes, having no incoming sticky edge (if any) and that in S are associated with the same concept names as in Q, such

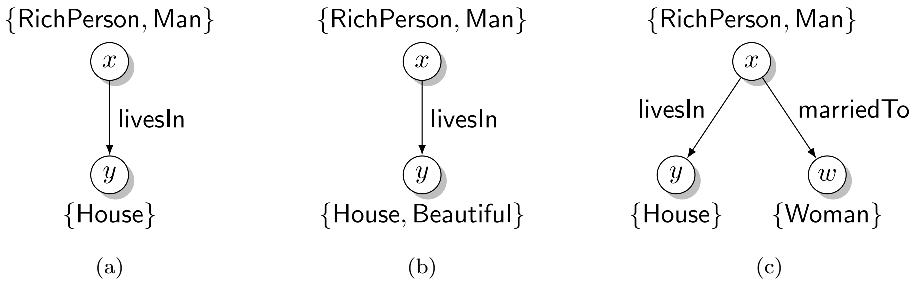


Figure 3.7: (a) A query Q, (b) the output of $\mathsf{addCompatible}(Q, y, \mathsf{Beautiful})$, and (c) the output of $\mathsf{addRelation}(Q, x, \langle\mathsf{marriedTo}, \mathsf{Woman}\rangle)$.

that the result is still a query. An example of pruning is given in Figure 3.8a, where only y is deleted in (a), because the deletion of w would not result in a tree.

Definition 3.19 (prune) Let Q and S be queries and let $\widetilde{E}$ be a set of sticky edges. Let

$$N := \big\{n \in V(Q) \mid V_{\mathrm{des,Q}}(n) = \varnothing,\ n \in V(S),\ \mathcal{V}_S(n) = \mathcal{V}_Q(n),\ \nexists e \in \widetilde{E} \ .\ \mathsf{ter}(e) = n\big\} \quad (3.17)$$

be the set of all nodes that are associated with the same set of concept names in both Q and S, but which are non-root leaves in Q and are not the terminal node of a sticky edge, and let

$$R := \langle\ V(Q) \setminus N, \quad E(Q) - N, \quad \mathcal{V}|_{V(R)}, \quad \mathcal{E}|_{E(R)}\ \rangle \quad (3.18)$$

be the query resulting from Q by deleting all of the nodes in N, along with the corresponding incident edges. Then, the operation $\mathsf{prune}(Q, S, \widetilde{E})$ is recursively defined as follows:

$$\mathsf{prune}(Q, S, \widetilde{E}) = \begin{cases} R & R = Q\ , \\ \mathsf{prune}(R, S, \widetilde{E}) & \text{otherwise}\ . \end{cases} \quad (3.19)$$

Note that in prune, the input argument S is not required to be a selection within Q, but it can be any query. However, in the example of Figure 3.8a, we used a selection for reasons of space.

We can now introduce the rest of the operations for query refinement, starting with two operations, called weaken and delete, for removing existing constraints from the current query, therefore resulting in more general one. The operation weaken is simply defined as the weakening of a query w.r.t. a selection, while the operation delete is a weaken followed by a prune. An example of weaken was already provided in Figure 3.4, while the result of the application of the delete operation to the query of Figure 3.4a is shown in Figure 3.8b.

Definition 3.20 (weaken, delete) Let S be a selection within a query Q and let $\widetilde{E}$ be a set of sticky edges. We define the following operations:

$$\mathsf{weaken}(Q, S) := Q \ominus S\ ;$$
$$\mathsf{delete}(Q, S, \widetilde{E}) := \mathsf{prune}\big(\mathsf{weaken}(Q, S), R, \widetilde{E}\big)\ ;$$

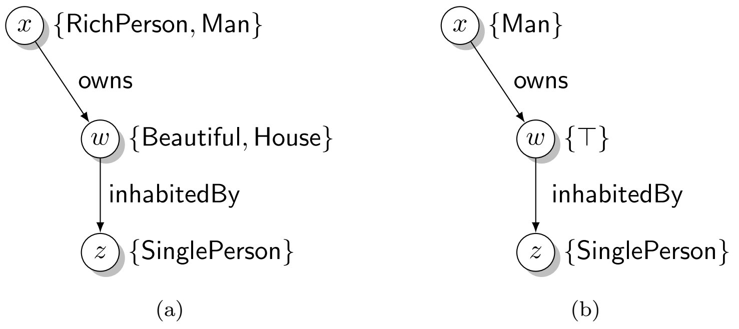


Figure 3.8: The result of the application of the operations (a) prune and (b) delete on the query of Figure 3.4a.

where R is the query obtained from S by replacing its node-labelling function with

$$\left(\left(\mathcal{V}_Q \mathbin{\widehat{\cap}} \mathcal{V}_S\right)[\varnothing \,/\, \{\top\}]\right)_{|V(S)} \; .$$

The last operation we introduce is the "substitution" of a selection with an equivalent, more general or more specific concept. In case of substitution with a more general (resp. more specific) concept, we speak of *generalisation* (resp. *specialisation*). Clearly, the resulting query will be equivalent to or more general/specific than the input query according to whether the substituting concept is equivalent to or more general/specific than the selection. The substitute operation takes as input a query, a selection within it, a set of sticky edges and a concept name chosen among those returned by getSupers, getEquivalents and getSubs. This substituting concept is then added to the set of concept names associated with the root of the selection, similarly to what is done when adding a compatible term, and afterwards the selection is deleted from the query using delete. An example of substitution with a more specific concept (that is, a specialisation) is provided in Figure 3.9. Generalisation and substitution with an equivalent concept or with more complex selections are analogous.

Definition 3.21 (substitute) Let S be a selection within a query Q and let $\widetilde{E}$ be a set of sticky edges. For a concept name $c \in V(\mathsf{getSupers}) \cup V(\mathsf{getEquivalents}) \cup V(\mathsf{getSubs})$ we define the following operation:

$$\mathsf{substitute}(Q, S, \widetilde{E}, c) := \mathsf{delete}(Q \uplus R, S, \widetilde{E})$$

where R is the atomic query $\langle o_S, \varnothing, \{o_S \mapsto \{c\}\}, \varnothing\rangle$.

3.2.3 Linearisation constraints

In this section, we consider queries resulting from the application of the operations introduced previously and define how these are linearised. For this purpose, we extend each operation by imposing additional constraints that every linearisation of the output query must satisfy and such that it is uniquely determined for any given linearisation of the input query. In other words, for

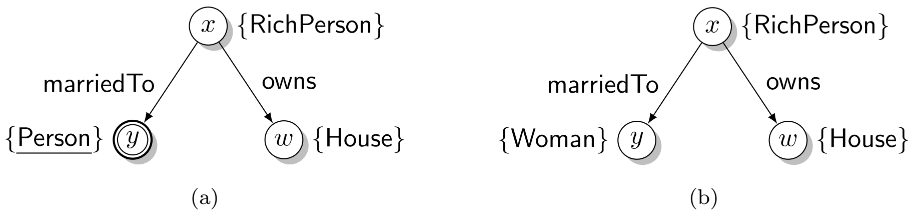


Figure 3.9: (a) A selection S within a query, and (b) the result of substituting S with the more specific concept Woman.

each linearisation $\lhd$ of the input query Q, there exists one and only one linearisation $\lhd'$ of the output query Q', that is, $\lhd'$ is a function of $\lhd$. In particular, given an operation op taking as input a query Q and an ordered sequence I of additional arguments, its counterpart $\mathsf{op}^{\lhd}$ takes as input the same query and extra arguments of op plus a linearisation $\lhd$ of Q. Then, $\mathsf{op}^{\lhd}(Q, I, \lhd)$ returns a pair $\langle Q', \lhd' \rangle$, where $Q' = \mathsf{op}(Q, I)$ and $\lhd' = f(Q', I, \lhd)$ is a linearisation of Q' obtained from the given linearisation of the input query and the additional arguments to op. Function f is called the *linearisation function* of $\mathsf{op}^{\lhd}$ and defines the *linearisation constraints* to be satisfied by $\lhd'$ w.r.t. $\lhd$.

Note that the proposed design is modular in that it allows us to independently modify the linearisation constraints imposed by each operation without affecting the structural changes (i.e., those on the query tree and labelling functions) that the original operation performs on the input query.

We first introduce three basic operations which will be subsequently combined together in the construction of the linearisation function of each core operation introduced in the previous section. We will use the following "building blocks":

DEFINITION 3.22 (Left-Insertion) Let $\langle A, < \rangle$ be a totally ordered set with $<$ strict, $a \in A$ and $b \notin A$. Then, $\mathsf{insL}(b, <, a)$ returns the (only) strict total order $<'$ on $A \cup \{b\}$ such that $< \subseteq <'$ and $b \lessdot' a$.

DEFINITION 3.23 (Right-Insertion) Let $\langle A, < \rangle$ be a totally ordered set with $<$ strict, $a \in A$ and $b \notin A$. Then, $\mathsf{insR}(b, <, a)$ returns the (only) strict total order $<'$ on $A \cup \{b\}$ such that $< \subseteq <'$ and $a \lessdot' b$.

Left-insertion takes care of inserting a new element b into a set A ordered by a strict total order $<$ in such a way that b immediately precedes a given element a of A and the previous order on the elements of A is preserved. Right-insertion performs a similar task, only that in this case b immediately follows a. Thanks to Lemmas 2.1 and 2.2, both operations are well-defined and can be regarded as functions, as the result they produce is uniquely determined by their input arguments.

Informally, the operation $\mathsf{addCompatible}^{\lhd}$ imposes the following linearisation constraint: "the compatible label immediately follows the maximum label (w.r.t. $\lhd$) of the focus node".

$$f_{\mathsf{addCompatible}^{\lhd}}(n, c, \lhd) := \mathsf{insR}\big(\langle n, c \rangle, \lhd, l_{\max}\big) \tag{3.20}$$

where $l_{\max} = \max_{\lhd}\big(\{\langle n, z \rangle \in \mathsf{labels}(Q) \mid z \in \mathcal{V}(n)\}\big)$.

Informally, the operation $\mathsf{addRelation}^{\lhd}$ imposes the following linearisation constraint: "the label of the range node immediately precedes the minimum label (w.r.t. $\lhd$) among those of the edges outgoing from the focus and immediately follows the edge label of the new property". Let $n' = V(Q') \setminus V(Q)$ and $e' = E(Q') \setminus E(Q)$.

$$f_{\mathsf{addRelation}^{\lhd}}(n, \langle r, c\rangle, \lhd) := \mathsf{insL}\big(l_{\text{prop}}, \mathsf{insL}(l_{\text{range}}, \lhd, l_{\text{prop}}), l_{\min}\big) \tag{3.21}$$

where $l_{\min} = \min_{\lhd}\big(\{\langle e, \mathcal{E}(e)\rangle \in \mathsf{labels}(Q) \mid e \in E_{\text{in}}(n)\}\big)$, $l_{\text{prop}} = \langle e', r\rangle$ and $l_{\text{range}} = \langle n', c\rangle$.

3.3 Main Results

The framework and the functional API we devised allow us to formally demonstrate the central property of the Query Tool, namely the fact that it produces only queries which are satisfiable, according to Definition 3.10.

Theorem 3.1 *The query obtained by means of a finite number of applications of the operations addCompatible, addRelation, substitute, weaken and delete to an initial atomic query is satisfiable.*

Proof. By induction on the number of applications of the operations. The base case is that in which the query is atomic and no operation is applied. Then, its roll-up is simply a concept name, therefore trivially satisfiable if the knowledge base over which the query is formulated is consistent.

For the inductive step, let Q be a query obtained by means of n applications of the operations and assume Q to be satisfiable. We need to show that a further application of each and any of the operations to Q results in a query Q' which is still satisfiable. This is trivial in the case of delete and weaken, because both of them always result in a query that is more general than the input one. The application of substitute when the substituting term is an element of getSupers or getEquivalents results in a query that is more general than or equivalent to the input one, therefore satisfiable too. In the case of substitute with a more specific term (that is, an element of getSubs), addCompatible and addRelation, the satisfiability of the resulting query follows from the definition of getSubs, getRelations and getCompatibles, respectively. □

In the remainder of this section we present some interesting properties of linearisations which can be formally proved in our framework, including a few general ones that hold for every linearisation. We start by showing that the conditions given in the definition of linearisation, specifically (3.10b) and (3.10c), imply that "the label of each edge is followed by all the labels of its end node".

Proposition 3.1 *Every linearisation $\lhd$ of a query Q is such that, for each edge $e \in E(Q)$, the following holds:*

$$\forall l \in \mathit{labels}(\mathit{ter}(e)), \qquad \langle e, \mathcal{E}(e)\rangle \lhd l \ . \tag{3.22}$$

Proof. Let $\lhd$ be a linearisation of a query Q, let $n = \mathsf{ter}(e)$ and $l = \langle e, \mathcal{E}(e)\rangle$, for some $e \in E(Q)$. As $\lhd$ is a linearisation of Q, it satisfies all the conditions of Definition 3.7. If $\mathcal{V}(n)$ is a singleton, then (3.22) follows directly by (3.10b). Let us now consider the case $|\mathcal{V}(n)| > 1$. By (3.10b), there is $l_1 \in \mathsf{labels}(n)$ s.t.

$l \blacktriangleleft l_1$. Since $|\mathcal{V}(n)| > 1$, there exists $l_2 \in \mathsf{labels}(n)$ distinct from l_1 and, as $\lhd$ is a linearisation of Q, either $l_1 \lhd l_2$ or $l_2 \lhd l_1$.

Suppose $l_2 \lhd l_1$. Then, since l immediately precedes l_1, we have $l_2 \lhd l \lhd l_1$ and by (3.10c) we obtain $l \in \mathsf{labels}(n)$, which is a contradiction of the fact that l is an edge label. Therefore, it must hold that $l_1 \lhd l_2$ and, by the transitivity of $\lhd$, also $l \lhd l_2$ holds. The case in which $l_1 \lhd l_2$ is analogous and, since l_2 was chosen arbitrarily among the labels of n, this concludes our proof. □

We say that a linearisation $\lhd$ of a query Q is *compatible* with a strict partial order $\prec$ on $V(Q)$ if, for all $n, m \in V(Q)$

$$n \prec m \implies \forall l_1 \in \mathsf{labels}(n), \forall l_2 \in \mathsf{labels}(m) \;.\; l_1 \lhd l_2 \tag{3.23}$$

Then, the following result is a direct consequence of Definition 3.7 and Proposition 3.1.

LEMMA 3.1 *Every linearisation of a query Q is compatible with the tree-order of (the underlying tree of) Q.*

Proof. Let $\lhd$ be a linearisation of a query Q and let $\prec$ be the tree-order of $T = \big(V(Q), E(Q)\big)$. Let $n_1, n_2 \in V(Q)$ be such that $n_1 \prec n_2$, hence there is a path from n_1 to n_2 in T. By (3.10a) and (3.22), with respect to $\lhd$, each edge in the path is preceded by all the labels of its start node and followed by all the labels of its end node. Then, our claim follows by the transitivity of $\lhd$. □

The next property we show is that a linearisation always "starts" from the root of the query.

PROPOSITION 3.2 *Every linearisation of a query is such that the minimum label is a label of the root.*

Proof. Let $\lhd$ be a linearisation of a query Q, and let $l_{\min} = \min_{\lhd}\big(\mathsf{labels}(Q)\big)$. We want to show that $l_{\min}$ is a label of the root of the query, that is, $l_{\min} \in \mathsf{labels}(o)$. By (3.10a), an edge label cannot be minimal w.r.t. $\lhd$ because it is always preceded by all the labels of its initial node. Thus, $l_{\min}$ is a node label, that is, $l_{\min} \in \mathsf{labels}(n)$ for some $n \in V(Q)$.

Now, suppose $n \neq o$. Since each node other than the root has a (unique) parent, there exists an edge $e = \langle z, n \rangle \in E(Q)$, for some node $z \in V(Q)$, with $z \neq n$. By (3.22), $\lhd$ is such that each edge label must precede all the labels of its end node, but this is not possible by the minimality of $l_{\min}$. Therefore, we conclude that $n = o$. □

The last general property of linearisations that we prove is that by restricting the linearisation of a query on any of its subqueries, we obtain a linearisation of that subquery.

LEMMA 3.2 *Let $\lhd$ be a linearisation of a query Q. Then, for each subquery S of Q, the restriction of $\lhd$ on $\mathsf{labels}(S)$ is a linearisation of S.*

Proof. Let $S \subseteq Q$ and let $\lhd_S$ denote the restriction of $\lhd$ on $\mathsf{labels}(S)$. Clearly, $\lhd_S$ is a strict total order on $\mathsf{labels}(S)$, but we need to show that in addition it satisfies the conditions of Definition 3.7.

As $S \subseteq Q$, each edge of S is also an edge in Q and each node of S is also a node in Q. Moreover, the initial and terminal nodes of an edge in S are nodes in S, hence also in Q. For each edge e in S we have that $\mathcal{E}_S(e) = \mathcal{E}_Q(e)$ and we write the edge label associated with e as $\langle e, \mathcal{E}(e)\rangle$ (which is the same both in S and Q). However, for each node n in S we have that $\mathsf{labels}_S(n) \subseteq \mathsf{labels}_Q(n)$.

Suppose there is a label $l \in \mathsf{labels}(S)$ for which $\lhd_S$ does not satisfy (3.10a) or (3.10c). Then, in both cases l causes the same condition not to be satisfied also by $\lhd$, which is a contradiction.

Now, suppose that $\lhd_S$ does not satisfy (3.10b) w.r.t. some $e \in E(S)$. Let $n = \mathsf{ter}(e)$ and $l_e = \langle e, \mathcal{E}(e)\rangle$. Then, there is no label $l \in \mathsf{labels}_S(n)$ such that $l_e \blacktriangleleft_S l$. Since $\lhd_S$ is a strict total order, there is indeed some $l_1 \in \mathsf{labels}(S)$ such that $l_e \blacktriangleleft_S l_1$, but then $l_1 \notin \mathsf{labels}_S(n)$, hence also $l_1 \notin \mathsf{labels}_Q(n)$. Assume w.l.o.g. that $\mathsf{labels}_S(n)$ is a singleton whose only element is l_2. Then, we have either $l_2 \lhd_S l_e$ or $l_e \lhd_S l_2$. Since $\lhd$ is a linearisation of Q, it satisfies (3.10b), thus there is $l' \in \mathsf{labels}_Q(n)$ such that $l_e \blacktriangleleft l'$.

In the case in which $l_2 \lhd_S l_e$, we have $l_2 \lhd l_e \lhd l'$ and since both l_2 and l' belong to $\mathsf{labels}(Q)_Q(n)$ while l_e does not, this is a contradiction of (3.10b). In the case in which $l_e \lhd_S l_2$, we have $l_e \blacktriangleleft_S l_1 \lhd_2 l_2$, hence $l_e \blacktriangleleft l' \lhd l_1 \lhd l_2$. Since l' and l_2 are both in $\mathsf{labels}(Q)_Q(n)$ while l_1 is not, this is again a contradiction of (3.10b). □

Theorem 3.2 *The query obtained through a finite number of ordered applications of the operations addCompatible, addRelation, substitute, weaken, delete to an initial atomic query admits one and only one linearisation satisfying all of the constraints that each operation imposes.*

CHAPTER 4

Implementation

Following the formal specification presented in the previous chapter, the Query Tool has been redesigned and reimplemented, undergoing deep and radical changes at its core. The result is a better organised and more flexible system, which we will here illustrate in detail. The most important and significant difference from the old Query Tool consists in the use of the OWL-API in place of the previous DIG-based reasoning engine. This allowed for the adoption of the OWL-DL standard and paid back in terms of execution speed and ease of use.

The project is currently in alpha stage and the latest development version of the source code can be checked out from the Git repository `http://russel.inf.unibz.it/~pguagliardo/querytool.git`.[1] The Query Tool provides the following features:

- Choice of the schema file to load (at present only *.owl files are supported);
- Choice of which reasoner to use (either Pellet or FaCT++);
- Creation of a new query by selecting a starting term among the available classes present in the schema;
- Selection of a focus node for adding a compatible term chosen from a list of available ones (depending on the current query and focus);
- Selection of a focus node for adding a new relation chosen from a list of available ones (depending on the current query and focus);
- Pretty-printing of the query as a tree with indentation of children.

4.1 Structure and Design

The core of the new Query Tool software is a Java^{TM} library consisting of the following main packages:

[1]The link is accessible only from within the Scientific Network of South Tyrol.

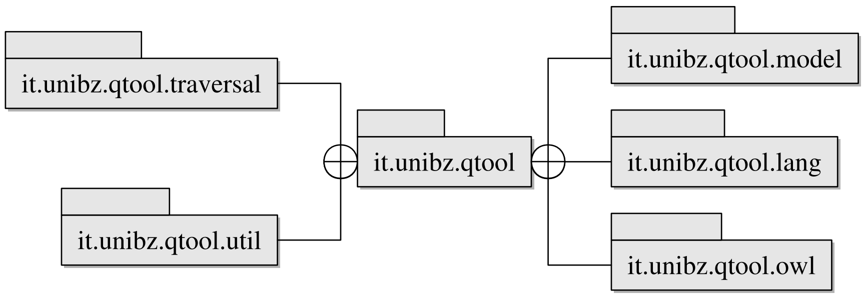


Figure 4.1: Package diagram of the Query Tool

- `it.unibz.qtool.model` contains the Java interfaces representing entities and relationships between them in a conceptual schema;
- `it.unibz.qtool.lang` provides language constructs for combining schema entities and relationships into complex expressions;
- `it.unibz.qtool.owl` contains the Java classes used for importing a conceptual schema from an OWL ontology and reasoning over it;
- `it.unibz.qtool` contains the Java classes representing queries and selections, and provides the interface to the reasoning services used for modifying a query in a consistent way.

The complete package diagram of the Query Tool is shown in Figure 4.1. As we shall see later on, the packages `it.unibz.qtool.traversal` and `it.unibz .qtool.util` include few additional interfaces and classes which are used for performing general operations on queries. For the moment, we concentrate our attention on the base package `it.unibz.qtool` and its `model`, `lang` and `owl` subpackages.

As shown in Figure A.1, the `model` package is organised as follows: the Java interface `Schema` represents a conceptual schema consisting of `SchemaElement`'s, each of which can be a `SchemaClass` or a `SchemaProperty`. Moreover, the latter may be a `SchemaRelation` or a `SchemaAttribute`, although attributes and datatypes are not currently supported (see Chapter 5). As we only deal with roles, a `SchemaRelation` is intended to represent a binary relation between two `SchemaClass`'s. The fundamental interface in the package is `SchemaAdapter`, which is responsible of importing a specific `Schema` from an external source such as a file. Note that this interface is parametrised with the kind of schema it is able to process.

Figure A.2 shows the structure of the `lang` package, containing the language constructs for building complex expressions from schema classes and relations. Two kinds of expressions can be constructed: `ConceptExpression` and `RoleExpression`, whose basic building blocks are respectively `SchemaClass` and `SchemaRelation`, in the sense that a schema class (relation) is a concept (role) expression itself. The only other `RoleExpression` is `InverseRole`, representing the inverse of a role expression, while a `ConceptExpression` can be built using the following constructs:

- `ComplementOf` represents the negation of a concept expression;

DL	**Description**	**Java interface**
$R \in \mathsf{R}$	role name	`SchemaRelation`
$R \in \mathsf{R} \cup \{R^- \mid R \in \mathsf{R}\}$	general role	`RoleExpression`
R^-	inverse role (with R general role)	`InverseRole`
$A \in \mathsf{C}$	atomic concept (concept name)	`SchemaClass`
C	general concept (can be atomic)	`ConceptExpression`
$\neg C$	negation	`ComplementOf`
$C \sqcap D$	conjunction	`IntersectionOf`
$\exists R.C$	existential quantification	`SomeRestriction`

Table 4.1: Correspondence between DL constructs and the Java interfaces representing them in the Query Tool.

- `IntersectionOf` represents the conjunction of two or more concept expressions;
- `SomeRestriction` operates on a role expression and a concept expression, and represents the complex concept $\exists R.C$ of DL, where R is a role and C a concept.

The correspondence between role and concept constructs in DL and the Java interfaces used for representing them in the Query Tool can be found in Table 4.1.

The class diagram of the base package `it.unibz.qtool` is shown in Figure A.4. The Java class `Node` represents a node in a labelled tree, thus having a single parent `Node`, which may be `null` if the node in question is root, and a list of child `Node`'s. Moreover, it is associated with a list of `SchemaClass`'s and the incoming edge from its parent (if any) is labelled by a `SchemaProperty`. The main class of the package is `Query`, which represents a tree as an aggregation of `Node`'s and keeps track of its root. The `Query` class contains methods implementing all of the operations defined in the Query Tool functional API (see Chapter 3, Section 3.2.2), which are used for modifying the query tree itself. In order to do this in a consistent way, so that the obtained query is always satisfiable, `Query` has methods for getting the compatible terms and relations, which depend on its current status, by triggering the necessary reasoning services discussed in Section 3.2.1 of Chapter 3. These are in turn provided by a `QueryReasoner` associated with the `Schema` over which the query is created. Before any reasoning task, a `Query` must be encoded into a `ConceptExpression`: this is accomplished by means of the method `getContextWRT(Node)`, that performs the roll-up of the query by implementing Algorithm 3.1. Note that, given a query `q`, a call to `q.encode()` yields the same result of `q.getContextWRT(q.getRoot())`, coherently with Definition 3.9. In fact, the `encode()` method is provided only for convenience. Some operations, like `substitute`, require a selection within the query: a `Selection` associates one or more nodes with a subset of their labels, thus defining a sort of "filter" on the query. When a selection is present, the `encode(Selection)` method of `Query` encodes the query taking into account only the nodes and their associated labels in the selection. For this reason, before encoding and/or using a `Selection`, we need to make sure that it is connected by calling its `isConnected()` method.

As shown in figure A.3, the only Java classes in the `owl` package are `OWLSchema`, `OWLSchemaAdapter` and `OWLQueryReasoner`, implementing the (external) interfaces `Schema`, `SchemaAdapter` and `QueryReasoner`, respectively. As mentioned earlier, the `OWLSchemaAdapter` opportunely binds the parameter `T`, which must be of type `Schema` (see Fig. A.1), of the `SchemaAdapter` interface to `OWLSchema`, whose creation it is responsible of. An `OWLSchema` is then associated with an `OWLQueryReasoner`, which interacts with the underlying reasoner by means of the OWL-API.

4.2 Code Examples

In this section we present some concrete examples of how the Query Tool API is used in practise from the developer's point of view.

Importing a schema

In order to get a `Schema` populated with data coming from an external source (currently only files are supported), one first needs to create an instance of a `SchemaAdapter` for the particular kind of schema to be imported, and then call the appropriate method of the adapter which actually loads the data from the desired source. For instance, to load an OWL ontology from a file, we create a new `OWLSchemaAdapter` specifying which reasoner to associate with the schema (either `ReasonerType.PELLET` or `ReasonerType.FACTPLUSPLUS`) and then call the method `importSchema(File f)`, as shown in Listing 4.1. Note that `OWLSchemaAdapter` is a `SchemaAdapter<OWLSchema>`, that is, an adapter for dealing with OWL schemas.

```java
Schema sch = null;
SchemaAdapter<OWLSchema> adp = new OWLSchemaAdapter(ReasonerType.
    PELLET);
try {
    sch = adp.importSchema(new File("path/to/ontology.owl"));
} catch (Exception e) {
    e.printStackTrace();
}
```

Listing 4.1: Code for importing an OWL ontology from a file.

Creating a query

A query is created over a previously imported schema and initialised with a `SchemaClass` chosen among all classes in the schema, which can be retrieved by calling the method `getSchemaClasses()` of `Schema`. The general steps to follow in order to create a new (atomic) query are listed below:

1. import a `Schema` by using the appropriate `SchemaAdapter` and associate it with a suitable `QueryReasoner`;
2. create an instance of `Query` over the imported conceptual schema;

3. initialise the query with a starting term chosen among the `SchemaClass`'s of the schema associated with the query.

As example, suppose we performed the first step by importing an OWL ontology as in Listing 4.1; then, to create and initialise a new query over the imported schema, we proceed as in Listing 4.2.

```
Collection<SchemaClass> classes = sch.getSchemaClasses(); // get
    all classes in the schema
SchemaClass term = /* ...
    code for capturing the user choice of an element from "classes
        "
... */
Query q = new Query(sch); // create a new query over the schema
q.init(term); // initialise the query with chosen starting term
```

Listing 4.2: Code for creating and initialising a new query over a previously imported schema `sch`.

In future versions of the Query Tool, we plan to directly initialise a query with $\top$ at the time of its creation, so that explicit initialisation will not be required anymore. The user can then substitute the starting term with a more specific one. Note that the representation of $\top$ is schema-dependent[2] and it can be obtained from a specific `Schema` by calling its `getTopConcept()` method (see Figure A.3).

Adding a compatible term

To perform this operation we need a focus `Node` and a `SchemaClass` compatible with the current query. The former is chosen among all nodes in the query, which can be obtained with a call to `getNodes()`. The latter must be chosen from the available compatible terms (if any), which depend on the current query and focus and are obtained by calling the `getCompatibles(Node n)` method of the query, where `n` is the focus node. Finally, a call to the method `addCompatible(Node n, SchemaClass c)` adds the chosen compatible `c` to the labels of the focus node `n`. Listing 4.3 shows an example of how the described process is realised in practise.

```
Collection<Node> nodes = q.getNodes(); // get all nodes in the
    query
Node focus = /* ...
    code for capturing the user choice of an element from "nodes"
... */
Collection<SchemaClass> compatibles = q.getCompatibles(focus); //
    get available compatible terms wrt focus
if (compatibles.isEmpty() == false) {
    SchemaClass comp = /* ...
        code for capturing the user choice of an element from "
            compatibles"
    ... */
    q.addCompatible(focus, comp); // add chosen compatible to the
        query
}
```

Listing 4.3: Code for adding a compatible term to a query `q`.

[2]For instance, in an OWL ontology $\top$ is represented by the OWL class `owl:Thing`.

Adding a relation

This operation requires a focus `Node` (as before), a `SchemaRelation` and its range, which is a `SchemaClass`. The new relation must be chosen among those which do not make the query unsatisfiable if attached to the focus node. These compatible relations (which depend on the current query and focus) are obtained by calling the query's `getRelations(Node focus)` method, which for each compatible relation also returns a set of suitable `SchemaClass`'s to be used as range. The chosen relation and its range can then be added to the query with a call to `addProperty(Node focus, SchemaRelation rel, SchemaClass range)`. An example of the above process is shown in Listing 4.4.

```
Collection<Node> nodes = q.getNodes(); // get all nodes in the
    query
Node focus = /* ...
    code for capturing the user choice of an element from "nodes"
... */
Map<SchemaRelation, Set<SchemaClass>> compRels = q.getRelations(
    focus); // get compatible relations and ranges
if (compRels.isEmpty() == false) {
    Set<SchemaRelation> rels = compRels.keySet()); // get
        compatible relations
    SchemaRelation chosenRel = /* ...
        code for capturing the user choice of an element from "
            rels"
    ... */
    Set<SchemaClass> ranges = compRels.get(rel); // get available
        ranges for chosen relation
    SchemaClass chosenRange = /* ...
        code for capturing the user choice of an element from "
            ranges"
    ... */
    q.addProperty(focus, chosenRel, chosenRange); // add chosen
        relation and range to the query
}
```

Listing 4.4: Code for adding a new (compatible) relation to a query `q`.

4.3 Usage Notes for the Graphical User Interface

Although the new implementation concerns only the core of the Query Tool, that is, the reasoning services and the operations for query manipulation, a textual interface has been devised, with the only purpose of testing the behaviour of the system with respect to the input from the user. We will here briefly describe this textual interface, but some screenshots of the new graphical user interface that we currently have in the works can be found in Appendix B.

The version 0.1 of the Query Tool is a command-line application the user interact with which the user interacts by means of the above mentioned textual interface. To run the program (Java 1.5 or higher is required) simply unpack the archive and launch the JAR file from a terminal as follows:

```
java -jar qtool.jar
```

The initial menu of the QueryTool is shown below.

```
+-----------------------------------+
|   QueryTool - version 0.1 (alpha)   |
+-----------------------------------+
[0] Load OWL ontology
[1] Exit
Your choice:
```

In order to load an OWL file, we must first choose the type of reasoner we want to associate with the imported schema. This can be either Pellet or FaCT++ as shown below.

```
[0] Pellet
[1] FaCT++
Your choice:
```

Note that for using FaCT++ the native library for your architecture must be in your java library path.[3] After choosing the reasoner, you will be prompted for the OWL file to load.

```
Filename: test-ontologies/cars.owl
```

The OWL ontology stored in the selected file is imported into a new in-memory schema, which is classified upon loading by the (previously chosen) reasoner associated with it.

```
Loading file 'test-ontologies/cars.owl'... done
FaCT++.Kernel: Reasoner for the SROIQ(D) Description Logic
Copyright (C) Dmitry V. Tsarkov, 2002-2009. Version 1.2.3 (05
    March 2009)
Imported ontology 'http://www.inf.unibz.it/~dongilli/ontologies/
    cars4-tiny'
```

Once the import process is successfully completed, the program shows the following main menu:

```
[0] New query
[1] Show query
[2] Add compatible
[3] Add relation
[4] Select
[5] Delete
[6] Weaken
[7] Substitute
[8] Exit
```

New query In order to create a new query over the imported schema, the user is asked to choose a starting term among all the classes in the schema.

Show query The program pretty-prints the current query tree indenting the children of each node as shown below:

```
[Car]
    has_make [Car_make]
        located_in_country [Country]
    equipped_with [Equipment]
```

[3]See the FaCT++ documentation for more information.

Add compatible First, the program asks for the selection of a focus node, but the root is automatically selected if it is the only node in the query. Then the class to be added has to be chosen from a list of compatible ones.

Add relation First, the program asks for the selection of a focus node, but the root is automatically selected if it is the only node in the query. Then the relation to be added has to be chosen from the list of those which are consistent with the context of the query w.r.t. to the selected focus. If there is more than one class as the range of the relation, then the user is asked to choose from the list of available ones. Otherwise the range is set automatically.

Select The query is printed and each node is associated with an integer. The user is then asked to choose one of the following selection types:

```
[0] Atomic
[1] Single
[2] Complete
[3] Custom
```

In the case of atomic selection, the user is asked to choose a node and a concept name within it; while for single and complete selections, the user is only asked to choose a node. In the case of custom selection, the query is printed and each node is associated with an integer identifier. Then, the user is asked to specify a selection using the following syntax:

$$(\texttt{:<node-id>}\ [\texttt{<concept-id>}]^*)^+\texttt{.}$$

where `<node-id>` is the integer associated with some node and `<concept-id>` is an integer identifying a concept name associated with the specified node. For instance, to select the root and the second and third concept names associated with it, one would write `:0 2 3`. Clearly, at least one (but possible more) node must be specified and if for some node no `<concept-id>` is given, then the chosen node is selected along with all the concept names associated with it.

Delete If a selection has not been set, the program asks for one. The specified selection or the existing one is then checked and whenever admissible (that is, connected) it is deleted from the current query.

Weaken Similar to the case of deletion, but a weakening is performed instead.

Substitute If a selection has not been set, the program asks for one. After checking the sanity of the selection, the user is provided with the lists of concepts that are equivalent to or more specific/general than the selection. The user chooses the substituting term, with which the selection is then replaced.

We conclude the chapter by pointing out that, unfortunately, at present no experiments for evaluating the usability of the new implementation have been carried out yet. However, the system seems to behave well with respect to the changes that have been introduced, especially those concerning the replacement of the reasoning engine, and we strongly believe that the paradigm of interaction with the user of the new GUI (see Appendix B) will greatly improve over the previous version. Clearly, this must be confirmed by new experiments, which will performed as soon as the new graphical interface is finally integrated with the core library (that is, the one described in Section 4.1) and will also allow us to fully estimate the impact on the user of the new features (like the possibility of building complex selection).

CHAPTER 5

Conclusion and Future Work

In this thesis we presented a framework which formally defines the theoretical foundations of the Query Tool, an experimental software meant to support users in the task of formulating meaningful queries over an ontology. We provided definitions which precisely specify the components and the behaviour of the Query Tool, from a mathematical point of view. Moreover, we described how a query can be linearised in a particular sequence of labels, satisfying some constraints which are relevant for a representation in natural language.

A first benefit of our work is given by the fact that, since each operation is thoroughly described in our functional API, it simplifies the task of the developer willing to implement a new system based on the Query Tool framework or extend/improve the existing one.

From the point of view of the implementation, the Query Tool was re-implemented from scratch following the formal specification, resulting in an extensible, better organised and more flexible system. The most important objective we attained is the replacement of the previous DIG-based reasoning engine with the OWL-API, which enabled us to use and take full advantage of the widespread OWL-DL standard endorsed by the W3C. The benefits deriving from the above transition include a faster execution of the reasoning tasks and the possibility of using state-of-the-art reasoners in a straightforward way.

In the rest of this chapter, we would like to go over a few interesting extensions and enhancements that could be possible in the future, in the hope that our work will constitute a starting point as the basis of further development and research.

5.1 Co-references in queries

A remarkable limitation of the current framework, and perhaps the most annoying one, is given by the fact that we can only represent conjunctive queries

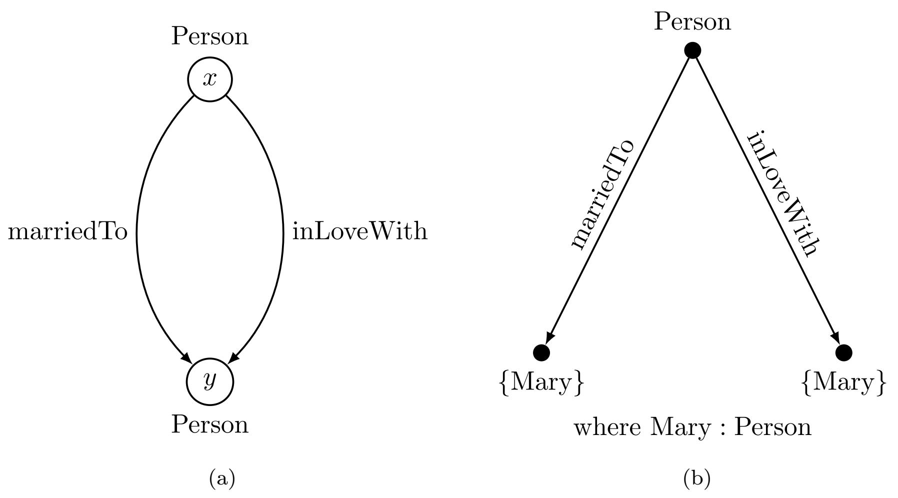


Figure 5.1: Queries with co-references

without co-references. As an example, consider the query:

$$\{x \mid \text{Person}(x), \text{marriedTo}(x, y), \text{Person}(y), \text{inLoveWith}(x, y)\} \tag{5.1}$$

asking for "all the persons married to a person they are in love with". Such a query is not expressible in our framework, as it cannot be represented as a tree. In fact, in order to deal with queries of this form one would need to employ general graphs rather than trees, as shown in Figure 5.1a for (5.1). Unfortunately, it is not possible to encode a query represented as a general graph into a DL concept expression [22, pp. 143–144].

A limited form of co-reference could be achieved through the use of nominals. The general idea we have in the works is shown in Figure 5.1b, which represents as a tree the query asking for "all the persons married to and in love with the individual Mary, who is a person". However, the details of whether, how and to which extent this approach can be cost-effectively exploited in practise are not clear yet.

5.2 n-ary relations

In principle, our framework does not depend on any particular modelling language, as long as the reasoning services we discussed in Chapter 3 are available for it. Thus, we could for example use E-R or UML diagrams as conceptual schemas, but in general these may contain relations other than binary, which would cause problems with respect to our tree-shaped representation of queries.

A standard approach for dealing with n-ary relations is given by *reification* [21], which is widely used in conceptual modelling and consists in viewing a relationship as an entity, called a *weak entity*. The goal of reification is usually that of making a relationship explicit, when additional information needs to be added to it. However, what is more interesting for our purposes is the fact that by means of reification we can actually "decompose" an n-ary relation into n

binary relations. An example is given by the Entity-Relationship diagrams of Figure 5.2, where the ternary relation **Kill** shown in Figure 5.2a is reified into the weak entity **Killing** and decomposed into the three binary relationships **Kill-M**, **Kill-V** and **Kill-W** of Figure 5.2b.

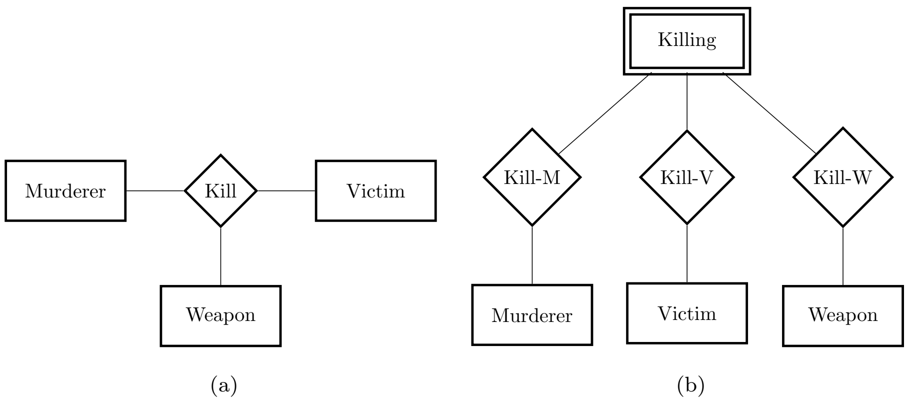


Figure 5.2: Reification of a ternary relationship

5.3 Attributes and datatypes

An extended framework for dealing with *attributes* (that is, properties relating a concept to a datatype) should be quite straightforward. In fact, the only difference with the current framework would be that a node associated with a datatype (i.e., the "range" of the attribute) cannot be the focus of a query for operations other than deletion. This basically means that such a node is always a leaf of the query tree and the only operation allowed on it is deletion. Then, since a node of this kind cannot be refined by adding a compatible term or attaching a new property, the query is never rolled-up with respect to it, thus avoiding the nonsensical eventuality that an edge associated with an attribute has to be inverted (going from the datatype to the subject).

The inclusion of these additional restrictions into the existing framework should pose no particular problem and it is just a matter of working out the technical details. In fact, in addition to *object properties* (or *abstract roles*), the DL language $\mathcal{SROIQ}(\mathcal{D})$ (that is, $\mathcal{SROIQ}$ extended with datatypes) supports also *data properties* (or *concrete roles*), hence the possibility of representing attributes is directly available. As for the implementation, although full support for dealing with attributes has to be added yet, their future presence has been already taken into account in the design of the Query Tool, as Figure A.1 witnesses.

APPENDIX A

Class Diagrams

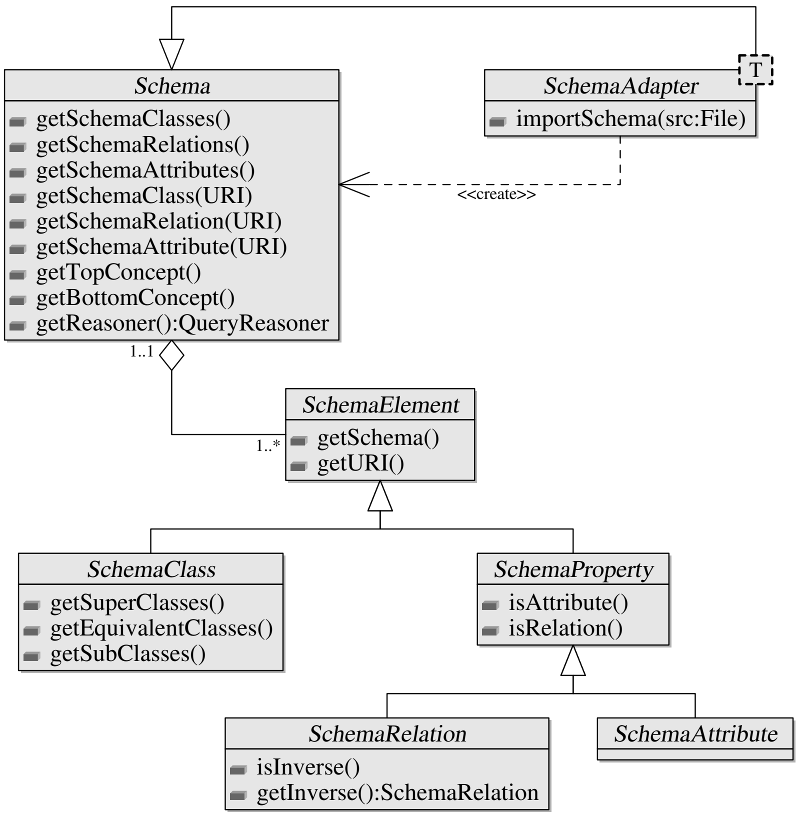


Figure A.1: Class diagram for the package `it.unibz.qtool.model`

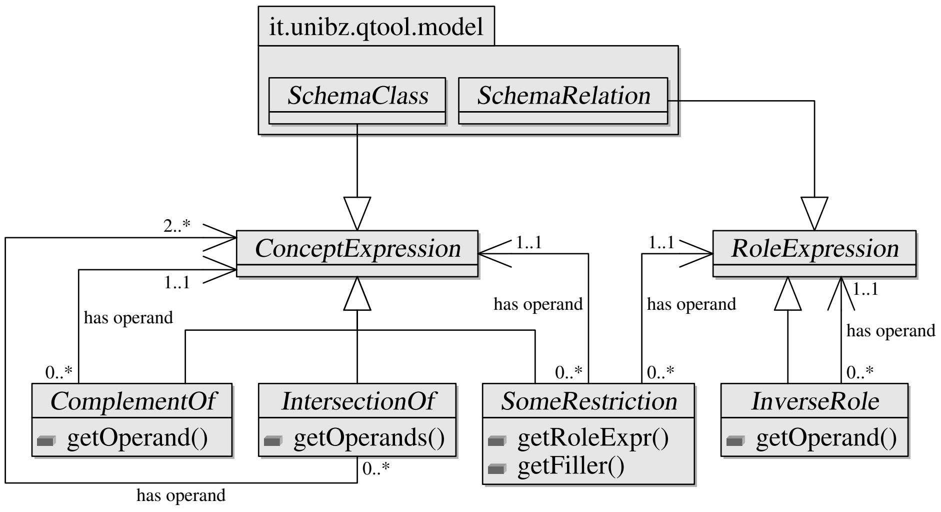


Figure A.2: Class diagram for the package `it.unibz.qtool.lang`

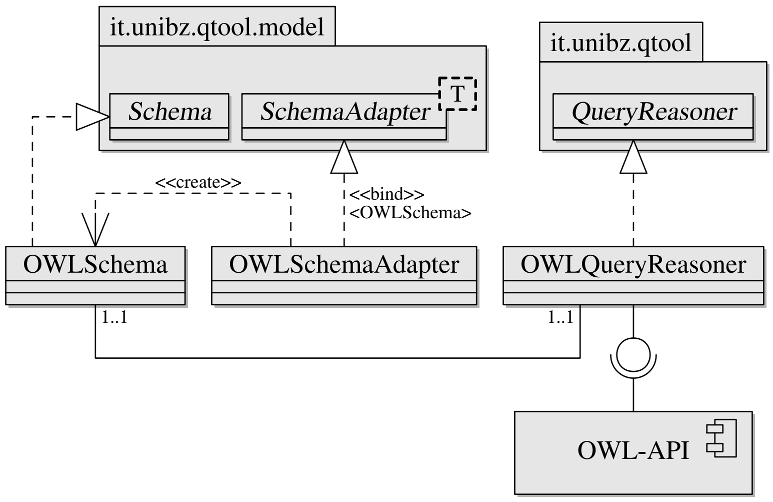


Figure A.3: Class diagram for the package `it.unibz.qtool.owl`

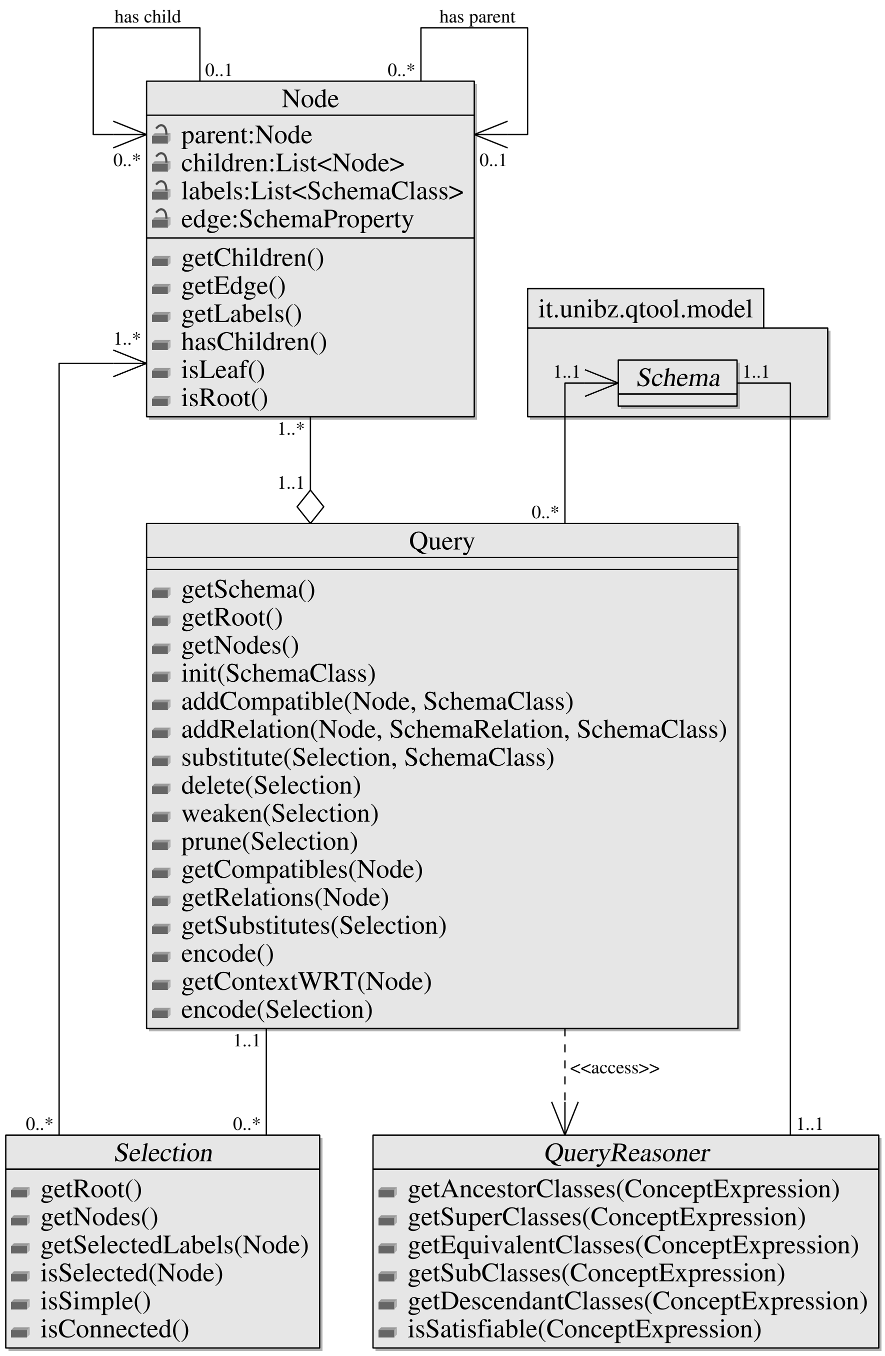


Figure A.4: Class diagram for the base package `it.unibz.qtool`

APPENDIX B

GUI Screenshots

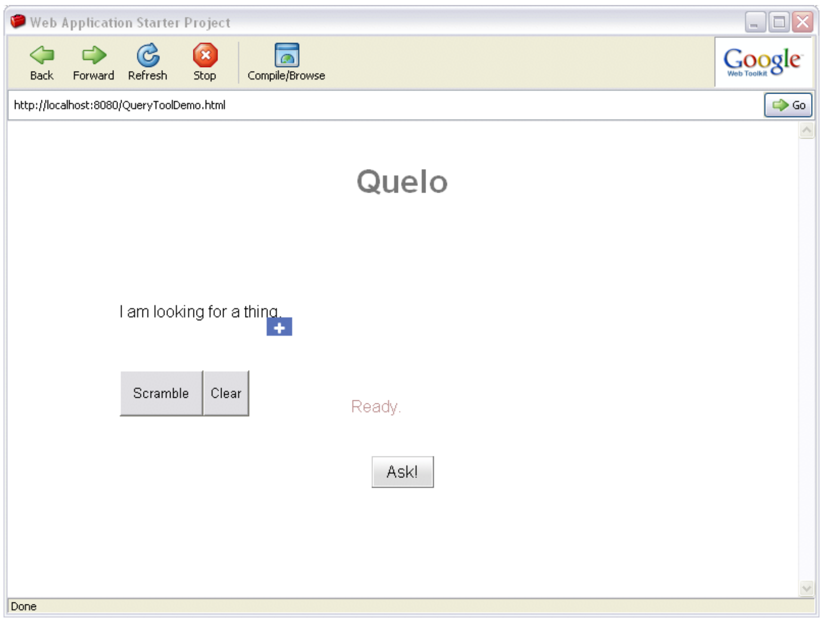

Web Application Starter Project
Back
Forward
Refresh
Stop
Compile/Browse
Google Web Toolkit
http://localhost:8080/QueryToolDemo.html
Go
Quelo
I am looking for a thing.
Scramble
Clear
Ready.
Ask!
Done

Figure B.1: The starting atomic query

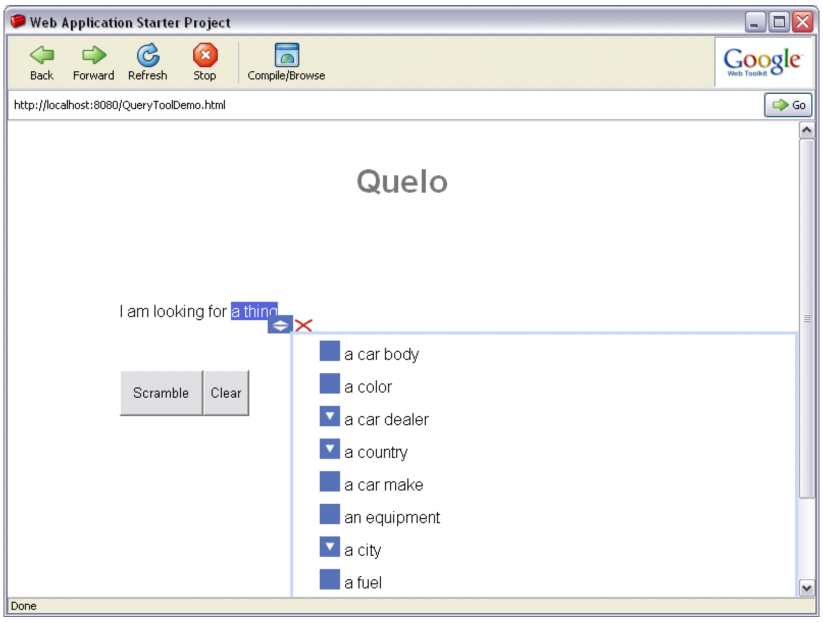


Figure B.2: The menu for substitution includes equivalent, more general and more specific terms. Note also the presence of a button for deletion.

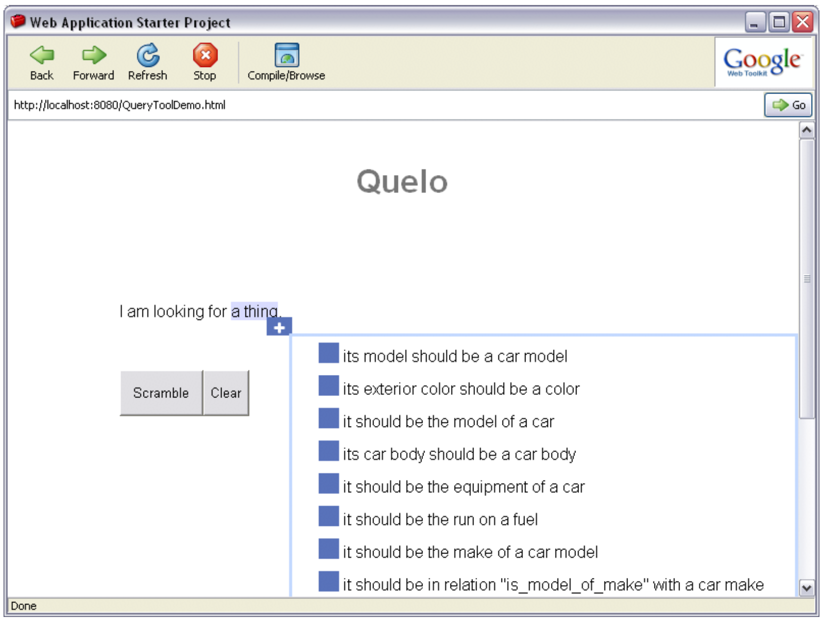
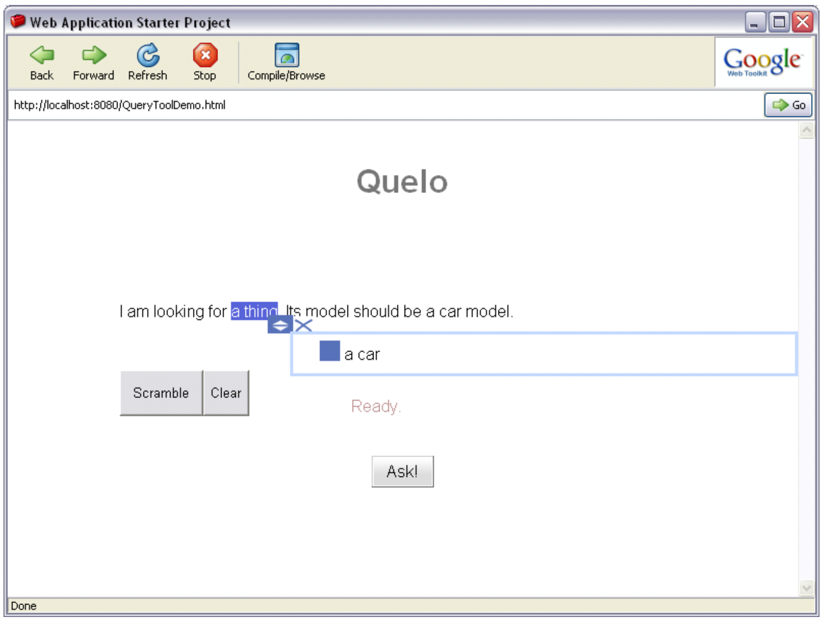

Web Application Starter Project
Back
Forward
Refresh
Stop
Compile/Browse
Google Web Toolkit
http://localhost:8080/QueryToolDemo.html
Go
Quelo
I am looking for a thing.
Scramble
Clear
its model should be a car model
its exterior color should be a color
it should be the model of a car
its car body should be a car body
it should be the equipment of a car
it should be the run on a fuel
it should be the make of a car model
it should be in relation "is_model_of_make" with a car make
Done

Figure B.3: The menu for addition includes compatible terms and relations.



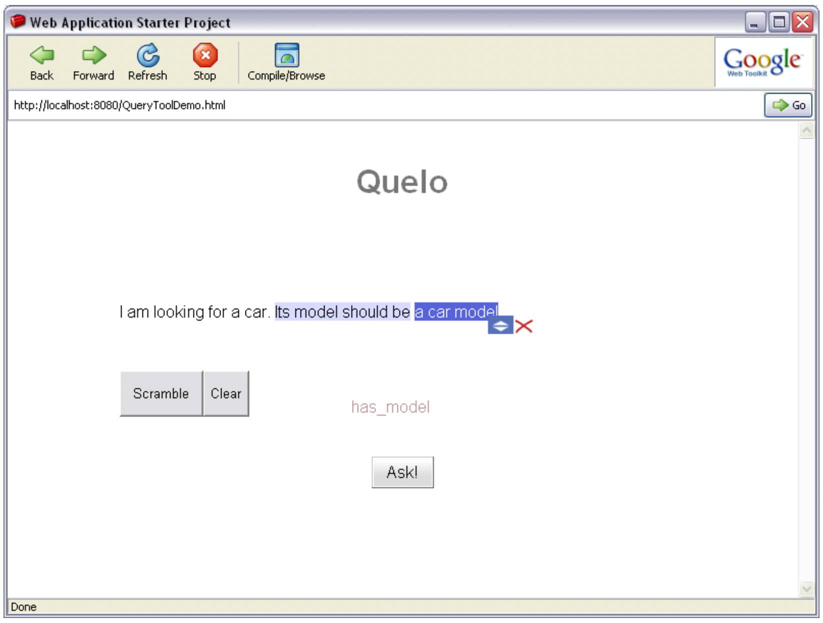
http://localhost:8080/QueryToolDemo.html

I am looking for a thing. Its model should be a car model.

a car

Ready.

Figure B.4: Only "car" is available as a substitute for "thing", because the thing we are looking for has a property "has model" with range "car model", and the system derives that it must be a car. Compare the results to those in Figure B.2.



http://localhost:8080/QueryToolDemo.html

I am looking for a car. Its model should be a car model.

has_model

Figure B.5: On mouse over, the portion of the query that is related to the focus (dark blue) is highlighted (light blue).

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995.

[2] T. Catarci, T. D. Mascio, P. Dongilli, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In ECAI 2004 [14].

[3] T. Catarci, T. D. Mascio, P. Dongilli, E. Franconi, G. Santucci, and S. Tessaris. An Ontology-Based Query Manager: Usability Evaluation. In HCItaly2005 [16], pages 95–100.

[4] T. Catarci, T. D. Mascio, P. Dongilli, E. Franconi, G. Santucci, and S. Tessaris. Usability evaluation in the SEWASIE (SEmantic Webs and AgentS in Integrated Economies) project. In HCII 2005 [15].

[5] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC '77: Proceedings of the ninth annual ACM symposium on theory of computing*, pages 77–90, New York, NY, USA, 1977. ACM.

[6] B. Davey and H. Priestley. *Introduction to Lattices and Order*, chapter 1. Cambridge University Press, second edition, 2002.

[7] M. Dean and G. Schreiber. OWL Web Ontology Language Reference. W3C Recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[8] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*, chapter 1. Springer, second edition, 2000.

[9] *Proceedings of the 2004 International Workshop on Description Logics (DL2004)*, Whistler, BC, Canada, 2004.

[10] P. Dongilli. Natural Language Rendering of a Conjunctive Query. Technical Report KRDB08-3, KRDB Research Centre, Faculty of Computer Science, Free University of Bozen-Bolzano, June 2008.

[11] P. Dongilli, P. R. Fillottrani, E. Franconi, and S. Tessaris. A multi-agent system for querying heterogeneous data sources with ontologies. In SEBD-2005 [23].

[12] P. Dongilli, E. Franconi, and S. Tessaris. Semantics driven support for query formulation. In DL2004 [9].

[13] P. Dongilli, S. Tessaris, and J. A. Bateman. Leveraging Systemic-Functional Linguistics to Enhance Intelligent Database Querying. In ISDA'06 [18].

[14] *Proceedings of the 16th Biennial European Conference on Artificial Intelligence (ECAI 2004)*, Valencia, Spain, 2004.

[15] *Proceedings of the 11th International Conference on Human-Computer Interaction (HCII 2005)*, Las Vegas, Nevada, USA, 2005.

[16] *Proceedings of the 4th Italian Symposium on Human Computer Interaction (HCItaly2005)*, Roma (Italy), 2005.

[17] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible $\mathcal{SROIQ}$. In *Proc. of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR2006)*, pages 57–67. AAAI Press, June 2006.

[18] *Proceedings of the Sixth International Conference on Intelligent Systems Design and Applications (ISDA'06)*, Jinan, China, 2006.

[19] J. J. Kelly. *The essence of logic.* Prentice Hall, 1997.

[20] M. Kilp, U. Knauer, and A. V. Mikhalev. *Monoids, Acts and Categories*, volume 29 of *Expositions in Mathematics*, chapter 1, pages 1–13. de Gruyter, 2000.

[21] A. Olivé. *Conceptual Modeling of Information Systems*, chapter 6, pages 123–134. Springer Verlag, 2007.

[22] U. Sattler, D. Calvanese, and R. Molitor. *The Description Logic Handbook: Theory, Implementation and Applications*, chapter 4, pages 137–177. Cambridge University Press, 2003.

[23] *Proceedings of the Thirteenth Italian Symposium on Advanced Database Systems (SEBD-2005)*, Brixen-Bressanone, Italy, 2005.

[24] The SEmantic Webs and AgentS in Integrated Economies (SEWASIE) project. `http://www.sewasie.org/`, 2005.

[25] I. Zorzi. An Ontology-Based Visual Tool for Query Formulation Support. Bachelor's thesis, Faculty of Computer Science, Free University of Bozen-Bolzano, 2005.

[26] I. Zorzi. Improving Responsiveness of Ontology-Based Query Formulation. Master's thesis, Faculty of Computer Science, Free University of Bozen-Bolzano, March 2008.