CGU: A common graph utility for DL Reasoning and Conjunctive Query Optimization

by

Jesus Alejandro Palacios Villa

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

Abstract

We consider the overlap between reasoning involved in *conjunctive query optimization* (CQO) and in tableaux-based approaches to reasoning about subsumption in *description logics* (DLs). In both cases, an underlying graph is created, searched and modified. This process is determined by a given *query* and *database schema* in the first case and by a given *description* and *terminology* in the second. The opportunities for overlap derive from an abundance of reductions of various schema languages to terminologies for common DL dialects, and from the fact that descriptions can in turn be viewed as queries that compute a single column.

Our main contributions are as follows. We present the design and implementation of a common graph utility that integrates the requirements for both CQO and DL reasoning. We then verify this model by also presenting the design and implementation for two drivers, one that implements a query optimizer for a conjunctive query language extended with descriptions, and one that implements a complete DL reasoner for a feature based DL dialect.

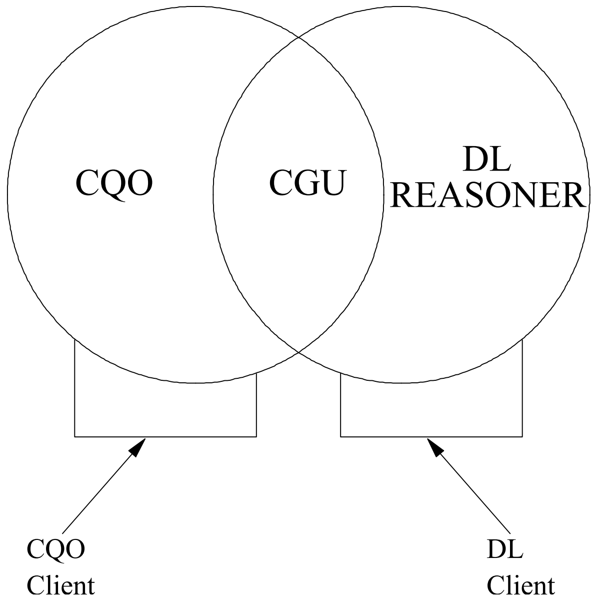

Figure 1: CGU CONCEPT DIAGRAM

Acknowledgments

After completing this significative goal, I want to acknowledge some of those that were important for accomplishing this task. I know the list is endless so I will try to be as brief and concise as possible.

First I would like to thank God for allowing me to get here, living the life I have lived, meeting the people I have met and receiving the gifts I have received. I also want to thank my wife Yalina for staying countless hours by my side during this stage, enjoying the good moments and persevering through the hard times and for putting her professional goals aside while I was going through this stage. For her, I give my love, my admiration and appreciation. To my parents, for all their support, care and love, for all their teachings and all the time invested in us all these years. They are still our model of family that now my wife and I would like to follow. To our grand mother Maggie, for her love and dedication, and for being our presence in Mexico when it was needed. To all our family in Mexico, for keeping in touch with us and encouraging us to keep going while also being anxious for us to be back with them.

I also want to thank my supervisor, professor Grant Weddell, for leading the way during my research work, for showing me so many different points of view of the database area and for investing so much time and interest during the realization of this project. Also my gratitude to him as a friend, for sharing so many anecdotes that enriched our conversations, and for showing me how sometimes one can talk and transmit so much with emotion and so little with words. To my course professors, David, Peter, Frank, Alex, Chrissane and Anne, for the enjoyable class experience I was able to experience.

I also want to thank the CONACyT (Science and Technology National Council) in Mexico, for sponsoring my studies at the University of Waterloo, allowing me to achieve this goal in my life and reinforcing my commitment to the development and progress of my country and my people. To the University of Waterloo, for providing such a rich environment where studying and researching is done with complete freedom of mind.

And last but not least, thanks to my friends who were part of my day to day life during these two years. Many, many thanks to the Chinaei family, Amir, Leila and Hamid, my friends and office mates, for sharing this time with me, helping me when I needed the most and sharing with me all they have. I wish them the best success in their future degrees and careers. Special thanks also go to Carlos, Claudia, Gabriel, Flor, Alberto and all our Latin-American friends who became our family while we were here. Thanks to Salvador, Susy and Ivan, who supported me and encourage me to come and help me staying. To all the members of our friendship group from ITESM, the mythical “chapubanda”, for enjoying as a team what each one of us has achieved individually and for all the funny and pleasant moments we have shared together. And thanks to Larry, Rick, and all the people in IHS for helping me to succeed in this stage and receiving me back now that the goal is completed.

To all of them and the ones I did not mention explicitly, many, many thanks.

Dedication

This thesis is dedicated to the memory of our beloved Betyna who unfortunately passed away in April of 2005. She did not have a chance to see this work completed but we are sure she would have enjoyed this milestone in our lives as much as we do. We will never forget her support and the happiness she brought to our lives. You will always be in our heart.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Overview

1.1 Tableaux in Conjunctive Query Optimization

We consider the overlap between reasoning involved in *conjunctive query optimization* (CQO) and in tableaux-based approaches to reasoning about description containment, called subsumption, in *description logics* (DLs). In both cases, an underlying graph is created, searched and modified. This process is determined by a given *query* Q and *database schema* $\mathcal{S}$ in the first case, and by a given *description* D and *terminology* $\mathcal{T}$ in the second.

In symbolic terms, the underlying problem in CQO can be abstracted as the following sentence:

$$\mathcal{S} \models (Q \sqsupseteq QP) \text{ and } (Q \sqsubseteq QP) \tag{1.1}$$

where QP denotes a low-level executable query, hereon called a *query plan.* In this formulation, the "$\sqsubseteq$" symbol is intended to suggest that the table specified by Q is a subset of the table computed by QP, while the "$\sqsupseteq$" symbol is intended to suggest the opposite, that the table computed by QP is a subset of the table specified by Q. Overall, the sentence states that the query plan correctly implements the query for any database satisfying the conditions imposed by the database schema $\mathcal{S}$.

The query optimization problem thus deals with finding a plan that correctly and efficiently implements a query. This problem has been dealt with in several ways. The whole process of finding a plan has been split in several stages and quite a few techniques have been developed to solve each one of those. One of the earliest techniques that tries to simplify the query while still in the logical schema is called the tableau chase, as described in [1]. Formally, a tableau query (T, u) is: (1) a set of tuples T containing variables and constants, and (2) a free tuple u, called the summary, consisting of variables occurring in T[1]. The tableau chase is a procedure where given a tableau query (T, u) and a set of dependencies Σ, the dependencies in Σ are applied to the query (T, u) in some order. This sequence is guaranteed to terminate (to be finite) if no new variables are created as a result of the application of the dependencies and can be infinite otherwise. The key reason for applying the dependencies to the query is that if the relation(s) to which the query is applied must satisfy the dependencies given by the schema, then the query itself must satisfy the same dependencies. Enforcing the dependencies directly on the tableau will usually result in a simpler (and in that sense more optimal) query to execute while still being equivalent to the original query. Another query optimization technique called *semantic query rewriting* consists of, given a query expressed in some abstract form, usually an algebra, rewrite the query in a semantically equivalent form that may result in a cheaper execution plan. The rewrite of the query is done by applying a set of semantic preserving rules whose application is sometimes driven by actual cost functions and sometimes by simple heuristics. For example, it is equivalent to make two selection steps or performing a single selection step with the clauses of both selections combined; for two projection steps, combine them into a single projection to obtain the same result; it is the same (semantically speaking) to do a projection and then a selection than to do the selection first and then the projection; and so on. A third approach we consider relevant for our work is presented in [2]. There, Deutsch et al describe a semantic optimization algorithm designed to work considering the physical data independence property. In summary, their approach works as follows: Given (1) a logical schema Λ, (2) a physical schema Φ, (3) an implementation mapping $\mathcal{M}$ (from the logical to the physical) and, (4) a query Q in the logical schema, find a query plan Q' that when executed against the physical schema it will implement exactly Q (see Figure 1.1).

The algorithm proposed is a two phase run where the first phase consists of the augmentation of the query with all the physical structures and access paths given by the constraints, resulting in a *universal plan*; and then the second phase removes redundant information from the plan (given the constraints and the query itself) until the minimal query plan is found. A more detailed explanation of the algorithm is presented in section 2.1.

[1]In general, every tuple in $\mathcal{T}$ represents a relation (from the relational model) in the query.

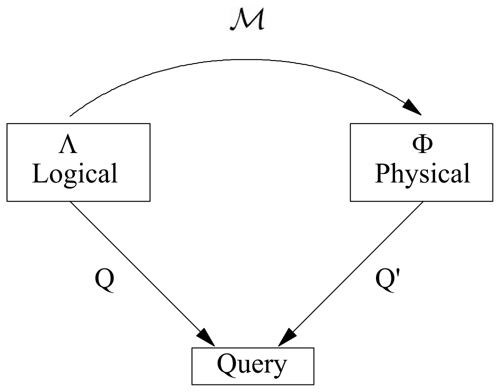

Figure 1.1: Diagramatic description of the physical data independence approach to semantic query optimization

1.2 Tableaux in Description Logics Reasoning

In contrast to the CQO problem (1.1), the general formulation of the subsumption problem in a DL is abstracted as follows:

$$\mathcal{T} \models (D_1 \sqsubseteq D_2). \tag{1.2}$$

where D_1, D_2 are general descriptions and $\mathcal{T}$ is DL a terminology. To understand the relevance, consider that a database schema can be reduced to a terminology in common DL dialects, and that a description D_i can be viewed as a query that evaluates to a table consisting of a single column. Thus, this sentence expresses the condition that D_1 evaluates to a subset of D_2 for any database satisfying the conditions imposed by the terminology $\mathcal{T}$. For expressive enough dialects, i.e. those that are boolean complete, formula 1.2 can also be expressed as

$$\mathcal{T} \models (D_1 \sqcap \neg D_2) \sqsubseteq \bot \tag{1.3}$$

where the symbol "$\sqcap$" denotes conjunction of descriptions, the symbol "$\neg$" denotes negation and the symbol "$\bot$" (read as "bottom") denotes the concept that has no elements (nothing). Note that it is only in boolean complete DLs that (1.2) and (1.3) are interchangeable since $\neg D_2$ requires negation of general concepts (as opposed to just primitive concepts). Then (1.3) is equivalent to checking if $(D_1 \sqcap \neg D_2)$ is *satisfiable.* Satisfaction checking and subsumption are closely related services provided by DL reasoners. As it will be seen in more detail in section 2.2, a problem expressed as a subsumption of descriptions can be transformed into a satisfaction check for a description that conveys the same meaning using the general identity:

$$D_1 \sqsubseteq D_2 \iff D_1 \sqcap \neg D_2 \text{ is unsatisfiable}$$

The alternative formulations, 1.2 and 1.3 have led, respectively, to *structural testing* and *satisfiability testing* approaches to DL subsumption problems, with both techniques incorporating some form of tableau reasoning procedure.

The structural testing approach was one of the first approaches to deal with subsumption before DL dialects with general negation emerged. Consider for example [3]. In that paper, Borgida et al use a DL language called CLASSIC that does not support negation of general concepts and therefore does not support disjunction. They show that, determining whether D_2 subsumes D_1, expressed in CLASSIC, requires a three phase process: (1) Converting D_1 into a description graph G_{D_1} using a similar set of transformation rules for the tableau in the previous case, (2) normalizing G_{D_1} by removing redundant information from the graph and finally (3) verifying whether D_2 subsumes D_1 by recursively applying a structural checking for the presence of D_2 in the graph G_{D_1}. Another similar example is found in [4] where Khizder et al. define a comparable algorithm for a dialect called CFD. The dialect CFD (or CLASSIC with Functional Dependencies) does not support number restrictions for Roles, but it adds a basic constructor to express path functional dependencies. In this approach, to verify the subsumption of D_1 by D_2, a graph is also created from a single individual and the initial description D_1 deposited in it. This graph is then "expanded" using a set of transformation rules with the same flavour as the ones presented in [3] but augmented to consider the possible existence of a terminology expressed as a set of subsumption constraints S. Once the graph is expanded, again a structural check is performed, but now with the appropriate considerations to handle the path agreements for the functional dependencies case. What we can see from both approaches, is that subsumption checking in these DLs basically involves (1) an expansion phase for a graph using a set of deterministic rewrite rules and then (2) a structural checking for the subsumer[2] description. As we will see below, the expansion required in (1) is then very similar to the tableau algorithms used for satisfiability testing.

For DL languages with full negation, the subsumption problem is reduced to a satisfiability test that in turn is done using tableau algorithms. Please note that the tableau referred here and the tableau (T, u) from the tableau chase procedure are two apparently different concepts that we will later unify them to refer to the

[2]In $D_1 \sqsubseteq D_2$, D_1 is called the subsumee and D_2 is called the subsumer.

same abstract element: a graph. In DL, a *tableau* can be thought of as a graph representing the information contained in the descriptions where nodes represent hypothetical individuals of the domain, labels in the nodes denote descriptions, and edges signal the existence of relations (roles or attributes) between nodes. Such algorithms create first a tableau with a single individual belonging to the description to be checked (e.g. $D_1 \sqcap \neg D_2$). Then they apply a set of consistency preserving transformation rules[3] that "expand" the tableau (creating new nodes and edges) based on the conditions presented by each of a possible variety of concept *constructors* (description generating operators) used in the description. To deal with disjunction of descriptions, the transformation rules introduce choice to encode a search for a consistent tableau. A tableau is consistent if it does not have an obvious contradiction, called a *clash* (like a single individual belonging to both C and $\neg C$). When the expansion rules can no longer be applied, the algorithm terminates, and if at least one of the tableaux generated in the expansion is consistent, then the original description is consistent (satisfiable).

1.3 Database Schema and Terminologies

As mentioned before, we can understand D_1 in Equation 1.2 as a single column query that, when evaluated, produces a subset of the table produced by D_2. Likewise, the terminology $\mathcal{T}$ can be understood as the representation of the constraints presented in the Database Schema $\mathcal{S}$. In fact there have been several approaches aimed to transform DB schemas into DL terminologies as shown in [5]. This can be observed, for example in [6, 7, 8] where the authors provide formal models of object-oriented databases using DL. Furthermore, in [9, 10, 11] semantic data models based directly in DL are introduced and also, Borgida et al. present in chapter 16 of [5] a set of rules to transform Entity-Relationship schemas into knowledge bases using the Description Logic $\mathcal{DLR}$ dialect. In this last approach the dialect $\mathcal{DLR}$ is chosen because it extends traditional DL semantics in a way that allows the modeler to capture not only binary relations but also relations of arbitrary arity. This transformation basically consists of a mapping from the elements in the ER schema to the set $\mathcal{T}$ of concepts, roles and descriptions in $\mathcal{DLR}$. In particular, every entity is assigned a primitive concept, every attribute in the ER schema is assigned a primitive binary relation in $\mathcal{T}$ and every ER relation is assigned a primitive n-ary relation in $\mathcal{T}$. Additionally, entity and relation specializations (IS-A) in the ER schema are converted to subsumption of their respective concepts and

[3]For example, for a list of the rules used for $\mathcal{ALCN}$ please see chapter 2 of [5]

relations in $\mathcal{T}$; attribute typings are converted to qualifying restrictions; and cardinality constraints (except for 0 and ∞) are transformed to number restrictions. As presented in [8], it can be formally proved that such mapping is correct and that there exists a one-to-one correspondence between legal database states conforming to the ER schema and models[4] of $\mathcal{T}$.

1.4 Summary and Overview of Thesis

In general, the optimization procedures consist of representing the query in some abstract way (a tableau, an algebra expression or a graph) and then manipulating it (usually applying some set of rewrite rules) until an equivalent query is found that is cheaper to execute. Moreover, if we would abstract the query from the approach in [2] as a graph, their rewrite rules will have a similar effect to the ones we see for reasoning in DL in the presence of a terminology and this is precisely what motivates us to explore the similarities between both approaches.

It is important to note that the merging of these two areas of research is having an increased interest due to the work required to deal with one of the largest databases available, the World Wide Web, or simply the Web as it is commonly referred. Of special interest is the work done in the Semantic Web area and in particular in the OWL and OWL DL projects. Our work could be used as a possible line of research in the area of query processing for OWL. We will discuss this further in the chapter 5 when we talk about future work.

The rest of this thesis is organized as follows: In section 1.5 we present the reader with a panoramic view, from databases in general to the focus of our research. Some readers might find this very basic and can skip it without detriment of their understanding of the rest of the content presented. In chapter 2 we briefly present the relevant work including the CQO approaches, query rewriting, a review of DL and tableau chase procedures and the common DL reasoners; in chapter 3 we present the architecture and design of our approach, we define our DL language and our query language plus the interface used for DL reasoning and CQO; in chapter 4 we reveal the algorithmic details of the main modules and will exhibit our main contribution explaining the behavior of our dual-purpose chase procedures and sketching the algorithm for the query plan generation using our expand functions; and finally in chapter 5 we discuss the conclusions and we establish our proposed lines for future work.

[4]Model is an interpreation that satisfies a terminology in DL. See section 2.2 for a more precise definition of "model" in DL.

1.5 Putting things in perspective

Databases have been one of the most extensively researched areas in computer science history and one of the most pervasive applications in the computing industry. This is no surprise if we consider that currently, databases are usually at the core of practically all of the data driven applications in most companies or institutions. Databases involve nearly every aspect of any organization, from human resources and accounting, to sales, order management and inventory control. Furthermore, databases are so basic and so essential that nearly every personal computer and even smaller computing devices have some type of database system in their application set to perform simple operations like calendar and agenda management and it has been explored to be used even for OS level operations. In general, database systems should be able to store large bodies of data and to manage them efficiently. Among the data management operations, arguably the most intensively used one is the query operation, i.e. the retrieval of data from the database meeting certain characteristics commonly expressed in terms of a particular language. A query is then the specification expressing the characteristics of the data sought in the database. To answer these queries, it is not trivial to decide what operations and in what order should they be performed against the actual stored data. The factors for this are on one hand that databases are designed to achieve logical and physical data independence, and on the other hand, query languages usually allow the user to express a particular query in many different ways. While each of the different ways to express a query may be a suggestion of how to execute it, the actual execution of the query should be independent of the query itself (as long as it returns the same result). This allows the database system to optimize the execution plan. In other words, regardless of how the query is expressed, the system should find a fast and efficient plan to answer it. This is called query optimization.

There have been many approaches and refinements in the area of query optimization during the last 20 years, particularly in the relational and object oriented models. Most of them rely on a process where the query is losslessly translated into some type of algebra expression and then it is transformed into an equivalent expression (if one exists) that is more efficient to execute. This step is usually referred to as the semantic query optimization. After such expression has been found, a detailed strategy to execute the query, i.e. the plan, is devised, mainly on the basis of the cost for the different access paths available to the data being queried. In order to do this, a very common approach used is to create a query graph and perform some operations on that graph to create the plan. An alternative approach proposed for the semantic optimization step, is the translation of the query, or at least part of it, into a *Description* expressed in some Description Logic (DL) dialect (an AI knowledge representation language) and then classify it with respect to schema concepts and views [12]. The classification process basically returns a more specific [stored] view (if one exists) such that when used in the plan it will avoid the execution of the view as part of the query, effectively reducing the whole execution time. After this, the plan is still generated in the usual fashion. This classification is commonly done using the reasoning capabilities of a standard DL reasoner that in turn is implemented using some sort of tableau (graph) representing the concept descriptions and roles or attributes and a rule based algorithm dealing with the descriptions presented in that graph. DL reasoning primarily refers to the deductive services provided by the language that allows the user to pose questions about the subsumption between concepts, effectively creating a hierarchy, i.e. a classification.

Yet another approach to query optimization proposed by [2] is that given the query and the schema including semantic constraints and the access paths to the data, the query is rewritten into a "universal" plan during a chase phase (usually called the "grow" phase) and then a back chase phase removes the redundancies from the plan according to the constraints (the "shrink" phase), producing an optimal plan.

Our work then was based on the observation that what happens during the grow phase of Conjunctive Query Optimization (CQO) is essentially similar to what happens during the deterministic part of a tableaux based DL reasoner, for example while it is determining subsumption of concepts. We believe that the graph created for CQO is essentially similar to the tableau created during the expand phase of DL reasoning. Additionally, we also believe that the shrink phase of CQO can be refined to a differential approach that in turn is very similar to satisfaction checking for DL. So with this in mind we wanted to explore the similarities between CQO and tableaux chase reasoning.

Chapter 2

Review

We now present the overview of the topics we consider relevant for this thesis. Please bear in mind that this is not in any way an extensive sumary of the research done in the conjunctive query optimization. This review can be seen as the tip of the iceberg for the work done in the tableau optimization for conjunctive queries. We begin by presenting some notation on conjunctive queries before presenting the main work from Deutsch an coleages about query optimization. We then review the basic concepts for DL and conclude with a brief overview of DL reasoners.

2.1 Conjunctive Query Optimization

Conjunctive Queries are probably the simplest type of queries appearing in a broad class of applications. However, while simple they are yet interesting enough to be studied. Informally, conjunctive queries are those that can be constructed using only "and", equality and joins (plus possible projection) so they are commonly referred as SPJ queries, in relation to the operations allowed (selection, projection and join). Formally they can be defined in several ways, using rule-based queries, tableau queries, conjunctive calculus, etc. For example, the general form of a conjunctive query in the rule based approach is:

$$ans(e_1, \ldots, e_m) \leftarrow R_1(u_1), \ldots, R_n(u_n)$$

and the equivalent query in the conjunctive calculus is:

$$\{e_1, \ldots, e_m | \exists x_1, \ldots \exists x_k (R_1(u_1) \wedge \ldots \wedge R_n(u_n))\}$$

where $x_1, \ldots, x_k$ are variables occurring only within the variables set of $u_1, \ldots, u_n$, (the body in the rule based approach), $e_1, \ldots, e_m$ are the selected (projected) variables and $R_1, \ldots, R_n$ are the relations joined. For a detailed presentation of these, please refer to chapter 2 in [1]. Our own choice of language for conjunctive queries is presented in chapter 3.

Some of the desirable properties conjunctive queries possess are the decidability of equivalence and containment, one of the reasons why we chose them to be contrasted against our choice of DL language, since the latter also enjoys the decidability property. Query containment refers to the question *Is query Q_1 contained in query Q_2?* We say that indeed $Q_1 \subseteq Q_2$ (Q_1 is contained in Q_2) if for all databases, the result of Q_1 is a subset of Q_2. As one might also expect, the two queries are equivalent, $Q_1 = Q_2$, if and only if each of them is contained in the other.

$$Q_1 = Q_2 \Longleftrightarrow Q_1 \subseteq Q_2 \wedge Q_1 \supseteq Q_2 \qquad (2.1)$$

As cited in [13], it turns out that in almost all cases, the only approach known for testing equivalence is by testing containment in both directions, i.e. the literal application of rule 2.1.

One of the main applications where query containment and equivalence tests are highly used can be found in the information integration area. In a typical setting, answers to queries have to be computed from the sources abstracted as views. Here a query submitted to the integrator system has to be computed in terms of the views (sources), and for the answer to be valid, the expansion of the views with their respective definitions has to be equivalent to the original query submitted. The generation of such possible answers, also queries, is called “synthesizing queries from views”[1]. This problem is then equivalent to generate all the possible combination of views containing the relations in the original query up to length less than or equal to the length of the query and test for equivalence. The generation of such possible answers has been dealt with in several ways as can be seen in the Information Manifold project and the Tsimmis project[2].

The other important use of query containment and equivalence is in the query optimization area where typically a query submitted in a particular language gets rewritten using some equivalence preserving rules in an attempt to make it simpler or more efficient to execute. There has been a lot of research in this area but we will briefly present below one of the branches that is of interest for our research work.

[1]See [14] for a theoretical exploration of the issue of Answering queries using views.
[2]See [13] for a comparison between the two projects.

In [2], Deutsch et al. propose a semantic optimization algorithm based on the physical data independence property, i.e. that the way data is physically encoded and accessible should be independent from the logical schema and the query formulation. To achieve this property, it is agreed that the problem should be expressed as follows: Rewrite a query Q written against a logical schema Λ into an equivalent query plan Q' written against a physical schema Φ given a semantic relationship between Λ and Φ. They also point out that to characterize precisely this semantic relationship there are two main approaches. The first one is to define an abstraction mapping $\mathcal{A}$ from Φ to Λ that expresses the instances of the logical schema in terms of those of the physical schema. This is similar also to the Global As View (GAV) perception used to characterize local and global schemas in an information integration setting if we consider the logical schema as the Global schema and the physical schema as the local schema [3]. The second approach is to define an implementation mapping $\mathcal{M}$ from Λ to Φ that now delineate the physical instances in terms of the logical schema, again similar to the Local As View (LAV) approach from information integration if using the same consideration as before (see again Figure 1.1).

In the first case, finding the plan Q' can be as simple as composing the query Q with the abstraction mapping $\mathcal{A}$. However, note that this will result in more than one plan if several mappings are defined for some or all of the elements in the logical schema. In that case an optimization phase is still required that evaluates those alternative plans' cost and then choose the cheapest. In the second case, finding the plan Q' means solve for X the equation $X \circ \mathcal{M} =_\Lambda Q$, where at the end $X = Q'$. This latter approach is the one used in [2], where the physical access structures, materialized views and other elements of the implementation mapping are captured using constraints. The authors then propose a two-phase algorithm to solve X that works as follows:

1. *"Chase Phase"*. Systematically bring all the relevant physical structures into the logical query by repeatedly applying a set of rewrite rules based on the applicable constraints until they no longer apply. A simplified example of a rewrite rule will be:

$$\begin{array}{lcl} \underline{\text{select}}\ O(\vec{r}) & & \underline{\text{select}}\ O(\vec{r}) \\ \underline{\text{from}}\ \ldots,\ R_1\ r_1, \ldots,\ R_m\ r_m, \ldots & \Rightarrow & \underline{\text{from}}\ \ldots,\ R_1\ r_1, \ldots,\ R_m\ r_m,\ S_1\ s_1, \ldots,\ S_n\ s_n, \ldots \\ \underline{\text{where}}\ \ldots\ \underline{\text{and}}\ B_1\ \underline{\text{and}}\ \ldots & & \underline{\text{where}}\ \ldots\ \underline{\text{and}}\ B_1\ \underline{\text{and}}\ B_2\ \underline{\text{and}}\ \ldots \end{array}$$

   The result of this phase will be a rewritten query that will basically hold all the possible physical plans expressible using the language suggested. This

[3]See [15] for a thorough discussion of GAV and LAV in an information integration environment.

plan is also referred as the Universal plan U.

2. *"Backchase Phase"*. Repeatedly apply the rewrite

$$\begin{array}{lcl} \underline{\text{select}}\ O(\vec{x}, y) & & \underline{\text{select}}\ O'(\vec{x}) \\ \underline{\text{from}}\ R_1\ x_1, \ldots,\ R_m\ r_m,\ R\ y & \Rightarrow & \underline{\text{from}}\ R_1\ x_1, \ldots,\ R_m\ r_m \\ \underline{\text{where}}\ C(\vec{x}, y) & & \underline{\text{where}}\ C'(\vec{x}) \end{array}$$

provided that the remaining conditions, selections and relations are implied by the ones in the original query and the constraints in the logical and physical schemas. Therefore, the intention of each one of these rewrites is to remove one binding $R\ y$ from the from clause of the query. Usually, such rewrites are applied until the minimal query is reached. A minimal query is then defined as the query Q such that there does not exist a subquery Q' with fewer bindings than Q while still being equivalent to Q.

The chase phase will result in the expansion of the original query into a universal query plan by adding the physical structures of the data, reason by which this phase is known as the "grow" phase, while on the other hand, the backchase phase will remove redundant terms from the query, making the query "shrink". If more than one Universal plan is found, apply the back chase to each one and then apply usual cost based optimization to keep the cheapest. It is precisely the "grow" phase of this algorithm we consider to be greatly similar with the expand phase for DL reasoning, as we will see later.

2.2 Description Logic

We provide below a summarized version of the important definitions of DL relevant to our research. For a more detailed presentation, please refer to the introductory chapters of [5]. *Description logics* (DL) is the denomination of a class of languages used in knowledge representation (KR) systems. The main strength and characteristic of these languages is that besides providing a way to store facts about the world under study (universe of discourse, or UD) they also provide reasoning services that allow the application to deduce implicit facts based on the knowledge represented explicitly in the system. Thus a knowledge representation system in DL allows the user to do three things: setting up a knowledge base (KB), manipulating it and reasoning about it. A KB is composed of two things, the TBox or terminology that contains the vocabulary, concepts (denoting just sets of individuals) and roles (denoting binary relations between individuals) of the application under consideration and the ABox containing assertions about specific individuals defined in terms of the vocabulary in the TBox. For example, a KB about the human resources in a company will store in the TBox things like "employees have a salary", "managers are employees who manage a department", and in the ABox things like "Adam is an employee" and "Adam manages the Engineering Department". Besides being able to handle basic concepts, DL also allow the user to build complex descriptions of concepts using a particular set of operators called "constructors". Every DL language is identified by the constructors it supports defining its expressive power and hence its reasoning complexity.

D (syntax)	*(comments)*
$\top$	Universal concept, also referred as Top
$\bot$	Bottom concept
C	Atomic concept
$\neg C$	Atomic negation
$D_1 \sqcap D_2$	General concept intersection
$\forall R.C$	Value restriction
$\exists R.\top$	Limited existential quantification

Table 2.1: DL Concept Constructors

The most basic descriptions are atomic concepts and atomic roles and more complex descriptions can be built using concept constructors following a predetermined grammar. For example, we can see in Table 2.1 the grammar for $\mathcal{AL}$, one of the minimal DL languages of interest. There, C denotes atomic concepts, i.e. those that can not be defined in terms of another concept, D, D_1 and D_2 denote general concepts, and R denotes atomic roles. The semantics in a DL language is generally defined using interpretations. An *interpretation* $\mathcal{I}$ consists of a non-empty set $\Delta^{\mathcal{I}}$ and an interpretation function $(\cdot)^{\mathcal{I}}$ that maps atomic concepts to subsets of $\Delta^{\mathcal{I}}$ and atomic roles to subsets of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is then extended to general concept descriptions in a natural way as presented in Table 2.2:

We can now define the terminologies for the KB using axioms. There are two types of axioms, *equalities* represented as $C \equiv D$ and *inclusions* represented as $C \sqsubseteq D$. An interpretation $\mathcal{I}$ satisfies an axiom if:

$(C \equiv D)^{\mathcal{I}} \rightarrow C^{\mathcal{I}} = D^{\mathcal{I}}$

$(C \sqsubseteq D)^{\mathcal{I}} \rightarrow C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$

In a terminology, equality axioms where the left hand side is a single concept are called definitions and respectively, inclusion axioms are called specializations. An interpretation that satisfies all the axioms in the terminology is called a *model*.

D	$(\cdot)^{\mathcal{I}}$
$\top$	$\Delta^{\mathcal{I}}$
$\bot$	$\emptyset$
C	$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
$D_1 \sqcap D_2$	$D_1^{\mathcal{I}} \cap D_2^{\mathcal{I}}$
R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
$\forall R.C$	$\{a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$
$\exists R.\top$	$\{a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}}\}$

Table 2.2: DL Concepts and their interpretation

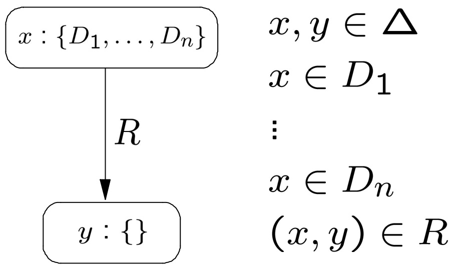

Figure 2.1: Example of a DL interpretation and its graph representation

As mentioned before, DL systems also support reasoning services about the knowledge base contained. The typical reasoning services for a TBox or terminology are to determine the satisfiability of a description, and to determine whether a description is more specific than another. The former refers to verifing whether or not a description is contradictory, while the latter implies deciding whether one description is subsumed by ($\sqsubseteq$) another. The latter service is also known as classification of concepts. Formally, satisfiability is defined as follows:

C is *satisfiable* w.r.t $\mathcal{T}$ if there exists an interpretation $\mathcal{I}$ model of $\mathcal{T}$ where $C^{\mathcal{I}} \neq \emptyset$

This essentially tells us that if we construct an interpretation $\mathcal{I}$ that satisfying all the axioms in $\mathcal{T}$ results in C being the empty set then the concept C is not satisfiable because we were not able to find an individual in Δ that satisfying $\mathcal{T}$ also satisfies C.

Even though we recognize two different reasoning services, satisfiability and subsumption, it is also known that in general[4], a satisfiability checking problem can be reduced to a subsumption problem and vice versa using a series of known equivalences. For instance to reduce subsumption to (un)satisfiability we use:

$$C \sqsubseteq D \iff C \sqcap \neg D \text{ is unsatisfiable}$$

Then using those equivalences we can just reduce the subsumption problems to satisfiability checking problems and answer all of them using a single DL reasoner providing a satisfiability checking service.

Traditionally, satisfiability checking in the implemented DL reasoners is done using tableau algorithms. In essence, the algorithm constructs a set of tableaux, starting with one node x_0 containing just a single description C_0 and then apply some consistency-preserving transformation rules until no more rules apply. If the tableau constructed this way does not contain a contradiction (called clash), then C_0 is satisfiable and unsatisfiable otherwise. The set of tableaux is obtained when dealing with the disjunction of concepts that requires to try finitely many tableaux and if at least one of them is satisfiable then the C_0 is satisfiable.

The usage of DL directly in query optimization has been tried in the past in three different ways. In the first one, queries are classified with respect to schema concepts in order to find the most specific concept that satisfy the full query. This means usually saving time since more specific concepts will have less individuals than the general ones. In the second approach, the query is actually expressed partially in DL and the rest in some other first order logic notation. The part that is expressed in DL is classified and then the rest of the query is tested against the most specific concept found in the classification process. Finally in the third approach, the DL is used to remove redundant terms similarly to the grow-shrink approach. First the query is expanded using the information in the schema and then all the query subterms that subsume the rest of the query (i.e. the redundant ones) are removed.

2.3 DL Reasoners

There are quite a few DL reasoners publicly available but since our interest is to see the similarities between CQO and DL reasoning we would have to modify the core functionality of one of the existing reasoners to achieve what we wanted. Since this

[4]It might not be possible to apply some of the equivalences if some of the constructors (like general negation of concepts $\neg$) are not available.

task was not practical we implemented our own reasoner based in the language of our choice that best suited our interests. However, we present below three of the most common reasoners to give the reader an idea of what can be easily attainable.

FaCT: Fast Classification of Terminologies. The FaCT reasoner is the result of research led by Ian Horrocks geared towards an optimized tableaux implementation of the DL reasoning algorithms. According to its website[5], FaCT is a DL reasoner used for concept classification and satisfiability testing. It actually includes two reasoners, one for the DL dialect called $\mathcal{SHF}$ ($\mathcal{ALC}$ augmented with transitive roles, functional roles and role hierarchies), and the other for the $\mathcal{SHIQ}$ dialect ($\mathcal{SHF}$ augmented with inverse roles and qualified number restrictions). Both reasoners use sound and complete tableaux algorithms. The main features of the FaCT reasoner are:

- A highly expressive DL, supporting reasoning even for database schemas.
- Support for reasoning with arbitrary knowledge bases (i.e. containing general concept inclusion dependencies)
- An optimized implementation for the tableaux algorithms (now the *de facto* standard for DL systems)
- A client-server implementation based on the CORBA architecture.

A newer version of the reasoner called FaCT++ is available as an implementation of an OWL-Lite reasoner in C++ including new optimizations and some new features.

RACER: Renamed A-Box and Concept Expression Reasoner[6]. RACER DL reasoner for the DL language $\mathcal{SHIQ}$. It supports reasoning with multiple TBoxes and A-Boxes (while FaCT only reasons with one TBox and no A-Box). Besides the basic features, it also supports reasoning with concrete domains that allow it to deal with integer number restrictions, linear polynomial equations over the reals, equalities and inequalities of strings, etc. Like FaCT, it also supports reasoning with knowledge bases containing general concept inclusion dependencies. Its main reasoning services are:

- Concept consistency with respect to a Terminology
- Concept subsumption with respect to a Terminology

[5]http://www.cs.man.ac.uk/~horrocks/FaCT/
[6]http://www.racer-systems.com/

- Concept classification with respect to a Terminology
- Consistency of an Interpretation with respect to a Terminology, and a few others.

Several other reasoners have appeared and disappeared throughout time but these two are the ones probably considered to be the most predominant.

Chapter 3

Architecture and Design

3.1 Overview

To explore in depth the overlap between CQO and DL subsumption reasoning we designed and implemented a subsystem containing the functions that were common to both approaches. Clearly, this subsystem could be called and shared between two (or more) applications built on top. It was evident that the main goal of this subsystem should be to create and manipulate a graph and to apply a defined set of rules over that graph representation. We thus named this subsystem the Common Graph Utility, or CGU (pronounced *see-gu*) for short. Figure 3.1 shows the main components of the architecture of our design including the supporting layers with the drivers built on top.

Memory Manager. This is the lowest layer. Its main function is to provide unified memory cells to the upper layers of the system. It is responsible for keeping the list of free cells. It requests memory on demand to the host operating system and places the released (not used anymore) cells back to the free list so memory can be reused. Its functions are only accessible to the Naive Lisp module.

Naive Lisp. This module handles the basic expressions similarly to most Lisp implementations. Its basic data elements are “atoms” and “lists” and the typical Lisp functions to create and manipulate symbolic expressions (S-expressions) like `read, car, cdr, cons, rplaca and rplacd`. This is the basic underlying component for the rest of the modules in the CGU.

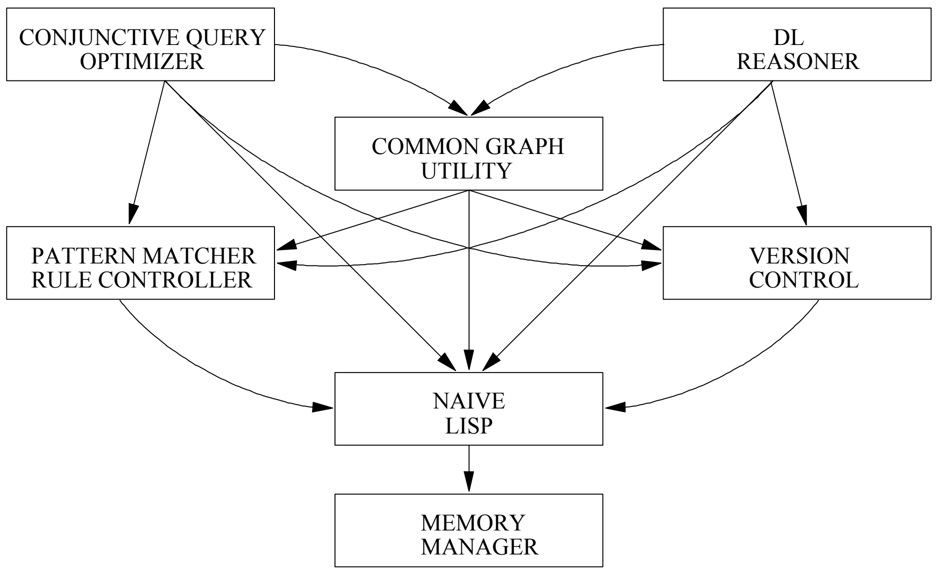

Figure 3.1: Modules and their interactions

Version Control. In order to have the ability of "remembering" a particular state of the graph, so changes can be applied and then undone, we created the version control functions. These functions basically duplicate on demand the pieces of memory that are modified and keep a log of which memory cells changed so they can be rolled back to a previous state when desired. The module's basic operations are pushVer, and popVer and the re-implementation of the lower level pointer navigation functions for S-expressions (`car, cdr, rplaca and rplacd`) in their "versioned" flavor.

Pattern Matcher/Rule Controller. These two utilities implement a basic yet powerful rule controller based on S-expressions. Its input is a rule-based control strategy built recursively using a particular set of constructs (e.g. call, repeat, sequence, if, etc.). The rules themselves fire when a particular pattern matches the input triggering an established set of actions that usually end with the rewrite of the original pattern matched to a new expression.

CGU. The core set of functions of our research. It is composed of two parts: (1) the graph manipulation functions and (2) the expand functions. The former provides an interface to create graphs (also called partial databases), add axioms to the terminology, create nodes, add descriptions to nodes, add edges, and then checking for the existence of particular nodes, descriptions and edges. The latter are the tableau algorithms modified to meet the requirements of both DL reasoning and CQO. There are two expand functions, the first one contains the deterministic part of the algorithm and the second one implements the non-deterministic part (disjunction handling).

Drivers:

DL Reasoner Module. The reasoner module behaves as a typical DL reasoner using a tell and ask interface where first a terminology is entered and then the typical question "$subsumes(D_2, D_1)$" is answered. In our case, this is done by translating it into a satisfaction check of the description $(D_1 \sqcap \neg D_2)$ transformed into negation normal form. The algorithm will then check for the satisfaction of that concept and will return the appropriate value.

CQO Module. This is the module for a conjunctive query optimizer that will use the CGU component to perform the optimization. It is a very basic algorithm where given a database schema and a query it will return a plan that implements that query for a database that satisfies the given schema. In simple terms, the query, given in a language provided in section 3.2.6, is transformed into a DL description and then added to a node in a Partial Database. This initial single node graph will be "expanded" with the schema previously inputted (abstracted as a terminology). Then using a second graph to generate the plan and a series of additional expand iterations, it will incrementally create a query plan based on the expanded query graph, choosing the appropriate access paths. This simple approach can be refined in multiple ways and we sketched some of those in the future research section.

Each one of these modules has different variables and functions that serve as their interface to the other layers. We briefly present below the main functions with a brief explanation of their usage. This does not intend to be an extensive documentation of the modules, but just a display of the key elements considered for our design and particular implementation.

3.2 Interfaces

3.2.1 Naive List Processor

This module defines the basic elements for the upper layers. In essence, it is an implementation of a very basic Lisp on C. We decided to create this basic layer, as we believe this allow us to use a powerful, yet quick and simple prototyping language as Lisp but without dealing with licensing issues or particularities of the different implementations. It supports two classical data types:

- boolean: Truth value, it can be either TRUE, or FALSE.
- string: An array of characters.

And like most Lisps, it works with the following built in types:

- atom: Similar to atoms in Lisp, an atom is an indivisible unit. It can be either a fixed length integer or a symbol which corresponds to a variable length string with no spaces. Integer atoms require only a single cell of memory, while atoms for variable length strings are encoded as one cell for the atom information plus a linked list of cells (called stringstore cells) each containing a fixed portion of the string. Atoms are unique within the system and have a definition pointer so they can also be used as variable names.
- lists: Again, similar to lists in Lisp, our lists are single linked lists where each cell of the list (called a *cons*) is composed of two pointers, the car and the cdr. The car points to the data item (either an atom or another list) and the cdr points to the next element of the list.
- nil: Null element, it can be either an atom or a list.

With those elements, we can form S-expressions recursively as follows[1]:

$\langle Sexpr \rangle$	*(comments)*
nil	null element
$\langle atom \rangle$	atom
$(\langle Sexpr \rangle \cdot \langle Sexpr \rangle)$	cons cell (list)

Table 3.1: S-expressions grammar

Using the above grammar we can create S-expressions like 10, a, $(nil \cdot nil)$, $(10 \cdot (20 \cdot (a \cdot nil)))$. These expressions are commonly represented using the diagrammatic convention shown in Figure 3.2. There, we can see that an atom is depicted as

[1]See the seminal paper from John McCarthy [16] for more information about symbolic expressions.

an item containing a definition pointer and the value of the atom while lists are depicted as an item that has two pointers. The nil element is usually encoded as a forward diagonal. Commonly, arbitrary length lists are written as $(e_1 e_2 e_3 \ldots e_n)$ to represent the S-expression $(e_1 \cdot (e_2 \cdot (e_3 \cdot (\ldots (e_n \cdot nil) \ldots))))$ like in the last example in the same figure.

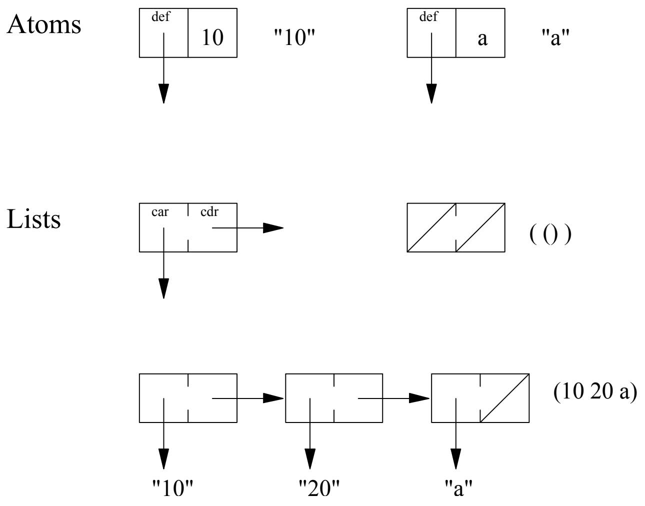

Figure 3.2: Examples of Atoms and Lists in our implementation

Consequently, the functions contained in the NLisp are the ones needed to create S-expressions from strings and to manipulate the S-expressions. The main functions are:

`lread`: *string* $\rightarrow$ *Sexpr*: Parses the string and converts it to the correspondent S-expression returning it in Sexpr.

`cons`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Creates a cons cell, points its car to $Sexpr_1$ and its cdr to $Sexpr_2$ and returns the newly created cons cell.

`car`: $Sexpr_1 \rightarrow Sexpr$: Only defined for lists, returns the *Sexpr* pointed by the car pointer of the cons passed in $Sexpr_1$.

`cdr`: $Sexpr_1 \rightarrow Sexpr$: Only defined for lists, returns the *Sexpr* pointed by the cdr pointer of the cons passed in $Sexpr_1$.

`rplaca`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Replaces the car of the cons passed in $Sexpr_1$ with $Sexpr_2$ and returns the modified cons cell.

`rplacd`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Similar to rplaca, it replaces the cdr of the cons passed in $Sexpr_1$ with $Sexpr_2$ and returns the modified cons cell.

`atom`: $Sexpr \rightarrow boolean$: Returns TRUE if the *Sexpr* is an atom or FALSE otherwise.

`null`: $Sexpr \rightarrow boolean$: Returns TRUE if the *Sexpr* is nil (the null element) or FALSE otherwise.

`set`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Assigns the definition pointer of the atom passed in $Sexpr_1$ to $Sexpr_2$ and returns the modified atom.

`eval`: $Sexpr_1 \rightarrow Sexpr$: If $Sexpr_1$ is an atom, returns the value of the definition pointer, otherwise, it assumes that it is a list where the first element is a function name, therefore it calls that function with the `cdr` of $Sexpr_1$ and returns the result in *Sexpr*.

`eq`: $Sexpr_1, Sexpr_2 \rightarrow boolean$: Equality test of S-expressions. When both are atoms and both are the exact same atom it returns TRUE. If both are lists then it performs a structural equality check and returns TRUE if both lists are structurally equal. In all other cases, it returns FALSE.

This module also contains the functions to create and update property lists for atoms. Property lists are used when an atom (the *symbol*) should be used as a variable and it requires to hold more than one *value* each of which is uniquely identified with an *indicator atom* typically reflecting the name of a property. Thus, a property list, is list or pairs where every pair is of the form (identifier value). The functions available to manipulate property lists are:

`putprop`: $symbol\_atom, value\_Sexpr, indicator\_atom$: Adds the property identified by *indicator_atom* to the property list of the *symbol_atom*. The value of the property is passed in *value_Sexpr*.

`remprop`: $symbol\_atom, indicator\_atom$: It removes the property identified by *indicator_atom* from the property list of the *symbol_atom*.

`getprop`: $symbol\_atom, indicator\_atom \rightarrow Sexpr$: It returns the *Sexpr* value of for the property identified by the *indicator_atom* in the property list for the *symbol_atom*.

For example, the list `(color red maker chevrolet)` reflects that for the property color the value is red, for the property model the value is astra and for the property age the value is 3. Such list could be created by the following calls:

`putprop(myvehicle,red,color)`, `putprop(myvehicle,chevrolet,maker)`

Then the call to `getprop(myvehicle,color)` would return `red` and the call to `remprop(myvehicle,color)` would remove it. This set of functions will prove very useful in the CGU module since all the description lists for the nodes are stored as values in a property list for the node name.

Additionally, the NLisp also contains functions to traverse lists, append them and do some other elaborated tasks but they are basically created using calls to the previous functions. In summary, it behaves as most of the other Lisp implementations creating and manipulating S-expressions providing the powerful approach that Lisp offers.

3.2.2 Version Control

The version control module provides the functions that allow the upper layers to consider the cons cell as a more powerful entity. Such cons cell must keep track of the different values of its car and cdr pointers throughout different points in time (identified by a version number). This capability is important for backtracking purposes (going back to a previous version). Note than only cons cells are versioned and atoms are not. However, the definition pointers of atoms can be assigned to a list first (like when using a property list) and then their values will be versioned after that. Figure 3.3 shows a single cons cell pointing to different S-expressions at three different versions. In the initial version 0, the car of the cons cell is pointing to the atom *a* and the cdr is pointing to *nil*. In version 1, the car is still pointing to *a* but the cdr is now pointing to *b*. In version 2, the car now points to *c* while the cdr still points to *b*. If we can remember the previous values (by storing them somehow) we can then for example go back to version 1 and know that the car points to *a* and the cdr points to *b*.

The version control functions allow to create these versioned cons cells and maintain them as the version number is increased and their car and cdr values are changed. The version control module defines two variables only accessible within the module:

`currentVer`: Integer representing the current version number. Its initial value is 0.

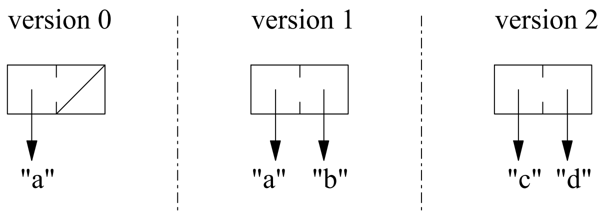

Figure 3.3: State of one cons troughout three different versions

`popLog`: List pointing to all the cons cells that changed in each one of the versions, from the initial version to the current version.

Using those two variables, it then defines the following functions:

`getcurrentVer`:$\rightarrow$ *integer*: Returns the Integer value of the current version number.

`resetLogVer`: It will empty the popLog and will set the current version number to 0.

`pushVer`: It increments the current version number by 1 and adds one entry to the popLog to hold all the cons cells modified in the new version. The name of this function derives from the idea that a new version will basically leave the previous values of every cons cell untouched. This is equivalent to "push" the current state of the system in a stack so that in can be recovered later.

`popVer`: It will decrement the current version v_i by 1 to v_{i-1} and it will traverse the popLog such that the values for all the cons cells in version v_i are removed and the previous values in version v_{i-1} are restored.

It then reimplements the pointer navigation and modification functions for cons cells.

`vcons`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Creates a cons cell in the current version, points its car to $Sexpr_1$ and its cdr to $Sexpr_2$ and returns the newly created cons cell.

`vcar`: $Sexpr_1 \rightarrow Sexpr$: Returns the *Sexpr* pointed by the car pointer in the current version of the cons passed in $Sexpr_1$.

`vcdr`: $Sexpr_1 \rightarrow Sexpr$: Returns the *Sexpr* pointed by the cdr pointer in the current version of the cons passed in $Sexpr_1$.

`vrplaca`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Replaces the car of the current version for the cons cell passed in $Sexpr_1$ with $Sexpr_2$ and returns the modified cons cell.

`vrplacd`: $Sexpr_1, Sexpr_2 \rightarrow Sexpr$: Similar to rplaca, it replaces the cdr of the current version for the cons cell passed in $Sexpr_1$ with $Sexpr_2$ and returns the modified cons cell.

The cons cells can be explicitly versioned if we use the set of functions with the prefix "v" (`vcons`, `vcar`, `vcdr`, `vrplaca`, `vrplacd`) or they can be implicitly versioned if using the default function (without the prefix) but with the version control module configured to overwrite them.

3.2.3 Pattern Matcher/Rule Controller

The *pattern matcher* is used to find whether an S-expression matches a specified pattern given in the input. The pattern grammar contains several symbols that allow the user to indicate how to match the S-expression. It also allows to specify variables for binding specific parts of the S-expression matched. The main function is:

`Match`: *pattern*, *Sexpression* $\rightarrow$ *boolean*: Returns TRUE if the *Sexpression* matches the *pattern* and FALSE otherwise. For example, given the pattern:

`(plus` > `operand1` > `operand2)` and the input: `(plus 10 20)`

the Match function will return TRUE, since the atom "`plus`" matches literally and the `operand1` is matched with `10` and `operand2` is matched with `20`. When it finishes matching, the atom `operand1` holds the value `10` and the `operand2` holds the value `20`. For more complex patterns and S-expressions, the search is performed in a left-to-right top-to-bottom direction on the pattern. Besides the ">", several other symbols can be used in the pattern to perform certain actions as described in Table 3.2.

In order to allow recursive usage of the patterns, the binding values in the variables can be associated with an identifier in every iteration. A stack of matched variables' identifiers is maintained and the values are assigned to the variables using a pair of accessor functions that ties the value with the current iteration identifier. Thus, the functions performing that logic are:

(symbol)	*(action)*
!	matches the next atom literally
?	matches one element
$+$	matches one or more elements
$*$	matches zero or more elements
$>\ V$	binds V to the next element
$<\ V$	matches iff the next element matches the current binding of V
$> + V$	binds V to a list of elements (one or more) matched by $+$
$> * V$	binds V to a list of elements (zero or more) matched by $*$
$<<\ V$	the current value of V is appended to the pattern
$<>$ $(P1)$ $(P2)$	matches $P1$ $P2$ in either order
or $EList$	matches if the next atom matches any member of $EList$
$>$ *or* $EList$ V	same as above but binds the atom matched to V
where C	succeeds if the S-expression C evaluates to non-null

Table 3.2: Special symbols used in the pattern input

`PushMatchVar`: It pushes the current identifier into the stack and gets a new identifier for the next iteration.

`PopMatchVar`: It pops out the identifier at the head of the stack and makes it available for the next iteration.

`GetBindVal`: *var* $\rightarrow$ *Sexpr*: It returns the S-expression stored as the value of *var* using the current match iteration identifier.

`PutBindVal`: *var*, *val*: It binds the value *val* to the variable *var* using the identifier of the current match iteration.

Finally, the pattern matcher module also provides a function to build an S-expression using the bindings of the variables previously assigned in a call to `Match`. This is useful especially in the next module, the Rule Controller, where matched patterns are rewritten using the values used in a previous call to `Match`. The function:

`Build`: *pattern* $\rightarrow$ *Sexpr*: Returns the S-expression built from the pattern using the bindings created by `Match`. It allows usage of some special symbols in the pattern as presented in Table 3.3:

The *rule controller* has one main function whose job is to apply a given rule control strategy to the S-expression passed as input. The signature of the function is:

(symbol)	*(action)*
$< V$	produces the binding of V
$<< V$	the list-valued binding of V is spliced into the pattern
$! A$	produces the value A

Table 3.3: Special symbols used in the Build function

`ApplyRuleControl`: *strategy*, *Sexpr* $\rightarrow$ *boolean.* The function returns TRUE if the application of the strategy was successful or FALSE otherwise. The strategy can be specified using the following constructs:

$\langle RuleName \rangle$ –The rule called $\langle RuleName \rangle$ is applied. The general syntax for a rule is $(\langle RuleName \rangle\, \langle LeftHandSide \rangle\, \langle RightHandSide \rangle\, \langle form_1 \rangle \ldots)$. If the given S-expression matches the $\langle LeftHandSide \rangle$, then (1) each of the $\langle form_i \rangle$ are built, (2) each of the built forms are evaluated and $\langle RightHandSide \rangle$ is built replacing the given S-expression.

`(call` $\langle StrategyName \rangle$`)`: The control strategy with the given name is applied on the argument S-expression. Returns the result of applying $\langle StrategyName \rangle$.

`(not` $\langle Strategy \rangle$`)`: $\langle Strategy \rangle$ is applied. Returns logical negation of the result.

`(or` $\langle Strategy \rangle$`...)`: Each $\langle Strategy \rangle$ is applied in sequence until the first that successfully applies a rule. Returns logical disjunction of results.

`(and` $\langle Strategy \rangle$`...)`: Each $\langle Strategy \rangle$ is applied in sequence until the first that fails to apply. Returns logical conjunction of results.

`(seq` $\langle Strategy \rangle$`...)`: Each $\langle Strategy \rangle$ is applied in sequence. Returns TRUE if any of the $\langle Strategy \rangle$s is applied.

`(rep` $\langle Strategy \rangle$`...)`: The argument $\langle Strategy \rangle$ is repetitively applied until it no longer returns TRUE. Returns FALSE if non of the $\langle Strategy \rangle$'s are applied and TRUE otherwise.

`(if` $\langle Pattern \rangle$ $\langle Strategy \rangle$`)`:If $\langle Pattern \rangle$ matches the given S-expression $\langle Strategy \rangle$ is applied and TRUE is returned.

`(env c[ad]`$^+$`r|eval` $\langle Strategy \rangle$`)`: $\langle Strategy \rangle$ is applied on the $c[ad]^+r$ of the given S-expression or even in the current S-expression (eval). Returns the result of applying $\langle Strategy \rangle$.

`(map` $\langle Strategy \rangle$`)`: $\langle Strategy \rangle$ is applied on each element of the given S-expression. Returns TRUE only if $\langle Strategy \rangle$ is applied to each of the elements.

These two modules are used by the CGU and the two drivers to perform different tasks, like the transformation of descriptions into Negation Normal Form (NNF) and the renaming of variables to enforce uniqueness. They are also used by the expand functions to recognize each one of the different concept constructors and to perform the appropriate transformations on the graph. For example consider the following rules for the NNF program:

```
((DoubleNeg (not (not > P)) (<< P))
(SimpleNeg (not (atomic > P)) (not(atomic < P)))
(Atomic  (atomic > P) (atomic < P))
```

The first rule deals with double negation of descriptions and removes the two negations. The second and third rules just recognize atomic concepts and negation of atomic concepts as being already in negation normal form. Then, with those rules, consider the following fragment of the strategy control for transforming descriptions into NNF:

```
(negNForm
  (or (or Atomic SimpleNeg)
      (if (not (not > P))
          (and DoubleNeg (env eval (call negNForm))))))
```

Now suppose that we call

`ApplyRuleControl(negNForm,(not(not(atomic EMP))))`

The call above will apply the control `negNForm` whose main strategy is a disjunction of two things. The first substrategy `(or Atomic SimpleNeg)` captures that if the formula is already an atomic concept or the negation of an atomic concept, then it is already in NNF. Thus, it will try to apply the rule `Atomic` to `(not(not(atomic EMP)))` that in turn will try to match `(atomic > P)` with `(not(not(atomic EMP)))`. The match will fail, so the rule controler will go to the next rule `SimpleNeg` failing as well. Since the first substrategy failed, it will continue to the next one (and it would continue until the first one that succeeds). Then second substrategy is applied by first trying to match `(not (not > P))` with `(not (not (atomic EMP)))`. The match will succeed and as a result `(atomic EMP)` will be stored as the value of the variable `P`. Also, since the condition of the "if" was true, the `DoubleNeg` rule is applied. This will match again `(not (not > P))` with `(not (not (atomic EMP)))` and as a result it will replace `(not (not (atomic EMP)))` simply with `(atomic EMP)`. Then it will recursivelly apply the `negNForm` on `(atomic EMP)`and it will succeed in the first application of the `Atomic` rule. This application of the control will return TRUE consequently the whole call will return TRUE and the S-expression `(atomic EMP)` that is, as one might expect, the transformation of `(not(not(atomic EMP)))` into NNF.

3.2.4 Common Graph Utility (CGU)

At the core of our implementation modules we find the Common Graph Utility. The CGU is the module containing the functions to manipulate the graphs that will represent queries or plans for the CQO driver and tableaux in the DL Reasoner. Traditionally, in the DL-Database community, a DL interpretation $\mathcal{I}$ can be considered also a database instance. If we recall from section 2.2, an interpretation is defined via a set of individuals in a domain and an interpretation function that maps concepts in DL to subsets of the domain and roles in DL to binary relations between elements of the domain. Then an interpretation or database, can be represented using a graph in which nodes represent individuals of the domain, individuals belonging to a subset are identified with a label tied to the individual, and labeled edges represent binary relations between individuals. Since during the reasoning process we only represent the information that we are certain to know, the database is not always complete, so we say that we have a partial database. Thus, all the functions that exist in the CGU are operations to be performed against one (or more) of these partial databases.

The CGU supports handling one or more partial databases and a global terminology shared among all of them. A partial database PD within our system, is uniquely identified by a *PDname*. It has a local terminology (additional to the global one shared across the databases) plus a list of nodes. Every node is identified by a *nodeName* and contains a set of labels ($\mathcal{L}(nodeName)$) that encode the following elements present in the graph related to that particular node:

- DL descriptions: A description can be any properly constructed description in $\mathcal{DLFDA}$. See subsection 3.2.5 for the complete list of $\mathcal{DLFDA}$ constructors and their corresponding S-expressions used.
- Edges: The presence of an edge labeled `A` from the current node to node `n1` is encoded as `(edge A n1)`.
- Inequality arcs: If two nodes are explicitly different, then there exist an inequality arc between them. We encode that `n1` is different from `n2` by placing

`(ineq n2)` in the list of labels for `n1` ($\mathcal{L}(n1)$) and `(ineq n1)` in the list of labels for n2 ($\mathcal{L}(n2)$).

- Equality between nodes: We encode the fact that several nodes can be equal by creating a tree rooted in an arbitrary node in the set. All those nodes will belong to the same equivalence class. Equivalent classes are identified by the name of the root node of the tree. Then all the subtrees contain nodes that are equal to the root and each one of those nodes has one tag indicating its parent node. The S-expression used for this is `(eq n)` where `n` is the name of the parent node.

- Event triggers: These are special tags encoding events that will be triggered when additional concepts are added to the node or when additional edges are added.

So in order to manipulate Partial Databases with the characteristics described above, we have the following functions available:

`createPD`: $PDName$: This function creates the "context" space to store the elements of a partial database and labels it $PDName$. The context space basically requires to hold the name $PDName$, two stacks of description-node pairs and the list of nodes.

`deletePD`: $PDName$: Deletes all the contents of the partial database $PDName$ and removes its context space from the system.

`setPD`: $PDName$: The system uses an indicator for the "current" partial database used by the rest of the functions below to execute their actions to the current PD. This function then sets the system variable `currentPD` to the PD identified by $PDName$.

`addGlobalIncDep`: LHS, RHS: It adds the inclusion dependency $LHS \sqsubseteq RHS$[2] to the global terminology (available to all the partial databases).

`addLocalIncDep`: LHS, RHS: Adds the inclusion dependency $LHS \sqsubseteq RHS$ to the terminology of the current PD.

`createNode`: $nodeName$: Adds the node $nodeName$ to the current partial database.

`addDescription`: $nodeName, D$: Adds the description D to the node $nodeName$.

[2]LHS=Left Hand Side. RHS=Right Hand Side

`addEdge`: $nodeName1, nodeName2, label$: Adds an edge from $nodeName1$ to $nodeName2$ labeled $label$.

`addEquality`: $nodeName1, nodeName2$: It makes the equivalence classes of the two nodes the same and it will merge their descriptions. At the beginning, all the nodes are their own equivalence class

`addInequality`: $nodeName1, nodeName2$: It adds a label to $nodeName1$ to indicate that it is different from $nodeName2$ and it will add the correspondent label to $nodeName2$ indicating that it is different from $nodeName1$.

`pushStack1`: $Description, NodeName$: It adds the duple $Description-NodeName$ to the top of the stack1.

`pushStack2`: $Description, NodeName$: It adds the duple $Description-NodeName$ to the top of the stack2.

`expand1`: Applies the deterministic rules to the duples in the stack1 until it is empty or a clash if found.

`expand2`: Applies the non-deterministic rules to the duples in the stack2 until it is empty or a clash if found.

Additional functions also exist to: traverse all the nodes in the PD; for every node, traverse the list of descriptions, test for a particular description existing in the list and deleting a particular description from the list. With these functions we can basically traverse and manipulate the graph as required for the drivers using the CGU.

3.2.5 DL Reasoner

The DL reasoner driver implements a complete subsumption reasoning procedure for the question:

$$\mathcal{T} \models (D_1 \sqsubseteq D_2)$$

The DL dialect chosen, called $\mathcal{DLFDA}$, is a variation of CLASSIC where we allow general negation of concepts (and therefore, disjunction of descriptions), do not handle roles but handle attributes and inverse attributes. The reason to just handle attributes (also called *features* in traditional DL literature) is because typical databases use attributes much more often than general roles and because attributes can be considered a special case of roles with at most 1 and at least 1 number restrictions.

Definition 1 *Let C denote primitive concepts and A primitive attributes. Derived descriptions for the Description logic $\mathcal{DLFDA}$ is defined by the grammar presented in Table 3.4:*

D(concrete syntax)	*(comments)*
`(top)`	everything
`(bottom)`	nothing
`(atomic C)`	primitive concept
`(not (atomic C))`	concept negation
`(and D1 D2)`	concept conjunction
`(equal (Pf1) (Pf2))`	path agreement
`(nequal (Pf1) (Pf2))`	path disagreement
`(at D A)`	inverse attributes
`(pfd C (A1...An) ())`	keys
`(or (forall Pf (not (atomic C))) D)`	rule-based disjunction
`(forall A D)`	attribute typing
`(or D1 D2)`	concept disjunction

Pf(concrete syntax)	*(comments)*
`()`	identity function
`(A PF)`	path composition

Table 3.4: $\mathcal{DLFDA}$ CONSTRUCTORS

The first column of Table 3.4 shows the constructors with the concrete syntax we use in the CGU. However, traditional DL papers use an abstract syntax as presented in Table 3.5 just for reference. Since our CGU uses the S-expressions to handle all the constructors, from this point forward, we try to refer to the DL constructors using their correspondent S-expression unless the abstract syntax allows to present the explanation in a clearer way.

A particular construct that we are adding as a special case is our rule-based disjunction. We include it as a special case because while a disjunction, this particular form is actually handled by the deterministic part of the algorithm as we will see in section 4.3. The name "rule-based" comes from the fact that it actually capture things that look like rules of the type $(\forall Pf.C) \rightarrow D$.

D(*abstract*)	D(*concrete syntax*)
$\top$	`(top)`
$\bot$	`(bottom)`
C	`(atomic C)`
$\neg C$	`(not (atomic C))`
$D_1 \sqcap D_2$	`(and D1 D2)`
$Pf_1 = Pf_2$	`(equal (Pf1) (Pf2))`
$Pf_1 \neq Pf_2$	`(nequal (Pf1) (Pf2))`
$D@A$	`(at D A)`
$C : \{A_1, \ldots, A_n\} \to id$	`(pfd C (A1...An) ())`
$(\forall Pf.\neg C) \sqcup D$	`(or (forall Pf (not (atomic C))) D)`
$\forall A.D$	`(forall A D)`
$D_1 \sqcup D_2$	`(or D1 D2)`
Pf(*abstract*)	Pf(*concrete syntax*)
id	`()`
$A.Pf$	`(A PF)`

Table 3.5: $\mathcal{DLFDA}$ DL ABSTRACT SYNTAX VS. CONCRETE SYNTAX

For the interpretation of the descriptions using the constructors above, we use the traditional approach of defining an interpretation $\mathcal{I}$ consisting of a domain $\Delta^{\mathcal{I}}$ and an interpretation function $(\cdot)^{\mathcal{I}}$. Note in particular that, while primitive concepts are considered subsets of the domain, attributes are considered total functions (as opposed to just binary relations like roles).

Definition 2 *The Semantics of the dialect $\mathcal{DLFDA}$ is given by $\mathcal{I} = \left(\Delta^{\mathcal{I}}, (\cdot)^{\mathcal{I}}\right)$ as follows (Table 3.6):*

In the reasoner, the concept of a knowledge base is captured by one partial database as the context where the descriptions occur. Descriptions are directly captured and the terminology is stored by means of the local inclusion dependencies.

The functions present in the DL Reasoner are mainly based in the "Proposal for a DL Interface" [17] where the authors suggest a DL interface based on transactions, with all the functions grouped as tell operations and ask operations. Our approach does not support transactions so all the functions assume a single user is running at any time. However, in the proposal they do not consider the existence of more than one knowledge base so we borrowed the prototype of the first function from

D	$(\cdot)^{\mathcal{I}}$
`(top)`	$\Delta^{\mathcal{I}}$
`(bottom)`	$\emptyset$
`(atomic C)`	$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
`(not (atomic C))`	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
`(and D1 D2)`	$D_1^{\mathcal{I}} \cap D_2^{\mathcal{I}}$
`(equal (Pf1) (Pf2))`	$\left\{a \in \Delta^{\mathcal{I}} \mid (pf_1)^{\mathcal{I}} a = (pf_2)^{\mathcal{I}} a\right\}$
`(nequal (Pf1) (Pf2))`	$\left\{a \in \Delta^{\mathcal{I}} \mid (pf_1)^{\mathcal{I}} a \neq (pf_2)^{\mathcal{I}} a\right\}$
`A`	$A^{\mathcal{I}} : \Delta^{\mathcal{I}} \to \Delta^{\mathcal{I}}$
`(at D A)`	$\left\{a \in \Delta^{\mathcal{I}} \mid \exists b. (A)^{\mathcal{I}} b = a \wedge b \in D^{\mathcal{I}}\right\}$
`(pfd C (A1...An) ())`	$\left\{\begin{array}{c} a \in \Delta^{\mathcal{I}} \mid \forall b \in C^{\mathcal{I}} \\ \left(\bigwedge (A_i)^{\mathcal{I}} a = (A_i)^{\mathcal{I}} b\right) \to a = b \end{array}\right\}$
`(or (forall (Pf) (not (atomic C))) D)`	$\left\{a \in \Delta^{\mathcal{I}} \mid \left((pf)^{\mathcal{I}} a \notin C^{\mathcal{I}}\right) \vee \left(a \in D^{\mathcal{I}}\right)\right\}$
`(forall A D)`	$\left\{a \in \Delta^{\mathcal{I}} \mid (A)^{\mathcal{I}} a \in D^{\mathcal{I}}\right\}$
`(or D1 D2)`	$D_1^{\mathcal{I}} \cup D_2^{\mathcal{I}}$
`()`	$\left\{(a,a) : a \in \Delta^{\mathcal{I}}\right\}$
`(A Pf)`	$\left\{(a,b) : (pf)^{\mathcal{I}} \left((A)^{\mathcal{I}} a\right) = b\right\}$

Table 3.6: $\mathcal{DLFDA}$ SEMANTICS

the RACER reasoner that does support several TBoxes. The functions present in our reasoner are:

`InKnowledgeBase`: *KBname*: Equivalent to a `setPD(`*KBname*`)`. If the Partial Database *KBname* does not exist already, it will create it first using `createPD(`*KBName*`)` and then execute the setPD call.

`implies_c`: $D1, D2$: This is equivalent to a call to `addLocalIncDep(`$D1$,$D2$`)` in the CGU

`equal_c` : $D1, D2$: Equivalent to two symmetric calls, one to `addLocalIncDep(`$D1$,$D2$`)` and another to `addLocalIncDep(`$D2$,$D1$`)`.

`satisfiable`: $D \to$ *boolean*: This is implemented by creating a node and depositing D. The PD is then expanded and if it finds a clash (expands return FALSE) then D is not satisfiable, if it returns TRUE then D is satisfiable.

`subsumes` : $D2$,$D1$ $\rightarrow$ *boolean*: As mentioned before, this is implemented by creating a node in the current partial database (set by a previous call to `InKnowlegeBase`) and depositing the Description ($D1 \sqcap \neg D2$) after being normalized using a negation normal form.

The negation normal form is reached by pushing negation inside, removing double negations and using the following transformations:

```
(not (and D1 D2))→(or (not D1)(not D2))
(not (or D1 D2))→(and (not D1)(not D2))
(not (equal (Pf1) (Pf2)))→(nequal (Pf1) (Pf2))
(not (nequal (Pf1) (Pf2)))→(equal (Pf1) (Pf2))
(not (forall (Pf) D))→(forall (Pf) (not D))
```

Once we have the partial database with that node and the normalized description we call the expand1 and if it succeeds then we call the expand2 function. If any of the expand functions returns FALSE, a clash has been detected meaning that the normalized description is unsatisfiable which traduces in indeed D1 subsumes D2. If, on the contrary, none of the functions find a clash, then they have just constructed a counter example of an individual that is a D1 but not a D2 contradicting the fact that D2 subsumes D1. Therefore, in the latter case the function subsumes(D2,D1) will return FALSE.

The functions *implies_c* and *equal_c* are the tell operations and the satisfiable and subsumes functions are the ask operations.

3.2.6 Conjunctive Query Optimizer

The query optimizer module implements a series of algorithms that allow the transformation of a conjunctive query (expressed in the grammar described below) into a DL description (using our DL dialect presented in the previous subsection) and then to a graph representing roughly a universal plan in the sense presented in [2]. Then working with a differential analysis approach along the lines the one presented in [18], generate a query plan to execute the query.

The interface to the CQO is composed of two operations in a "tell and ask" fashion, similar to the DL reasoner module. The tell operation essentially deals with inputting the database schema at the logical and physical levels and the ask operation generates the plan based on the query passed as an input. The prototype for the tell operation is:

`addtoSchema(`*SchemaConstraint*`)`: Add the *SchemaConstraint* to the database schema. The schema constraints can be any of the following (Table 3.7):

(schema constraints)	*(description)*
`(class C)`	Class Definition
`(property C A)`	Property Definition for class C
`(is-a C D)`	General Restriction. D is any $\mathcal{DLFDA}$ description
`(address C)`	Define C as an address to an object (general object)
`(string C)`	Define C as string (data typing)
`(integer C)`	Define C as integer (data typing)
`(index C n)`	Define C as index with n parameters.
`(param C Pi Ci)`	Define that parameter Pi of Index C is of type Ci

Table 3.7: Schema constraints used to define the database schema

The first three constraints are used to describe the schema at the logical level and the last five are used to describe the schema at the physical level. We will later present a more detailed usage of such constraints to define a sample schema in section 4.5.

The function provided for generating the plan (ask operation) is:

`compile(`Q`)`$\rightarrow$ P: Generate the plan P that implements the query Q on a database conforming to the database schema previously inputted in with a series of `addtoSchema` calls. Q has to be a valid conjunctive query as defined by the following grammar:

Definition 3 *Let $\{A_1, \ldots, A_n\}$ be query variables. The syntax for the query language used to define queries for the conjunctive query optimizer is :*

Our chosen query language was originally presented in [4], where the authors use this grammar as a way to characterize conjunctive queries and then to reason about duplicate elimination using DL. This is the grammar we will use through the rest of the thesis to express conjunctive queries. Its semantics is defined by extending our concept of database or interpretation $\mathcal{I}$ to also contain a mapping function $(\cdot)^{\mathcal{E}}$ from queries to bags of tuples as defined below:

Q	*(comments)*
`(as` C A`)`	variable naming
`(equal` A_1 `(`Pf_1`)` A_2 `(`Pf_2`))`	equality clause
`(select` $A_1 \ldots A_n Q$`)`	projection
`(from` $Q_1\, Q_2$`)`	natural join
`(elim` Q`)`	duplicate elimination

Table 3.8: Query language for the CQO

Definition 4 *Semantics of queries. Let $(\cdot)^{\mathcal{E}}$ be a mapping function from queries to bags, then the sematics of the query language is given by grammar in Table 3.9. Query variables occurring in a given query are assumed to satisfy standard conditions of wellformedness and are never reused within the query.*

Q	$(\cdot)^{\mathcal{E}}$
`(as` C A`)`	$\left\{\left\vert \langle A : v\rangle : v \in C^{\mathcal{I}}\right\vert\right\}$
`(equal` A_1 `(`Pf_1`)` A_2 `(`Pf_2`))`	$\left\{\left\vert \langle A_1 : v, A_2 : w\rangle : v, w \in \Delta, (Pf_1)^{\mathcal{I}} v = (Pf_2)^{\mathcal{I}} w\right\vert\right\}$
`(select` $A_1 \ldots A_n$ Q`)`	$\left\{\left\vert \langle A_1 : v@A_1, \ldots, A_n : v@A_n\rangle : v \in Q^{\mathcal{E}}\right\vert\right\}$
`(from` $Q_1\, Q_2$`)`	$\{Q_1 \bowtie Q_2\}$
`(elim` Q`)`	$\left\{v : v \in Q^{\mathcal{E}}\right\}$

Table 3.9: Query language semantics

For example, to express the query *Give me the name of the employees working in the computing department*[3] we will use the query described in Table 3.10:

The compile function generates the plan that will implement the query Q using the following plan language constructs:

Definition 5 *Let P be a query plan generated by the compile function. Then P is given by the language presented in Table 3.11:*

The semantics of the plan language can be given using the query language so that by transitivity it translates into bags of tuples. Thus, the semantics for the constructs in the plan language is then presented in Table 3.12:

The scan operation is equivalent to get all the individuals returned from the index C when the values $v_1, \ldots, v_n$ have been bound. All the individuals will be

[3]Assuming there are objects of type employee having properties ename and works_in and objects of type department having a property dname.

```
(select ename
  (from
    (as employee E)
    (from
      (equal ename () E (ename))
      (from
        (as department D)
        (from
          (as ‘computing’ P)
          (from
            (equal D (dname) P ())
            (equal E (works_in) D ())))))))
```

Table 3.10: Simple query to return name of employees

bindings for v. The load operation is an assignment from v_2 to v_1. The check operation is a condition to discard the value if the current bindings of v_2 and v_1 are not the same value. The nest operation is just equivalent to consider all the values in a nested loop join fashion and the keep operation is equivalent to a projection operation, i.e. keep only the binding for $v_1, \ldots, v_n$ and discard the rest. The last three operations in the plan language (Table 3.11) are just proposed here but they are beyond the scope of this thesis and will be considered in a future work.

P	*(comments)*
(scan $C\ v\ v_1, \ldots, v_n$)	index Scan
(load $v_1\ v_2$)	load into v_1 the value of v_2
(loadr $v_1\ v_2$)	traverse the R attribute from v_2 and load it into v_1
(check $v_1\ v_2$)	check whether v_1 has the same value of v_2
(checkr $v_1\ v_2$)	check whether v_1 has the same value of $v_2.R$
(param v)	declare v as a parameter and load it
(nest $P_1, \ldots, P_n$)	nest subplans P_1,...,P_n iteratively
(keep $v_1, \ldots, v_n\ P$)	keep bindings for v_1,...,v_n from plan P
(first P)	get the first tuple from P
(empty $v_1, \ldots, v_n$)	return the null tuple with schema $v_1, \ldots, v_n$
(cat $P_1\ P_2$)	union plans P_1 and P_2

Table 3.11: Plan Language

(plan)	*(semantics)*
(scan $C\ v\ v_1, \ldots, v_n$)	(from (as C v)
	(from (equal v (P_1) v_1 ())
	...
	(from (equal v (P_{n-1}) v_{n-1} ())
	(equal v (P_n) v_n ()))...))
(load $v_1\ v_2$)	(equal v_1 () v_2 ())
(loadr $v_1\ v_2$)	(equal v_1 () v_2 (R))
(check $v_1\ v_2$)	(equal v_1 () v_2 ())
(checkr $v_1\ v_2$)	(equal v_1 () v_2 (R))
(nest $P_1, \ldots, P_n$)	(from P_1
	...
	(from $P_{n-1}\ P_n$)...)
(keep $v_1, \ldots, v_n\ P$)	(select $A_1 \ldots A_n\ Q$)

Table 3.12: Plan language declarative semantics

Chapter 4

Algorithmic details

We will now present the algorithms and some implementation issues relevant to each of the important modules in the project. We will try to review the most relevant decisions made while using several examples that will present the functionality in a concrete way.

4.1 Naive Lisp

Our Lisp implementation was built using a data structure called a memblock. Each memblock could be of any of these three types (using a union in C):

1. Free cell: Used in the memory manager for keeping the list of available memory. It only has one field called next to create the circular linked list.
2. Stringstore cells: Used to store fragments of strings. These cells can hold up to a fixed amount of characters (currently 8) so we could store a string (the value of an atom) of any length just by using the appropriate amount of stringstore cells. For example, to store the string “intermitent” we will store “intermit” in one stringstore and “ent” in another. It has two fields:
   - strdata: An array of characters
   - next: pointer to the next strstore.
3. S cells: Used to represent numeric atoms, string atoms, cons cells and the null element nil.These cells have a type field (1=numeric atoms, 2=string atoms,

3=cons cell, 4=nil) and a reference count field that indicates the number of references to the given cell. For example, a reference count of 3 in the atom "A" indicates that there are 3 other objects (C variables, cons cells or atoms) pointing to the atom "A".

Atoms are unique in the system while cons cells are not. To guarantee uniqueness, the atoms are referenced in a hash table using the contents of the first string-store cell as the key. For numeric atoms, the value is used as the key.

Every time and S-cell is used, its reference count is incremented via a function called `assign`. Likewise, every time an S-cell is no longer used, its reference count is decremented via a function called `release`. When the reference count gets to 0 during a call to `release`, the memory is freed and returned to the list of free cells. Besides a manual call to `assign` usually because of C variables, the functions `cons` and `set` call `assign` on the second parameter because it is now referenced by the first parameter. Similarly, the functions `rplaca` and `rplacd` call `release` on the second parameter because it is no longer referenced by the first parameter.

4.2 Version Control

The version module implements every logically versioned cons cell as a list of cons cells, two for every version in which the logical cons cell has a different value. For this purpose it has to distinguish between the physical single cons cells (accessed by the functions `_cons`, `_car`, `_cdr`, `_rplaca`, and `_rplacd`) and the logical versioned cons cells. Intuitively, for a single versioned cons cell, the list of cons cells behaves like a stack where the element at the top contains the values for the current version and the elements below contain the values for the previous versions. In order to show the reader how the versioned cells work, consider the sequence of calls to the versioning functions presented in Figure 4.1:

Figure 4.2 shows the `x` cons cell as seen logically (above) and as seen physically (below) throughout the execution of the program in Figure 4.1. After the original creation of the cons cell (in line 1) we have that a single logical cons cell is implemented using two physical cons cells (see Figure4.2(i)). In other words, the call to vcons(a,b) is translated to two low level cons calls essentially in this way:

```
vcons(a,b) := _cons(_cons(a,b), nil)
```

Then in line two, we call `pushVer()` whose only effect is to increment the current version number leaving the cons cell unaffected. At this point, calls to `vcar` and

```
1: x=vcons(a,b)
2: pushVer()
3: vrplacd(x,d)
4: y=vcar(x)
5: pushVer()
6: vrplaca(x,c)
7: z=vcdr(x)
```

Figure 4.1: Sample usage of versioning functions

`vcdr` will still return the same values as before the call to `pushVer()`. Now, after the execution of line 3, the cons cell `x` looks like the Figure 4.2(ii). The call to `vrplacd` to replace the cdr value of the cell in a newer version forces the creation of two new cons cells "pushing" the new value to the top of the list. The pseudocode for the `vrplacd` operation is presented in algorithm 1:

Algorithm 1 pseudocode for vrplacd

```
Require: a valid cons cell $s$, the new value $v$ for the cdr
1: if version of the cell is lower than current version then
2:     t1:=_cons(_car(s),_cdr(s)) {Copy the head of the stack}
3:     _rplacd(s,t1) {Place the copy in the cdr of the head}
4:     t2:= _cons(_car(_car(s)),v){Create a new low level cons cell with the previous
       logical car value and the new cdr value}
5:     _rplaca(s,t2) {Point the head of the stack to the newly created cons cell}
6:     Add s to the poplog list of the current version
7: else
8:     _rplacd(_car(s),v) {Modify the cdr value of the cell at the car }
9: end if
```

Figure 4.3 shows the modification of the physical cons cells according to algorithm 1. Figure 4.3(i) shows the cell at line 1, Figure 4.3(ii) shows the cell after line 3 is executed and Figure 4.3(iii) shows the cell after line 4 is executed. Note that only when the version of the cell is smaller than the current version, the cell is added to the pop log and new physical cons cells are created. If the version is the same, only the values are modified but no cons cells are added to the list. A very similar algorithm is implemented for vrplaca.

Since we are always keeping the current version at the top and `vcar` and `vcdr` only require the values for the current version, it is clear that retrieving the vcar or

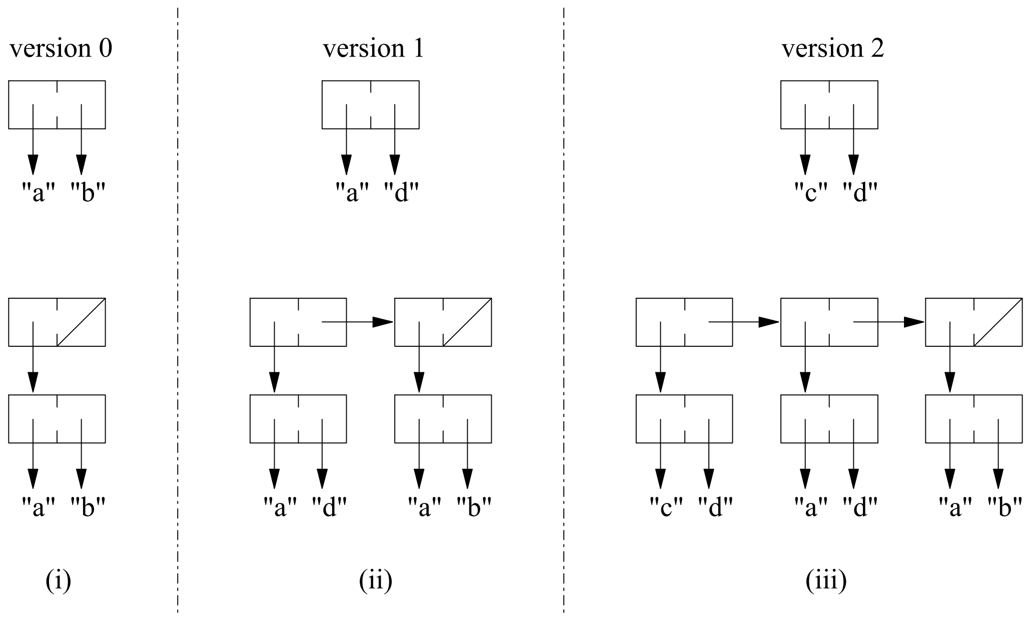

Figure 4.2: A single logical cons cell as seen logically (high level) and physically (lower level)

vcdr values for any versioned cons cell is a constant time operation. In particular, vcar and vcdr are simply implemented as:

```
vcar(x):=_car(_car(x))
```

```
vcdr(x):=_car(_cdr(x))
```

Since these functions are implemented at this level, the Pattern Matcher/Rule Controller, the CGU and the drivers built on top can use the versioning features it without having to worry about keeping the proper values for the different versions themselves.

4.3 Common Graph Utility

The CGU interface is based on the manipulation of the Global Terminology and one or more partial databases. Both are stored using the S-expression encoding presented in Table 4.1.

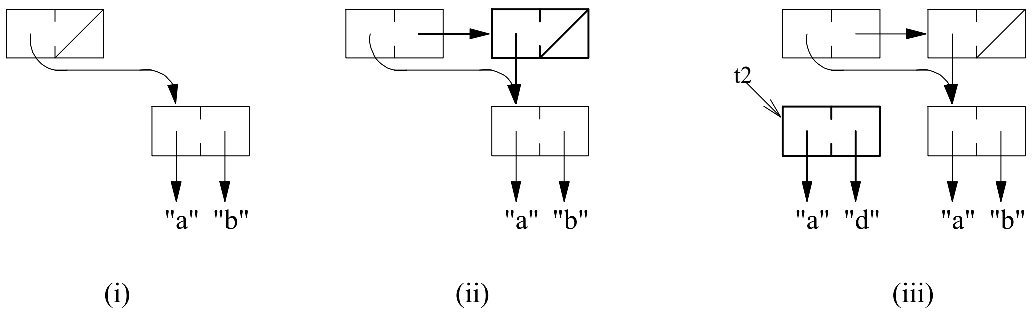

Figure 4.3: Stages of the vrplacd algorithm

(terminal symbols)	*(definition)*
$\langle \mathit{CGUData} \rangle$	$((\langle \mathit{GlobalT} \rangle)\, \langle PD_1 \rangle \ldots \langle PD_n \rangle)$ where $n \geq 0$
$\langle \mathit{GlobalT} \rangle$	$\mathit{nil} \mid (\langle LHS \rangle \cdot \langle \mathit{GlobalT} \rangle)$
$\langle LHS \rangle$	$\mathit{top} \mid \langle \mathit{atom} \rangle$[1]
$\langle PD_i \rangle$	$(\langle PDName \rangle\, (\langle \mathit{Stack}1 \rangle)\, (\langle \mathit{Stack}2 \rangle)\, (\langle \mathit{NodesList} \rangle))$
$\langle PDName \rangle$	$\langle \mathit{atom} \rangle$[2]
$\langle \mathit{Stack}1 \rangle$	$\mathit{nil} \mid (\langle NodeName \rangle\, \langle \mathit{Description} \rangle)\, \langle \mathit{Stack}1 \rangle$
$\langle \mathit{Stack}2 \rangle$	$\mathit{nil} \mid (\langle NodeName \rangle\, \langle \mathit{Description} \rangle)\, \langle \mathit{Stack}2 \rangle$
$\langle NodeName \rangle$	$\langle \mathit{atom} \rangle$[3]
$\langle \mathit{NodesList} \rangle$	$\mathit{nil} \mid (\langle NodeName \rangle \cdot \langle \mathit{NodesList} \rangle)$
$\langle \mathit{atom} \rangle$	Lisp atom
$\langle \mathit{Description} \rangle$	S-expression for a description in $\mathcal{DLFDA}$

Table 4.1: S-expression encoding for CGU data

Initially, the CGU data is the empty list. Either inclusion dependencies can be added to the global terminology or partial databases can be created and manipulated. A typical session for an application using the CGU consists of the (optional) definition of the global terminology by one or more calls to `addGlobalIncDep`; then a call to create a partial database with `createPD` and its subsequent `setPD` call; thereafter (also optionally) create some initial nodes; then push description-node pairs to stack1 and; finally a call to the expand functions. Consider for example the code in Figure 4.4 showing a typical use of the CGU interface.

Lines 1 and 2 add each one an inclusion dependency to the global terminology.

[1]The atom representing the name of an atomic concept
[2]The atom representing the name of the partial database
[3]The atom representing the identifying name of the node

```
1: addGlobalIncDep( (atomic C1),(forall A1 (atomic D)) )
2: addGlobalIncDep( (atomic C2),(forall A2 (atomic E)) )
3: createPD(SamplePD)
4: setPD(SamplePD)
5: createNode(NodeX)
6: createNode(NodeY)
7: pushStack1((atomic C1), NodeX)
8: pushStack1((atomic C2), NodeY)
```

Figure 4.4: Sample usage of the CGU interface

The function `addGlobalIncDep` recognizes that both left hand sides (LHS) are atomic concepts and adds one entry to the list with the atom for each of the atomic concepts. The right hand side (RHS) is stored as a property of the atom representing the LHS using the identifier GlobalT. Line 3 creates an empty partial database with the name SamplePD. At this point the partial database is nothing more than the list with the name and null entries for the two stacks and the nodes list. The S-expression for the partial database looks like: `(SamplePD () () ())`. Line 4 just sets the global variable for currentPD to point to the partial database SamplePD. Lines 5,6 create the nodes NodeX and NodeY which consists in adding the atoms NodeX and NodeY to the list of nodes for the SamplePD and adding a property called SamplePD to each one of them. Finally, lines 7 and 8 will push the description-node pairs `(atomic C1)-NodeX` and `(atomic C2)-NodeY` to stack1. After the example code in Figure 4.4 has been executed, the S-expression for the CGUData is:

```
((C1 C2) (SamplePD (((atomic C2) NodeY)((atomic C1) NodeX)) () (NodeX
NodeY)))
```

Figure 4.5 shows the diagrammatic representation of the CGUData after some inclusion dependencies have been added, n partial databases have been created and m nodes exist in partial database PD1

The complete S-expression for Figure 4.5 is:

$\texttt{((}\langle LHS\rangle\ldots\langle LHS\rangle\texttt{)(PD1 (}\langle Stack1\rangle\texttt{)(}\langle Stack2\rangle\texttt{)(N1...Nm))}\ldots\langle PD_n\rangle\texttt{)}$

Note that atoms' property lists do not appear in S-expressions. Therefore, the property lists for the left hand side of the inclusion dependencies and for the nodes in the partial database are presented separately in Table 4.2

[4]The property list for an atom is returned when we evaluate that atom, hence the title of the second column is *eval*($\langle atom\rangle$).

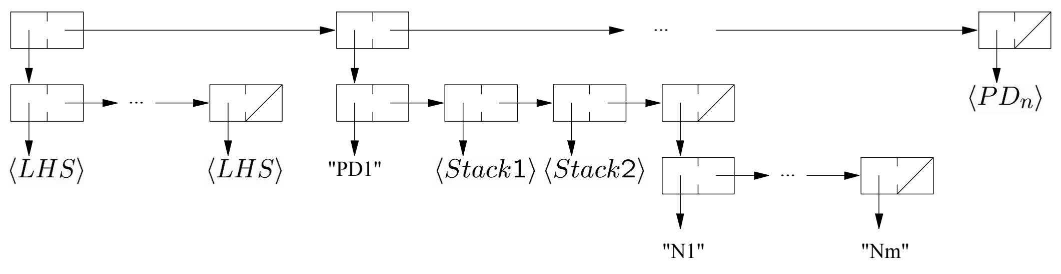

Figure 4.5: CGU DATA

$\langle atom \rangle$	$eval\,(\langle atom \rangle)^4$
C1	`(GlobalT (forall A1 (atomic D)))`
C2	`(GlobalT (forall A2 (atomic E)))`
NodeX	`(SamplePD ())`
NodeY	`(SamplePD ())`

Table 4.2: PROPERTY LISTS FOR ATOMS CREATED DURING EXECUTION OF CODE IN FIGURE 4.4

As mentioned above, typically after the code in Figure 4.4 has been executed, the commands that will follow are calls to `expand1` and/or `expand2`. However, it is possible but unlikely, that some of the other graph manipulation functions are used directly on the partial database before calling the expand functions. Consequently, having the terminology and the initial description/nodes in the stacks, the calls to `expand1` and `expand2` will attempt to create in the graph the same logical implications that the descriptions represent and at the same time check if that description or its implications did not introduce an inconsistency. These expand functions are precisely the core of the CGU in terms of the reasoning performed on the graph so we will review them in more detail below.

4.3.1 Expand functions

The `expand1` function roughly implements the rules that have a deterministic effect on the graph. It generally covers the rules applied to all the constructors of $\mathcal{DLFDA}$ except the general concept disjunction and the attribute typing. These last two are the ones covered by `expand2`. The structure of the `expand1` algorithm is really simple as we can see in algorithm 2.

Note that `expand1` will return TRUE when, having deposited the descriptions in the nodes requested, no clash is detected, and it will return FALSE as soon as the

Algorithm 2 Expand1 general structure

```
1: while stack1 not empty do
2:    (D,n):=popStack1()
3:    Attempt to deposit D into node n and trigger the appropriate actions
4:    if clash detected then
5:       return false
6:    end if
7: end while
8: return true
```

first inconsistency is detected. The phrase “trigger the appropriate actions” in line 3 is more or less analogous to applying the equivalent rule of the tableau algorithm presented in chapter 2 of [5]. The complete set of rules is presented below:

When trying to add D to n apply:

Deterministic Rules

C rule

1. If $\neg C$ is in $\mathcal{L}(n)$ then clash.
2. Otherwise,
   (a) Add C to $\mathcal{L}(n)$
   (b) If $C \sqsubseteq D_1$ is in $\mathcal{T}$, then try to add D_1 to n
   (c) Check if the node was waiting for the concept C and trigger the action requested.

$\neg C$ rule

1. If C is in $\mathcal{L}(n)$ then clash
2. Otherwise,
   (a) Add $\neg C$ to $\mathcal{L}(n)$
   (b) Remove all the triggers that were waiting for the concept C.

$\forall Pf.\neg C \sqcup D_2$ rule

1. Traverse the path Pf as far as possible.

2. If you get to the end of the path Pf at node n_{Pf}, check if C is in $\mathcal{L}(n_{Pf})$. If it is, then add D_2 to n., If it is not, then wait for C at node n_{Pf} and then try to add D_2 to n.
3. If the complete path Pf does not exist, then wait for the path to be completed and then proceed as above.

$D_1 \sqcap D_2$ rule

1. Try to add D_1 to n.
2. Try to add D_2 to n.

$Pf_1 = Pf_2$ rule

1. Traverse the path Pf_1 creating nodes and edges as necessary.
2. Traverse the path Pf_2 creating nodes and edges as necessary.
3. Add an equality arc between the two nodes n_{Pf_1}, n_{Pf_2}.

$Pf_1 \neq Pf_2$ rule

1. Traverse the path Pf_1 creating nodes and edges as necessary.
2. Traverse the path Pf_2 creating nodes and edges as necessary.
3. Add an Inequality arc between the two nodes n_{Pf_1}, n_{Pf_2}.

$D@f$ rule

1. Create a new node n_{new}. Add an edge f from n_{new} to n.
2. Try to add D to n_{new}.

$\forall f.D$ rule

1. If there is an edge f from n to any node n_f, try to add D to n_f.
2. If n is *root*, create a new node called f and try to add D to f.
3. Otherwise, let the non-deterministic rule to handle the description.

$C : \{A_1, \ldots, A_n\} \rightarrow id$

1. If any of the edges in $\{A_1, \ldots, A_n\}$ is missing, wait for all the edges to be added. When added, proceed to 2.

2. When all the edges are present, go to each of the nodes reached and check all the corresponding incoming edges to see if you get to a C. If when checking all the nodes reached from n by traversing the edges $\{A_1, \ldots, A_n\}$ it is detected that they coincide with the nodes reached by traversing the edges $\{A_1, \ldots, A_n\}$ from another node y, and that $C \in \mathcal{L}(y)$ then, add an equality arc between n and y.

Non-Deterministic Rules

$\forall f.D$ rule

1. If there is an edge f from n to any node n_f, try to add D to n_f
2. Otherwise, create a new node n_{new}, add an edge f from n to n_{new} and try to add D to n_{new}.

$D_1 \sqcup D_2$ rule

1. Set a check point
2. Add D_1 to n. if it leads to a clash backtrack to the previous checkpoint. If no clash found, return TRUE.
3. If clash detected in 2, then Add D_2 to n. If no clash detected return TRUE.

Note that the two "waits" in the $\forall Pf.\neg C \sqcup D_2$ rule are handled using some special descriptions placed in $\mathcal{L}(n)$. The addition of equality arcs is handled using a node merge algorithm similar to encoding equivalence classes in a tree, commonly known as the disjoint set union algorithm. Also note that the phrase "try to add D to n" represent the recursive application of the rules since D can be any description while "add C to $\mathcal{L}(n)$" is an unconditional addition of the concept C to the labels of node n. In our particular implementation, "try to add D to n" is translated to push $\langle D, n\rangle$ to the stack1. Also note that `expand1` has an independent stack from `expand2` and whenever descriptions handled with the non-deterministic rules appear in the stack1 they are just pushed to the top of stack2.

For the algorithm of `expand2`, we have rougly the implementation of the two non-deterministic rules as we can see in algorithm **??**. The reason to have those two sets of rules was to be able to handle the query optimization part more efficiently. We noticed that only the deterministic rules were needed to create the query graph including the terminology. On the other hand, the expand2 rules were still needed to have a complete DL reasoner.

Another peculiarity of our CGU is that it also assumes the existence of a virtual node called *root*. There is only one *root* in every partial database. This node is virtual because it does not materially exist in the list of nodes for any of the partial databases but we can still instruct the CGU to add descriptions to the root. These descriptions can only use the constructs $\forall Pf.C$, $Pf_1 = Pf_2$, $Pf_1 \neq Pf_2$, where none of the Pf are just *id*. It is clear to see that if we only use those constructors then $\mathcal{L}(root)$ is always empty and therefore it is not needed. Also, for every node in the partial database, there is one edge labeled "*n*" from the *root* node to the "*n*" node.At this point the existence or not of this special node is transparent, but its use will become evident when we discuss the CQO driver.

4.4 DL Reasoner

The algorithmic details for the DL Reasoner driver have been mostly revealed in chapter 3. This is partially because the implementation of the reasoner functions can be mapped, almost directly, to simple calls of the CGU functions with very little extra manipulation added. However for the purpose of understanding the subtle minor manipulation required, consider the sample session of the DL reasoner below:

```
1: inKnowledgeBase(AbstractKB)
2: implies_c((atomic C1), (forall A (atomic C3)))
3: implies_c((atomic C2), (and (atomic C1) (forall A (atomic C4)))
4: implies_c((atomic C3), (not (atomic C4)))
5: satisfiable((atomic C2))
```

In this example we are interested in knowing whether $C2$ is satisfiable or not given the terminology presented above. The terminology says that $C1$ are those things that have an attribute A that is a $C3$ (line 2). It also says that $C2$ are those objects that among other things are $C2$ and that have an A that is a $C4$ (line 3), and it also specify that $C3$ and $C4$ are disjoint (line 4). It is not hard to see, that given the semantics for the DL constructors and the terminology, $C2$ is not satisfiable since it is required to have an attribute A to something that is at the same time a $C3$ and a $C4$ which clearly contradicts the last inclusion dependency of the terminology. Lets see then how the satisfiable function works so it detects that contradiction. The algorithm for the satisfiable function is:

Require: C

```
pushStack1( (forall x C), root)
if expand1() then
  return expand2()
else
  return false
end if
```

After running `expand1` in line 2, the CGU creates a node x with $C2, C1$ in $\mathcal{L}(x)$ and it will try to add $\forall A.C3$ to n and it will also try to add $\forall A.C4$ to n. Since there is no outgoing edge A from n, the processing of those two calls will be deferred to the call of expand2. The call to expand2 in line 3 will then create a new node $n1$ and it will add an edge from n to $n1$ labeled A. It will also deposit $C4, C3$ in $\mathcal{L}(n1)$ and when trying to add $\neg C4$ to $n1$ it will detect that $C4$ is already in that node and then it will return false i.e. $C2$ is not satisfiable. The graph created is presented in Figure 4.6.

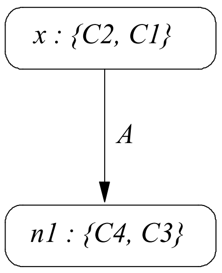

Figure 4.6: Graph created to test the satisfiability of concept C2

For the subsumption test, the algorithm is pretty similar but with two changes as we can see below:

```
Require: D2, D1
1: $C \leftarrow$ transform $D1 \sqcap \neg D2$ to NNF
2: pushStack1( (forall X (C), root)
3: if expand1() then
4:   if expand2() then
5:     return false
6:   else
7:     return true
8:   end if
9: else
10:   return true
```

```
11: end if
```

The two changes are that we transform the description $D1 \sqcap \neg D2$ into negation normal form and that we invert the result of the expand functions. This is because we are actually checking for the unsatisfiability of that description for the test `subsumes(D2,D1)` to be true. In practice, we could replace lines 2-10 by returning the negation of the call to `satisfiable(C)`. For example, we could remove the line 3 from the terminology definition at the beginning of this section (4.4) and then test whether $C2 \sqsubseteq \forall A.(C3 \sqcap C4)$. We can see that when we transform $C2 \sqcap \neg(\forall A.(C3 \sqcap C4))$ to NNF we will end up with $C2 \sqcap \forall A.(\neg C3 \sqcup \neg C4)$ which requires the non deterministic check of both $\neg C3$ and $\neg C4$. It is clear that the check for satisfiability will return false proving that indeed $\forall A.(C3 \sqcap C4)$ subsumes $C2$ as expected.

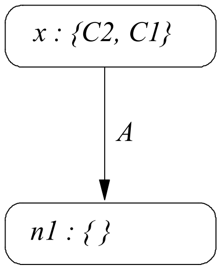

Figure 4.7: GRAPH CREATED AFTER TESTING WHETHER OR NOT D2 SUBSUMES D1.

4.5 CQO

The conjunctive query optimizer, as mentioned throughout this research work, shares several processes with traditional DL reasoning procedures. It is then expected that our proposed query optimizer makes intensive use of the CGU to find the execution plan for a given query Q, in particular of the expand functions. As presented in chapter 3, the CQO interface only has two functions. One to input the schema and one to get the plan of a query. The algorithmic details of the addtoSchema function are trivial since they are almost literal calls to the addLocalIncDep function in the CGU. However, the interesting issue, is how we can actually capture the database schema using such inclusion dependencies.

4.5.1 Capturing conceptual design

For presentation purposes, we will use the widely used ER diagram tool to describe the elements captured in a database. In simple terms, an ER diagram contains three basic elements: Entities, Attributes and Relationships. Assume we have the ER diagram presented in Figure 4.8.

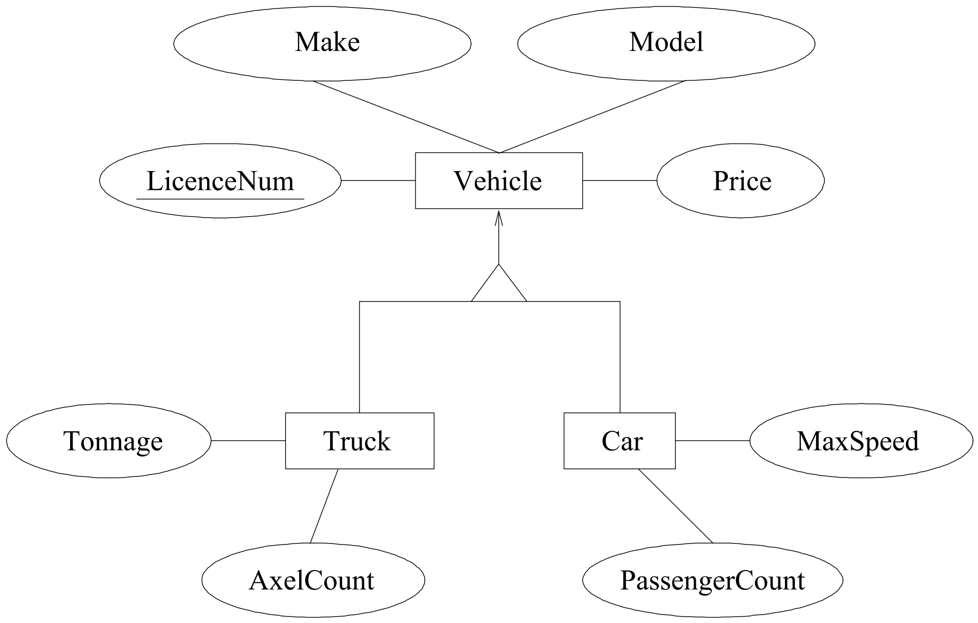

Figure 4.8: ER diagram for a Vehicles DB

Using the first three rows in Table 3.7 we can capture the following constraints:

- There are three Entities: Vehicles, Trucks and Cars
  - `(class (atomic Vehicle))`
  - `(class (atomic Truck))`
  - `(class (atomic Car))`
- Vehicles have a LicenseNum[ber], a Maker, a Model and a Price
  - `(property (atomic Vehicle) LicenseNum)`
  - `(property (atomic Vehicle) Maker)`

  - `(property (atomic Vehicle) Model)`
  - `(property (atomic Vehicle) Price)`

- A Truck is a kind of Vehicle. A Car is a kind of Vehicle
  - `(is-a (atomic Vehicle) (atomic Truck))`
  - `(is-a (atomic Vehicle) (atomic Car))`

- All the Vehicles are Cars or Trucks
  - `(is-a (atomic Vehicle) (or (atomic Car) (atomic Truck)))`

- No Car is a Truck and no Truck is a Car, i.e. they are disjoint.
  - `(is-a (and (atomic Car) (atomic Truck)) (bottom))`
  - `(is-a (atomic Car) (not (atomic Truck)))`

- LicenseNum[ber] is the key for Vehicles
  - `(is-a (atomic Vehicle) (pfd (atomic Vehicle) (LicenseNum) ()))`

Moreover, although not present in the ER diagram and somewhat contrived for the sake of showing the capability, we could express the following constraints:

- Foreign keys. Consider for example that the maker attribute is indeed another entity.
  - `(is-a (atomic Vehicle) (forall Maker (atomic MakerType))`

- Inverse Attributes. A Vehicle must be driven by a Person
  - `(is-a (atomic Vehicle) (at (atomic Person) Drives))`

Essentially, all the class definitions are stored directly as primitive concepts, all the property definitions declare the attributes for the primitive concepts and all the is-a are translated directly to inclusion dependencies. For example:

```
addtoSchema((is-a (atomic Vehicle) (or (atomic Car) (atomic Truck))))
```

is simply translated to the call:

```
addGlobalIncDep((atomic Vehicle) (or (atomic Car) (atomic Truck)))
```

4.5.2 Capturing Physical Design

To capture the physical design, we basically generalize the notion of an index so it can be used to characterize not only indices as known in databases but also simple table scans and even field navigation within a record are abstracted in our design as indices. In the CGU, an index is captured as the subgraph presented in Figure 4.9.

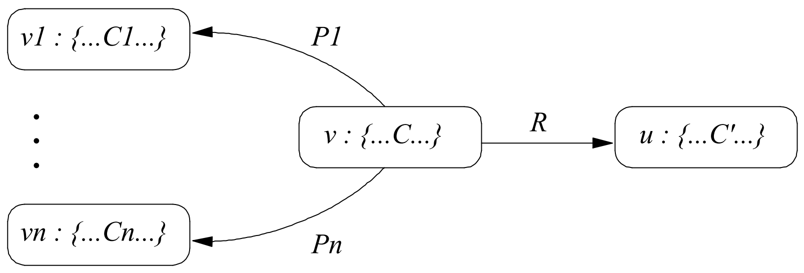

Figure 4.9: Subgraph characterizing an index in the CGU

Where C is called an index; $P1$,...,Pn, R are special attributes, Pi's being the parameters of the index and R the resulting object of type C'. $C1$,...,Cn are respectively the types of the parameters $P1$,...,Pn.

Consider now that at the physical level we have a table "Vehicle" with four columns: LicenseNum, Maker, Model, Price and that we have an index on LicenseNum and an index on Maker. At the physical level could say:

- Vehicles are records identified by an address on a table in memory.
  - `(address (atomic Vehicle))`
  - The above is transformed to `addLocalIncDep((atomic Vehicle),(atomic Address))`
- LicenseNum, Model and Price are integers; Maker is a string
  - `(integer (atomic LicenseNumType))`
  - `(integer (atomic ModelType))`
  - `(integer (atomic PriceType))`
  - `(string (atomic MakerType))`

- We can scan the whole Vehicle table
  - `(index (atomic VehicleIndex) 0)`
- There is an index on LicenseNum
  - `(index (atomic LicenseNumIndex) 1)`
  - `(parameter (atomic LicenseNumIndex) P1 (LicenseNumType))`
- There is an index on Maker
  - `(index (atomic MakerIndex) 1)`
  - `(parameter (atomic MakerIndex) P1 (MakerType))`
- The Vehicle record has a Price field
  - `(index (atomic PriceField) 1)`
  - `(parameter (atomic PriceField) P1 (PriceType))`

Notice that the index on LicenseNum is defined analogously to the MakerIndex and that we will have to define an index on each of the 4 fields to indicate the capability of extracting the field given the address of the record. Notice that in our simple model, we then only recognize integers, strings, addresses, and indices and we do not really differentiate among different types on indices.

Similar to the processing of the schema constraints at the logical level, the constraints at the physical level are also translated into axioms in the terminology. Thus, for example, among other things, the following will be executed for the above constraints:

```
addGlobalIncDep((atomic PriceType),(atomic Integer))
addGlobalIncDep((atomic MakerType),(atomic String))
addGlobalIncDep((atomic vehicle),(atomic Address))
addGlobalIncDep((atomic MakerIndex),(and (atomic INDEX)
                                    (and (forall P1 (atomic MakerType))
                                    (and (forall R (atomic vehicle))
                                         (equal (R maker) (P1))))))
addGlobalIncDep((atomic PriceField),(and (atomic INDEX)
                                    (and (forall P1 (atomic vehicle))
                                    (and (forall R (atomic PriceType))
                                         (equal (P1 price) (R))))))
```

The last two are created for every index and essentially they capture the types of all the parameters and the type of the result. They also need the equality condition that given a combination of the parameters, the returned element will have those values in their appropriate attributes.

4.5.3 Compiling a Plan

Once we have defined the database schema we can now generate plans for the desired conjunctive queries using the function `compile(Q)`. The algorithm used is presented in Algorithm 3:

Algorithm 3 Conjuctive Query Optimizer Main Module

```
1: Receive $Q$
2: Normalize $Q$ to $Q'$
3: Create description $D_Q$ from $Q'$
4: Create $T = \{D_{T_1 Q'}, \ldots, D_{T_n Q'}\}$ from $\neg D_Q$
5: Create Query Partial Database $PD_Q$
6: Push $D_Q$-root to Stack1
7: Create Plan Partial Database $PD_P$
8: Initialize Plan $P$
9: while $T$ is not empty do
10:    Call expand1 on $PD_Q$
11:    if no further binding is possible then
12:       return fail
13:    end if
14:    Select next node $n$ to bind from $PD_Q$ and mark it as "bound"
15:    Push the [index] definition of the node $n$ in the root of $PD_P$
16:    Update the Plan $P$ with the new binding
17:    Deal with the R attribute. (Copy or check in the plan) and update the plan
       and the graph as appropriate.
18:    Call expand1 on $PD_P$
19:    while head(T) on $PD_P$ is inconsistent do
20:       remove head(T) from $T$
21:    end while
22: end while
23: return $P$
```

The algorithm is basically a two phase approach similar to the one presented in [18]. In the first phase a query graph is created after the query is transformed into a DL description, and then, in the second phase, a differential approach is used to generate the plan. The plan graph is originally empty and as nodes are being bound in the query graph they are being added to the plan graph and their correspondent actions in the plan are included.

Normalization in line 2 refers to the process of converting the query from the grammar described in chapter 3 to a more convenient form that allow us to transform it to a description D_Q in a straightforward way. During this transformation, additional concepts are added to help us identify three types of variables: Query Variables, Iterative Variables and Existential Variables. The Query Variables (QVAR) refer to those that are in the outermost select clause, the iterative variables (IVAR) are the ones that will be joined in an iterative semantics and the existential variables (EVAR) refer to the existentially quantified variables. The set T refers to each one of the disjuncts obtained by normalizing $\neg D_Q$.

Once D_Q is deposited in the root of the Query Partial Database (PD_Q) we enter into a loop that will try first to create the query graph by expanding the description in line 10 during the first iteration. At the end of this first expansion the query graph will look similar to a universal plan as the one suggested in [2]. From now on, in each one of the next iterations it will try to bind one more iterative variable until the plan is completed or no more variables can be bound. In every cycle, when a variable is bound, it is copied to the plan graph by pushing into the stack1 of the Plan Partial Database (PD_P) the corresponding description of the iterative variable that was bound in the current iteration. Additionally special code was added to consider the cases were the binding of an index results in an attempt to bind a variable already bound (resulting in the addition of a check clause to the plan) and when binding a node in the plan graph that is equal to more than one node containing query variables. At the end of the loop, we check if this new variable binding resulted in one of the conditions of the query being met (translated as one of the elements of T leading to an inconsistency in the plan graph). Every time a condition is met, it is removed from T, therefore our termination condition is when T is empty (i.e. the query is now answered by the plan).

To give the reader an example of what will be the result of this process, consider the query in table 4.3:

Taking that query as input, we can then translate it into a canonical form that makes the translation of the query into a description a straightforward process. The canonical form uses a slightly different query language that allows a n-ary from construct (from $Q_1 \ldots Q_n$), where n can even be 0. An empty from has the semantics of the null tuple. The canonical query form looks like this:

In the canonical form, $v_{Q_1} \ldots v_{Q_r}$ are the query variables, $v_{I_1} \ldots v_{I_s}$ are the

Give me the price of the vehicles whose maker is :m

```
(select vprice, :m
(from (as vehicles v)
      (from (as PriceType vprice)
            (from (as param :m)
                  (from (as MakerType :m)
                        (from (equal v (price) vprice ())
                              (equal v (maker) :m ())))))))
```

Table 4.3: PRICE OF VEHICLES FROM MAKER :M

```
(select $v_{Q_1} \ldots v_{Q_r}$
(from  (as $C_1$ $v_{I_1}$)
       ⋮
       (as $C_r$ $v_{I_s}$)
       (elim  (select $v_{E_1} \ldots v_{E_t}$
              (from  (as $C_1$ $v_{E_1}$)
                     ⋮
                     (as $C_r$ $v_{E_u}$)
                     (from
                            (equal $v_{E_1}$ ($Pf_1$) $v_{E_2}$ ($Pf_2$))
                            (equal $v_{E_{v-1}}$ ($Pf_{v-1}$) $v_{E_v}$ ($Pf_v$))))))))
```

Table 4.4: QUERY CANONICAL FORM

iterative variables and $v_{E_1} \ldots v_{E_u}$ are the existentially quantified variables. Also note that

$$\{v_{Q_1} \ldots v_{Q_r}\} \subseteq \{v_{I_1} \ldots v_{I_s}\} \cup \{v_{E_1} \ldots v_{E_t}\}$$

and that in turn

$$\{v_{E_1} \ldots v_{E_t}\} \subseteq \{v_{E_1} \ldots v_{E_u}\} \cup \{v_{E_1} \ldots v_{E_v}\}$$

Thus, the canonical transformation for our query is presented in Table 4.5:

Then from the canonical form, getting the description D_Q is easy. It consists in translating every query variable v_{Q_i} into $\forall v_{Q_i}.QVAR$, every iterative variable v_{I_i} into $\forall v_{I_i}.IVAR$, every `(as` C_1 v_{I_1}`)` into $\forall v_{I_i}.C_i$, every `(equal` v_{E_1} `(`Pf_1`)` v_{E_2} `(`Pf_2`))` into `(equal (`$v_{E_1}.Pf_1$`) (`$v_{E_2}.Pf_2$`))` and every existential variable into $\forall v_{E_i}.C_i$, and finally express the conjunction of all of the above. For our particular query, the description D_Q is presented in Table 4.6:

```
(select vprice, :m
(from (as vehicles v)
      (as PriceType vprice)
      (as param :m)
      (as MakerType :m)
      (elim (select v, vprice, :m
            (from
                  (from
                       (equal v (price) vprice ())
                       (equal v (maker) :m ()))))))))
```

Table 4.5: Query after transformation of Canonical form

```
(and (forall vprice (atomic QVAR))
(and (forall :m (atomic QVAR))
(and (forall v (atomic vehicles))
(and (forall v (atomic IVAR))
(and (forall vprice (atomic PriceType))
(and (forall vprice (atomic IVAR))
(and (forall :m (atomic param))
(and (forall :m (atomic MakerType))
(and (forall :m (atomic IVAR))
(and (equal (v price) (vprice))
     (equal (v maker) (:m)))))))))))
```

Table 4.6: Description D_Q

The description $\neg D_Q$ is obtained by negating the whole D_Q directly and then applying the negation normal form. The result is the disjunction of the negation of each of the conjuncts in D_Q as we can see in Table 4.7.

We could use $\neg D_Q$ directly as our condition to terminate the while in line 9, however it is more efficient if we just consider each one of the disjuncts individually. The reason is because the or rule requires an exponential check while checking each one of the disjuncts individually can be done using the deterministic rules. Thus, T as referred in line 4 in the CQO algorithm is just the set with each one of the disjuncts of $\neg D_Q$.

Then, the algorithm continues with the creation of the partial database for the query and the partial database for the plan. The description D_Q is then placed in

```
(or (forall vprice (not (atomic QVAR)))
(or (forall :m (not (atomic QVAR)))
(or (forall v (not (atomic vehicles)))
(or (forall v (not (atomic IVAR)))
(or (forall vprice (not (atomic PriceType)))
(or (forall vprice (not (atomic IVAR)))
(or (forall :m (not (atomic param)))
(or (forall :m (not (atomic MakerType)))
(or (forall :m (not (atomic IVAR)))
(or (nequal (v price) (vprice))
    (nequal (v maker) (:m)))))))))))
```

Table 4.7: DESCRIPTION $\neg D_Q$

the root node of the query PD ready to be expanded in the first iteration. The plan is initialized with an outer keep operation of all the query variables and a single nest operation plus a place holder that will be replaced by subplans in every iteration as more variables are bound in the query graph and copied to the plan graph. This iterative process is what will drive the creation of the plan. In our case, the initial plan looks like this:

```
(keep vprice, :m
   (nest []))
```

In the very first call to `expand1` in line 10, the description D_Q is expanded into the query graph presented in Figure 4.10.

Initially, the only variable that can be bound is the parameter :m. Consequently this is the variable selected to be bound next in step 14. Its selection causes `(param :m)` to be added to the plan and the node :m to be created in the plan graph. The node is created because the algorithm pushes the definition of the node :m in the root of the plan PD and then we call `expand1` on the plan PD (line 18). Then T is checked to see if one or more of the elements when added to the plan PD makes it inconsistent. This step is our stopping condition. An empty T set means that the plan graph now captures all what is required by the query. Since the very first element of T is not inconsistent with the plan graph then we leave it in the set T and the first iteration ends.

At the beginning of the next iteration in line 10, the call to `expand1` in the query PD causes that a new possible node that can be bound be detected. The new node

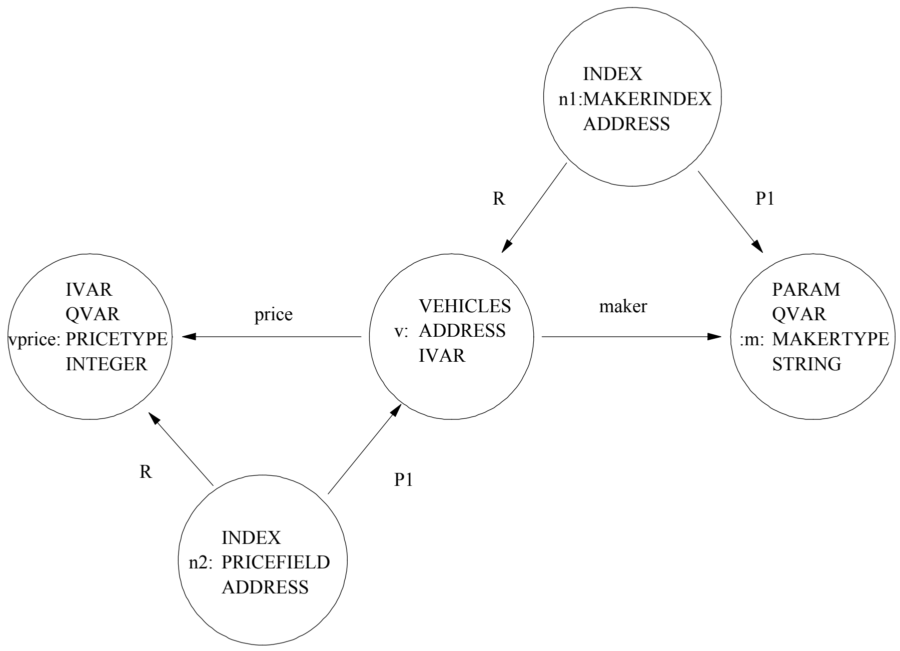

Figure 4.10:

corresponds to the n1 node since its only required parameter :m has been bound. Hence, this new node is the next one bound, its definition pushed to the plan graph, that in turn is expanded. The binding of n1 causes the nesting of the subplan `(scan MakerIndex n1 :m)` indicating that the index MakerIndex is scanned. Since n1 is an index, we need to deal with the result (R) attribute (see line **??**). Dealing with R refers to the addition of the plan the clause `(copyr v n1)`. This expresses that the result of the index scan will be ultimately assigned to the variable v. Then T is checked again and since the first element can not be removed T is left unchanged.

In the final iteration, with :m, n1and v bound and the call to expand1 in query PD detects that n2 can be bound next. It is then bound what causes the addition of the subplan `(scan PriceField n2 v)` and also the addition of `(copyr vprice n2)` when dealing with the R of n2. The expansion of the plan graph after these two last nodes are bound in the query graph causes the plan graph to be completed. Then when the elements of the set T are checked for inconsistency, all the elements are found inconsistent and then removed from T. Since now T is empty the `while` terminates and the final plan is returned. The final plan looks like this:

```
(keep vprice, :m
(nest
   (param :m)
   (scan MakerIndex n1 :m)
   (copyr v n1)
   (scan PriceField n2 v)
   (copyr vprice n2)))
```

Although the correctness proof is beyond the scope of this thesis, we can see that the plan above implements the query from Table 4.3 returning all the prices labeled vprice for the vehicles whose maker is :m.

Chapter 5

Summary and Conclusions

We started this journey with the idea of exploring the similarities between two apparently different processes, but those we believed had many things in common. The two processes, subsumption of descriptions in DL and conjunctive query optimization, shared, in essence, the creation and manipulation of a graph as a way to capturing and reasoning about the knowledge contained in the universe under scrutiny. We then elaborated on the design and algorithms of those common functions to create and manipulate the graphs enclosed in an utility set that we called CGU and finally we discussed an presented the details of how that set of functions can indeed be used to implement a DL reasoner and a simple conjunctive query optimizer. Along the way, there were several lessons learned relating to common functionality between DL reasoning and CQO.

On the DL reasoner side:

a) *Separation of Expand1 and Expand2.* Due to the fact that we have a set of rules that behave deterministically and another set of rules that behave non-deterministically, and considering that the CQO process essentially uses only the deterministic set we had to separate the application of all the deterministic rules in `expand1` from all the other rules in `expand2`. This allowed us to use the non-deterministic rules only for the DL reasoner as required while the CQO could use the rules in `expand1` as many times as needed with out going into a non-deterministic check.

b) *The special root node in the Partial Database.* To encode the fact that every node in the query graph becomes a variable that has to be bound in the plan graph we introduced a special node called "root" that has an edge (representing an attribute) with the same name as the node it is pointing to. This virtual node allowed us to have a starting point for the query graph and gave us the capability of unifying, for that node only, attribute names and node names. Because of this characteristic, when used as a DL reasoner, instead of creating a node x and then reasoning with:

$$T \models D_1 \sqsubseteq D_2 \text{ or } T \models D_1 \sqcap \neg D_2 \text{ is consistent?}$$

we could simple use (from root):

$$T \models \forall x.D_1 \sqsubseteq \forall x.D_2 \text{ or } T \models \forall x.(D_1 \sqcap \neg D_2) \text{ is consistent?}$$

c) *The issue about Query Variables and node equalities.* To identify that a node was also a query variable, we introduced the atomic concept QVAR and we found that even though for DL purposes if could be copied when two nodes merge, it was necessary that we leave the QVAR description in the node where it was originally placed. In this sense we had to handle this atomic description differently from the others when two nodes merge. Our solution was that even when the two nodes will not share the QVAR description, they will still have a way to recognize that the other was a QVAR without having to visit all the nodes in the equivalence class tree.

On the CQO side:

a) *Schema can be very rich.* Using the DL language chosen, we were able to describe many things that were usually described in DB schemas and used for query optimization like entities, properties, foreign keys, etc., but we were also able to include things like disjoint clauses and cover clauses (union of concepts) that basically enrich the schema language and allow us to work on some different rewrites for optimization.

b) *DL fundamentally enriches the power of some rewrites, and also the power of CQL.* We basically found the CQO benefitted from the DL in the sense that we could use the extra knowledge to deduce even more things, one example being the reasoning about duplicate elimination using FDs as presented in [4]

5.1 Future work

At the end of this journey we found that there is a vast world of research directions that we would like to explore because they pose intriguing questions given the existence of the tools we created. Some of those directions include:

- Generalized Path Functional Dependencies. The constructor included in our DL language to define keys was a particular case of the path functional dependencies. We would like to explore more what will be the implications and design problems when using a more general class of functional dependencies like
$$C_1 \sqsubseteq C_2 : \{Pf_1, \ldots, Pf_n\} \rightarrow Pf$$

- Join Order Selection. Our proposed CQO algorithm does not really consider the problem of which variable bind next when there is more than one. The rules required to do that in this setting seem also very valuable and worthwhile exploring.

- Cost Models. Again, indices in our setting, do not distinguish between a table scan from a fast B-plus tree index and from the different types of joins in terms of the cost of the different approaches. A CQO with that capability is highly desirable.

- Join plus union all. Our plan language does not support union and the addition of this single operator is considered to bring the ability to expand the query language to support a more general class of queries.

And finally there are many other considerations that would be nice to explore as different interfaces to the query optimizer, for example we could accept valid conjunctive SQL queries as input and the transform them so that ultimately we could still generate the plans. Another idea is to couple this with the DL reasoning required in the semantic web, in particular for OWL-DL and test how it will behave in that setting. That will also probably required the addition of roles to the set of rules in the algorithms which is also intriguing.

As we can see there are many other possible research directions out of this and we hope that we can see some of them materialize in the not so far future.

Appendix A

Naive Lisp (NLISP)

The Naive Lisp (including the memory manager) requires the following files to be compiled:

- store.h
- snode.c
- soper.c
- evaluate.c
- parser.h
- parser.c
- hash.c
- memmgr.c

The main header file for the Lisp module is the store.h header file. This file declares all the functions available to the upper layers as we can see below.

```
/***************************************************************
                          store.h  -  description
                             -------------------
    begin                : Tue Aug 3 2004
    copyright            : (C) 2004 by Alex Palacios
    email                : japalaciosvilla@uwaterloo.ca
 *************************************************************/

//Max length of strings in string cell
#define STRSIZE 8

// define S cell types
#define INTTYPE 0
#define STRTYPE 1
#define CONSTYPE 2
#define NILTYPE 3

// define TRUE and FALSE
#define TRUE 1
#define FALSE 0
#define boolean short int

typedef union mblk *Sptr;

typedef union mblk {
    struct stringcell {
        char strdata[STRSIZE+1];
        Sptr next;
    } strstore;

    struct cell {
        short int stype;
        int rcount;
        Sptr nexthash;
        union Stypes {
            struct type0 {
                Sptr def;
                union atmcont {
                  int intval;
                  Sptr next;
                } c;
            } atom;

            struct type2 {
                unsigned int version;
                Sptr car;
                Sptr cdr;
            } list;
        } t;
    } Scell;

    struct freeblk {
        Sptr next;
    } freecell;

} MemBlock;

//Functions in snode
Sptr getNil(void);
void release (Sptr);
Sptr newSnode(short int);
Sptr assign(Sptr);
Sptr decrement(Sptr);
Sptr stringToStrStore(char*);
void strStoreToString(Sptr,char*);
int stringcmp(Sptr, char*);
int storecmp(Sptr, Sptr);
void finish(int);

//Functions in soper
int getStype (Sptr) ;
int getAtomIntVal (Sptr);
Sptr getAtomStrStore (Sptr);

Sptr cons(Sptr, Sptr);
Sptr car(Sptr);
Sptr cdr(Sptr);
Sptr rplaca(Sptr, Sptr);
Sptr rplacd(Sptr, Sptr);
Sptr cvr(char*,Sptr);
boolean Atom(Sptr);
boolean Null(Sptr);
boolean Listp(Sptr);
```

```
Sptr set(Sptr, Sptr);
Sptr eval(Sptr); // in evaluate.c
Sptr mkatom(char*);
Sptr lread(char*);
// new in soper
boolean eql(Sptr, char*);
boolean eq(Sptr, Sptr);
boolean neq(Sptr, Sptr);
int atomcmp(Sptr, Sptr);
int compareS(Sptr, Sptr);
Sptr gensym(char*);
Sptr getprop(Sptr, Sptr);
Sptr putprop(Sptr, Sptr, Sptr);
Sptr remprop(Sptr, Sptr);
Sptr getpropL(Sptr, Sptr);
Sptr putpropL(Sptr, Sptr, Sptr);
Sptr rempropL(Sptr, Sptr);
Sptr append(Sptr, Sptr);
Sptr last(Sptr);
Sptr insertSortList(Sptr, Sptr);
Sptr listUnion(Sptr, Sptr);
boolean member(Sptr, Sptr);
Sptr assq (Sptr, Sptr);

Sptr getp_fd(Sptr, Sptr, Sptr (*) (Sptr));
Sptr putp_fd(Sptr, Sptr, Sptr, Sptr (*) (Sptr), Sptr (*) (Sptr,Sptr));
Sptr remp_fd(Sptr, Sptr, Sptr (*) (Sptr), Sptr (*) (Sptr,Sptr));
Sptr findPrevList(Sptr, Sptr);
Sptr findListHead(Sptr, Sptr);
Sptr addList (Sptr, Sptr);
Sptr removeList (Sptr, Sptr);
Sptr updateList (Sptr, Sptr, Sptr);
Sptr getprophead(Sptr, Sptr);
Sptr getpropLhead(Sptr, Sptr);
Sptr getph_fd(Sptr, Sptr, Sptr (*) (Sptr));
```

Appendix B

Versioning

The versioning module requires only two files to be compiled. The files are:

- version.h
- vlsversion.c

Those files need to be compiled on top of the NLisp files. The version.h header file is the one declaring the interface to the upper layers. The header file is presented below:

```
/***************************************************************
                          version.h  -  description
                             -------------------
    begin                : Wed Jan 12 2005
    copyright            : (C) 2005 by Alex Palacios
    email                : japalaciosvilla@uwaterloo.ca
 ***************************************************************/
void resetLogVer(void);
void vprintS(Sptr);
void vsprintS(char*, Sptr);

Sptr _cons(Sptr, Sptr);
Sptr _car(Sptr);
Sptr _cdr(Sptr);
Sptr _rplaca(Sptr, Sptr);
Sptr _rplacd(Sptr, Sptr);

Sptr cons0(Sptr, Sptr);
Sptr vcons(Sptr, Sptr);
Sptr vcar(Sptr);
Sptr vcdr(Sptr);
Sptr vrplaca(Sptr, Sptr);
Sptr vrplacd(Sptr, Sptr);

Sptr vaddList (Sptr, Sptr);
Sptr vaddendList (Sptr, Sptr, Sptr);

int getVersion(Sptr);
int getCurrentVersion(void);
boolean currentVersion(Sptr);

void pushVer(void);
void popVer(void);
void setPopLog(Sptr, Sptr, Sptr);
```

Appendix C

Pattern matcher/Rule controller

This module requires the following files:

- pattern.h
- pattern.c
- rewrite.c

The header file pattern.h contains the declaration of the functions used in both subcomponents. The pattern.c implements the functions of the Pattern Matcher and the rewrite.c implements the functions for the Rule Controller. The header file is presented below:

```
/***************************************************************
                          pattern.h  -  description
                             -------------------
    begin                : Thu Sep 30 2004
    copyright            : (C) 2004 by Alex Palacios
    email                : japalaciosvilla@uwaterloo.ca
 ***************************************************************/

void initVars(void);
void finalizeVars(void);
Sptr topCopy(Sptr);
Sptr getBindVal(Sptr);
Sptr putBindVal(Sptr, Sptr);
Sptr pushMatchVar(void);
Sptr popMatchVar(void);
Sptr Build(Sptr);
boolean Match(Sptr,Sptr);
Sptr Bindq(Sptr);
boolean ApplyRuleControl (Sptr, Sptr);
Sptr LoadRules(Sptr);
Sptr LoadControl(Sptr);
Sptr UnLoadRules(Sptr);
Sptr UnLoadControl(Sptr);
Sptr initControl(char *, char *);
Sptr finalizeControl(Sptr);
```

Appendix D

CGU

The files required in the CGU module are:

- reasonerapi.h
- reasonerapi.c
- expand.c

The reasonerapi.h contains the interface functions for the CGU. The reasonerapi.c contains the implementation of those functions except for the two expand functions that are implemented in expand.c. The header file is presented below:

```
/***************************************************************
                          reasonerapi.h  -  description
                             -------------------
    begin                : Thu Nov 18 2004
    copyright            : (C) 2004 by Alex Palacios
            Shishir Agarwal
    email                : japalaciosvilla@uwaterloo.ca
 ***************************************************************/

/**
@short Terminate the control program for the Negation Normal Form.

It will basically free the memory used for all the variables in the
control program for the Negation Normal Form. If called in the
```

```
middle of the program the variables will get reloaded in the next
call to negationNormalForm */
void finalizeNNFControl(void);

// define the cases used in Expand1
#define TOP 1
#define BOTTOM 2
#define Atomic_C 3
#define Not_Atomic_C 4
#define NFPD 5
#define And_D1_D2 6
#define Equal_Path 7
#define Path_Id1 8
#define Path_Id2 9
#define Nequal_Paths 10
#define DatF 11
#define Forall_F_D1 12
#define Or_D1_D2 13

#define Delete TRUE
#define Do_Not_Delete FALSE

/**
@short Initializes the temporary patterns used by function in
reasonerapi.c file.
*/
void init_tmpPatterns(void);

/**
@short Release all the temporary patterns that were initialized
by init_tmpPatterns.
*/
void finalize_tmpPatterns(void);

/**
@short Creates a new Partial Database.

Creates a new PD and inserts it in the <list of PDs>
If "CGUData" is NULL then it intializes it also.
```

```
A PD is a list of 5 elements
     1st element --> pointer to "PDid"
     2nd element --> pointer to "poplog"
     3rd element --> pointer to stack1 (used for expand1)
     4th element --> pointer to stack2 (used for expand2)
     5th element --> a pointer to list of nodes.
It returns the pointer to "newPD".
Note: stack1 and stack2 are lists. Each element of the list is a
      pair of 2 things.
        1st thing is node-id and 2nd is description.
@param PDid Atom. Id of PD to be created.
@returns Pointer to the newly created PD.
*/

Sptr createPD (Sptr PDid) ;

/**
@short Sets the currPD to PD with id = "PDid"
Sets the currPD pointer to PD whose Id is "PDid".
@param PDid Atom. Id of the Pd which has to be made current PD.
@returns Pointer to currPD, i.e, PD with id = "PDid"
*/

Sptr setPD(Sptr PDid) ;

/**
@short Deletes the PD with id = "PDid"
Deletes the PD from the PDList and also releases the memory.
@param PDid Atom. Id of the PD to be deleted
*/
void deletePD (Sptr PDid);

/**
@short Prints the PD
Prints the PD data
@param Pointer to PD that has to be printed.
*/
```

```
void printPD(Sptr pd) ;

/**
@short Prints "every thing" in the CGUData.
Prints all the Global Dependencies, Partial databases etc.
*/
void printEveryThing(void);

/**
@short Add a global inclusion dependency
It adds a global dependency. If "CGUData" is NULL then it intializes
it also.
@param LHS It can be either atom or list.
  If its atom, the lhs and rhs of inclusion dependency added remain
  same as LHS and RHS.
  If it is a list and it matches (atomic > C), then lhs of inclusion
  dependency added becomes "C" and rhs remains same as RHS.
  If it is a list and it doesn't match (atomic > C), then lhs of
  inclusion dependency added becomes "top" and rhs becomes
  (or RHS (not LHS))
@param RHS List.
*/

Sptr addGlobalIncDep (Sptr LHS, Sptr RHS) ;
/**
@short Same as addGlobalIncDep
At present, it just calls addGlobalIdep. But the idea is, it can be
used to add inclusion dependencies in only one particular database,
normally in currPD.
@param LHS see description in addGlobalIncDep
@param RHS see description in addGlobalIncDep
*/

Sptr addLocalIncDep (Sptr LHS, Sptr RHS) ;

/**
@short Adds the node to currPD.
Adds the node to currPD. Also adds a property with indicator as id
of currPD and value as null description list in the property list
of that node. If there is a global inclusion dependency with LHS as
"top", then it pushes the RHS of that IDep and this node in stack1.
@param NodeId Atom. The Id which the new node should have.
@returns NodeId
*/
Sptr addNode(Sptr NodeId) ;

/**
@short no idea...ask Alex :)
*/
Sptr newQC(Sptr LHS, Sptr RHS) ;

/**
@short Deletes CGUData.
Deletes all the inclusion dependencies and all the PDs. And thus
releases all the memory.
Sets "everyTing" and "currPD" to NULL. Also calls resetLogVer and
finalizeNNFControl.
*/
void finalizePD(void) ;

/**
@short Returns true if the stack is empty.
Returns true if the stack is empty.
@param stackHead Pointer to a cons cell which is head of the stack
@returns TRUE or FALSE depending on whether stack is empty or not.
*/
boolean isEmptyStack (Sptr stackHead) ;

/**
@short Pushes in stack a pair of description and node.
It forms a stack element which is a list of description and node.
It then pushes that element in the stack.
@param stackHead head of the stack in which to push
@param Desc The description to be added.
@param name The name of the node to be added.
@returns The stackHead
*/
```

```
Sptr pushStack (Sptr stackHead,Sptr Desc,Sptr name) ;

/**
@short Pops and returns the top element of the stack
Pops and returns the top element of the stack.
@param stackHead Pointer to a cons cell which is head of the stack
@returns The top element of the stack.
*/
Sptr popStack(Sptr stackHead) ;

/**
@short Returns the PD pointed to by currPD.
@returns the PD pointed to by currPD.

*/
Sptr getCurrPD(void) ;

/**
@short Adds an edge from "node1" to "node2" with label as "label".
It adds an edge from "node1" to "node2" with label as "label" if
there isn't already an edge from "node1" with "label". After adding
the edge, it checks whether there is an (waitf...) description in
"node1" that can be processed. If that is the case, then it does so.
By adding an edge, it means adding the description
(edge <sink node> <label>) in source node and adding description
(redge <source node> <label>) in sink node.
@param node1 Atom. The source node
@param node2 Atom. The sink node
@param label Atom. The label of thee edge.

*/

void add_Edge(Sptr node1,Sptr node2,Sptr label) ;

/**
@short Checks if inequality edge exists between node1 and node2 or not.
Checks if inequality edge exists between node1 (or its eqclass) and
node2 (or its eqClass) or not.
By existence of in equality edge we mean, existence of description
```

```
(ineq <other node>).
@param node1 Atom. First node
@param node2 Atom. Second node
@returns TRUE if inequality edge exists or FALSE if it doesn't.
*/
boolean exists_Ineqality_Edge(Sptr node1, Sptr node2);

/**
@short Adds equality edge between node1 and node2.
It actually randomly selects either node1 or node2 and sets its
equivalance class to other.
@param node1 Atom. The first node.
@param node2 Atom. The second node.
@returns TRUE if there was no clash while adding the equality and
         FALSE if there was a clash.
*/
boolean add_Equality (Sptr node1,Sptr node2);

/**
@short Adds inequality edge  between node1 and node2.
It adds descriptions (ineq <other node>) in both the nodes indicating
an inequality edge between the two nodes. If the two nodes have
redges with same labels then those nodes which are pointed to by
redges are also made inequal.
@param node1 Atom. first node
@param node2 Atom. second node
@returns TRUE if there was no clash otherwise FALSE.
*/

boolean add_Inequality (Sptr node1, Sptr node2);

/**
@short Traces a given path from a node (or creating the path, if it
doesn't already exist) and returns the final node reached.
@param fromNode Atom. Starting node
@param Path List. List of edges forming the path
@returns The node reached by tracing the path.
*/
Sptr findPath(Sptr fromNode,Sptr Path) ;
Sptr findEdge(Sptr node, Sptr label) ;
void addDescription(Sptr Node,Sptr Desc);
void delDescription(Sptr Node,Sptr Desc) ;
Sptr getEqClass(Sptr Node) ;
Sptr getRHS (Sptr LHS);
Sptr createNode(Sptr lbl) ;
Sptr isDescInNode (Sptr Pat, Sptr node, boolean deletePat) ;
void finalizePatterns(void);
void checkPath(Sptr Ni,Sptr N,Sptr Flist,Sptr C,Sptr D2);
```

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Alin Deutsch, Lucian Popa, and Val Tannen. Physical data independence, constraints, and optimization with universal plans. In *Proceedings of the 25th VLDB Conference*, pages 459–470, Edinburgh, Scotland, 1999.

[3] Alex Borgida and Peter F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, pages 277–308, June 1994.

[4] Vitaliy L. Khizder, David Toman, and Grant Weddell. Reasoning about duplicate elimination with description logic (preliminary report). *Lecture Notes in Computer Science*, 1861:1017–1032, 2000.

[5] Franz Baader, Diego Calavanese, Deborah MacGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.

[6] Sonia Bergamaschi and Bernhard Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Applied Intelligence*, 4(2):185–203, 1994.

[7] Alessandro Artale, Francesca Cesarini, and Giovanni Soda. Describing database objects in a concept language environment. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):345–351, 1996.

[8] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *Journal of Artificial Intelligence Research (JAIR)*, 11:199–240, 1999.

[9] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. Classic: A structural data model for objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 58–67, 1989.

[10] Howard W. Beck, Sunit K. Gala, and Shamkant B. Navathe. Classification as a query processing technique in the candide semantic data model. In *Proceedings of the Fifth International Conference on Data Engineering ICDE*, pages 572–581, 1989.

[11] Sonia Bergamaschi and Claudio Sartori. On taxonomic reasoning in conceptual design. *ACM Transactions on Database Systems*, 17(3):385–422, 1992.

[12] Martin Buchheit, Manfred A. Jeusfeld, Werner Nutt, and Martin Staudt. Subsumption between queries to object-oriented databases. Technical Report RR-93-44, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Erwin-Schrödinger Strasse
Postfach 2080
67608 Kaiserslautern
Germany, 1993.

[13] Jeffrey D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[14] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 95–104, San Jose, Calif., 1995.

[15] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS)*, June 2002.

[16] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[17] Sean Bechhofer, Ian Horrocks, Peter F. Patel-Schneider, and Sergio Tessaris. A proposal for a description logic interface. In *Proceedings of the 1999 Description Logics Worksop*, pages 33–36, 1999.

[18] Petrus Kai Chung Chan. Optimizing OQL on legacy main-memory data structures with existential graphs. Master's thesis, University of Waterloo, 1997.