Describing and Using Query Capabilities of Heterogeneous Sources* (Extended version)

Vasilis Vassalos[†] Yannis Papakonstantinou[‡]

Abstract

Information integration systems have to cope with the different and limited query interfaces of the underlying information sources. First, the integration systems need descriptions of the query capabilities of each source, i.e., the set of queries supported by each source. Second, the integration systems need algorithms for deciding how a query can be answered given the capabilities of the sources. Third, they need to translate a query into the format that the source understands. We present two languages suitable for descriptions of query capabilities of sources and compare their expressive power. We also describe algorithms for deciding whether a query "matches" the description and show their application to the problem of translating user queries into source-specific queries and commands. Finally, we propose new improved algorithms for the problem of answering queries using these descriptions.

1 Introduction

Users and applications today must integrate multiple heterogeneous information systems, many of which are not conventional SQL database management systems. Examples of such systems are Web sources with forms interfaces, object repositories, bibliographic databases, etc. Some of these systems provide powerful query capabilities, while others provide limited query interfaces. Systems that integrate information from multiple sources have to cope with the different and limited capabilities of the sources. In particular, integrating systems must allow users to query the data using a single powerful query language, without having to know about the diverse capabilities of each source. Such systems need descriptions of the query capabilities of the participating sources, i.e., descriptions of the set of queries that each source supports. They also need algorithms for adapting to the diverse capabilities of the sources as specified by the descriptions.

To further motivate the need for source descriptions, let us consider the typical integration architecture of Figure 1. *Mediators* decompose incoming client queries, which are expressed in some common query language, into new common-language queries which are supported by the *wrappers*. Then the wrappers translate the incoming queries into source-specific queries and commands. Both mediators and wrappers need the descriptions:

- The mediators use the description to adapt to the query capabilities of the sources. For example, consider a source that exports a "lookup" catalog *lookup*(*Employee*, *Manager*, *Specialty*) for the employees of a company. The description indicates that this source supports only

*Research partially supported by NSF grant IRI–96–31952, ARO grant DAAH04–95–1–0192, and Air Force contract F33615–93–1–1339.

[†]Computer Science Dept., Stanford University, Stanford, CA 94305. email: vassalos@db.stanford.edu

[‡]Computer Science and Engineering Dept., UCSD, San Diego, CA 92093. email: yannis@cs.ucsd.edu

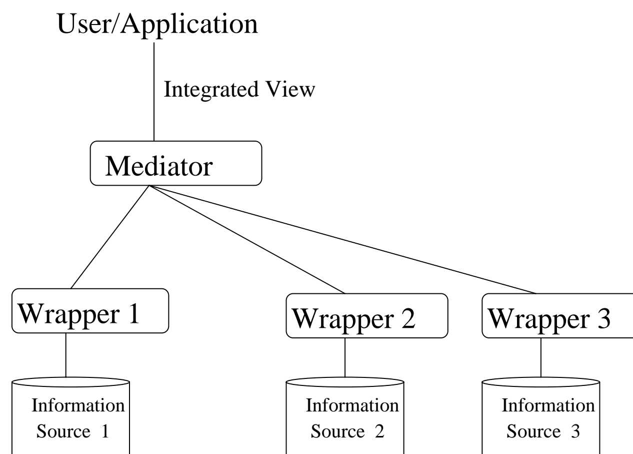

Figure 1: A common architecture for integration

selection queries. Let us now assume that the client requests the managers who have at least one employee specialized in *Java* and at least one employee specialized in *Databases*. Notice that this query is answered with a self join of the *lookup* table on *Manager*. The mediator knows that all the data needed for answering this query reside on "lookup"[1] but finding out *how* to retrieve this data is nontrivial. The query processing algorithm must infer from the description that only selection queries can be submitted and then can come up with the following plan for retrieving the required data: First the mediator retrieves the set of managers of *Java* employees, then the set of managers of *Database* employees, and finally it intersects the two sets.

- The wrappers need descriptions of the source capabilities in order to translate the supported common-language queries into queries and commands understood by the source interface. In particular, each description is associated with *actions* [ASU87, PGGMU95] that perform the translation. Using this approach, in the TSIMMIS project at Stanford we have wrapped a number of real life bibliographic sources.

What is an appropriate language for describing the set of supported queries and its translation to source-specific queries? Since we need descriptions of supported queries along with translating actions, Yacc programs look like a valid candidate. However, Yacc programs do not capture the logical properties of queries — they perceive queries as mere strings. This behavior imposes limitations on both the mediators and the wrappers. For example, the description may specify that an acceptable `WHERE` clause is "`lname='Smith' AND fname='John'`". The wrapper then does not know how to translate a query that asks for "`fname='John' AND lname='Smith'`" because it ignores the commutativity of `AND`. The mediator faces even more severe problems, as we discuss in Section 9.

Since we want to preserve the salient connection between description and translation in Yacc, we propose the use of Datalog variants as a more powerful description language. In particular, context-free grammar rules can be thought of as Datalog rules with 0-arity predicates. The introduction of new languages for describing query capabilities brings up two questions studied in this paper:

[1]In many cases the data reside at multiple sources and the mediator may have to locate them first. However, finding *where* are the important data is a problem orthogonal to *how* they can be obtained. In this paper we only deal with the latter problem.

(i) are these languages expressive enough? (ii) Given a description of the wrappers' capabilities, how can we answer a client query using only queries answerable (*i.e.*, supported) by the wrappers? We refer to this problem as the Capabilities-Based Rewriting (CBR) problem [PGH96, HKWY96]; it is also clearly related to the Answering Queries Using Views problem [LMSS95, RSU95, LRU96] (see Section 4). In this paper, we focus on sources that support conjunctive queries, *i.e.*, their capabilities are a subset of CQ [AHV95]. The contributions of this paper are as follows:

- We introduce the description language p-Datalog, we formally define the set of queries described by p-Datalog programs, and present complete and efficient procedures that (i) decide whether a query is described by a p-Datalog description. This is the algorithm run by the wrapper and note that it also finds out what translating actions must be executed. (ii) decide whether a query can be answered by combining supported queries (the CBR problem). This algorithm is run by the mediator. Our algorithm runs in time non-deterministic exponential in the size of the query and the description, a substantial improvement over the algorithm described in [LRU96], which was non-deterministic doubly exponential.
- We study the expressive power of p-Datalog. We reach the important result that p-Datalog can *not* describe the query capabilities of certain powerful sources. In particular, we show that there is no p-Datalog program that can describe all conjunctive queries over a given schema. Indeed, there is no program that describes all boolean conjunctive queries over the schema.
- We describe and extend RQDL, a provably more powerful language than p-Datalog, which also keeps the salient features of p-Datalog.
- We provide a reduction of RQDL descriptions into p-Datalog augmented with *function symbols*. The reduction has important practical and theoretical value. From a practical point of view, it reduces the CBR problem for RQDL to the CBR problem for p-Datalog, thus giving a complete algorithm that is applicable to all RQDL descriptions. (The algorithm presented in [PGH96] only works for certain classes of RQDL descriptions.) From a theoretical point of view, it clarifies the difference in expressive power between RQDL and p-Datalog.

The next section introduces the p-Datalog description language. Section 3 describes the algorithm run by the wrappers. Section 4 describes a CBR algorithm run by the mediators. Section 5 discusses expressive power issues. Section 6 introduces RQDL. Section 7 describes the reduction of RQDL to p-Datalog with function symbols and Section 8 describes the wrapper and mediator algorithms for RQDL. Section 9 discusses the related work. Section 10 gives conclusions and future work.

2 The p-Datalog Source Description Language

It is well known that the most popular real-life query languages, like SPJ queries [AHV95] and Web-based query forms are equivalent to conjunctive queries. A Datalog program is a natural encoding of conjunctive queries: it "represents" all its expansions. First, we describe informally a Datalog-based source description language and demonstrate it with examples. A formal definition follows in the next subsection.

In the simple case of weak information sources, the source can be described using a set of parameterized queries. Parameters, called *tokens* in this paper, specify that some constant is expected in some fixed position in the query [PGGMU95, LRU96, LRO96]. For example, query

forms found in Web sites expect constants in some of their fields [LRO96]. Without loss of generality, we assume the existence of a designated predicate ans that is the head of all the parametrized queries of the description.

Example 2.1 Consider a bibliographic information source, that provides information about books. This source exports a predicate $books(isbn, author, title, publisher, year, pages)$. The source also exports "indexes," $author_index(author_name, isbn)$, $publisher_index(publisher, isbn)$ and $title_index(title_word, isbn)$. Conceptually, the tuple (X, Y) is in $author_index$ if the string X resembles the actual name of an author and Y is the ISBN of a book by that author. Similarly, (X, Y) is in $title_index$ if X is a word of the actual title and Y is the ISBN of a book with word X in the title. The following parameterized queries describe the wrapper that answers queries specifying an author, a title or a publisher.

$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), author_index(\$c, Id)$$
$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), title_index(\$c, Id)$$
$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), publisher_index(\$c, Id)$$

where \$c denotes a *token*. The query

$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), author_index(\texttt{Smith}, Id)$$

can be answered by that source, because it is derived by the first parameterized query by replacing $\$c$ by the constant `Smith`. □

In the previous example, the source is described by parameterized conjunctive queries. Note that if, for instance, the source accepts queries where values for any combination of the three indexes are specified, we would have to write $2^3 = 8$ parameterized conjunctive queries. The next example uses IDB predicates (i.e., predicates that are defined using source predicates and other IDB predicates) to describe the abilities of such a source more succinctly. Finally, example 2.3 uses recursive rules to describe a source that accepts an infinite set of query patterns.

Example 2.2 Consider the bibliographical source of the previous example. Assume that the source can answer queries that specify any combination of the three indexes. The p-Datalog program that describes this source is the following:

$$\begin{array}{lll} ans(Id, Aut, Titl, Pub, Yr, Pg) & \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), & \\ & \quad ind_1(Id), ind_2(Id), ind_3(Id) & (1) \\ ind_1(Id) & \leftarrow title_index(\$c, Id) & \\ ind_1(Id) & \leftarrow \epsilon & (2) \\ ind_2(Id) & \leftarrow author_index(\$c, Id) & (3) \\ ind_2(Id) & \leftarrow \epsilon & \\ ind_3(Id) & \leftarrow publisher_index(\$c, Id) & \\ ind_3(Id) & \leftarrow \epsilon & (4) \end{array}$$

ϵ denotes an empty body, *i.e.*, an ϵ–rule has an empty expansion. Notice that ϵ–rules are unsafe [Ull89]. In general, p-Datalog rules can be unsafe but that is not a problem under our semantics. Note also that the number of rules is only polynomial in the number of the available indexes, whereas the number of possible expansions is exponential.

The query

$$ans(Id, Aut, Titl, Pub, Yr, Pg) \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), author_index(\texttt{Smith}, Id)$$

can be answered by that source, because it is derived by expanding rule 1 using rules 2, 3 and 4, and by replacing $\$c$ by the constant `Smith`. We can easily modify the description to require that *at least* one index is used. □

In general, a p-Datalog program describes all the queries that are expansions of an *ans*-rule of the program. In particular, p-Datalog rules that have the *ans* predicate in the head can be expanded into a possibly infinite set of conjunctive queries. Among the expansions generated, some will only refer to source predicates[2]. We call these expansions *terminal expansions*. A p-Datalog program can have unsafe terminal expansions. We say that the p-Datalog program *describes* the set of conjunctive queries that are its safe terminal expansions. (see formal definitions in the next subsection).

Example 2.3 Consider again the bibliographical source of Example 2.1. Assume that there is an abstract index $abstract_index(abstract_word, Id)$ that indexes books based on words contained in their abstracts. Consider a source that accepts queries on books given one or more words from their abstracts. The following p-Datalog program can be used to describe this source.

$$
\begin{array}{ll}
ans(Id, Aut, Titl, Pub, Yr, Pg) & \leftarrow books(Id, Aut, Titl, Pub, Yr, Pg), ind(Id) \\
ind(Id) & \leftarrow abstract_index(\$c, Id) \\
ind(Id) & \leftarrow ind(Id), abstract_index(\$c, Id)
\end{array}
$$

□

As another example of a recursive source description, we can think of a transportation company, such as FedEx, that has an information source capable of answering queries about flights. Assume that the source can answer whether there exists a flight between cities A and B that makes n stops. We can model such a source with a p-Datalog program.

2.1 Formal description of p-Datalog.

We assume familiarity with Datalog, e.g. [Ull89, AHV95]. Besides the constant and variable sorts, we use a third disjoint set of symbols, the set of token variables or *tokens*.

Definition: A *parametrized Datalog rule* or *p-Datalog* rule is an expression of the form $p(u) \leftarrow p_1(u_1), \ldots, p_n(u_n)$ where $p, p_1, p_2, \ldots, p_n$ are relation names, and $u, u_1, u_2, \ldots, u_n$ tuples of constants, variables and tokens of appropriate arities. A *p-Datalog program* is a finite set of p-Datalog rules. □

Tokens are variables that have to be instantiated to form a query. We now formalize the semantics of p-Datalog as a source description language.

Definition: Let P be a p-Datalog program with a particular IDB predicate *ans*. The set of *expansions* $\mathcal{E}_P$ of P is the smallest set of rules such that:

- each rule of P that has *ans* as the head predicate is in $\mathcal{E}_P$;
- if r_1: $p \leftarrow q_1, \ldots, q_n$ is in $\mathcal{E}_P$, r_2: $r \leftarrow s_1, \ldots, s_m$ is in P (assume their variables and tokens are renamed, so that they don't have variables or tokens in common) and a substitution θ is the most general unifier of some q_i and r then the resolvent

$$\theta p \leftarrow \theta q_1, \ldots \theta q_{i-1}, \theta s_1, \ldots, \theta s_m, \theta q_{i+1}, \ldots, q_n$$

  of r_1 with r_2 using θ is in $\mathcal{E}_P$.

[2]We stated that source predicates are the EDB predicates of our descriptions.

The set of *terminal expansions* $\mathcal{T}_P$ of P is the subset of all expansions $e \in \mathcal{E}_P$ containing only EDB predicates in the body. The set of queries *described by* P is the set of all rules $\rho(r)$, where $r \in \mathcal{T}_P$ and ρ assigns arbitrary constants to all tokens in r. The set of queries *expressible* by P is the set of all queries that are equivalent to some query described by P. □

The above definitions can easily be extended to accommodate more than one "designated" predicates (like *ans*). Unification extends to tokens in a straightforward manner: a token can be unified with another token, yielding a token. When unified to a variable, it also yields a token. When unified to a constant, it yields the constant.

In the context of the above description semantics, we will use the terms p-Datalog *program* and *description* interchangeably.

Informally, we observe that expansions are generated in a grammar-like fashion, by using Datalog rules as productions for their head predicates and treating IDB predicates as "nonterminals" [ASU87]. Resolution is a generalization of non-terminal expansion; rules of context-free grammars can simply be thought of as Datalog rules with 0 arguments.

Rectification: For deciding expressibility as well as for solving the CBR problem the following rectified form of p-Datalog rules simplifies the algorithms. We assume the following conditions are satisfied:

- No variable appears twice in subgoals of the query body. Instead, multiple occurrences of the same variable are handled by using distinct variables and making equalities explicit with the use of the equality predicate *equal*.
- No variable appears twice in the head of the query. Again, equalities are made explicit with use of the predicate *equal*.
- No constants or tokens appear among the ordinary[3] subgoals. Instead, every constant c or token $\$c$ is replaced by a unique variable C, and an equality subgoal $equal(C, c)$ or $equal(C, \$c)$ is added to equate the variable to the constant.
- No variables appear only in an *equal* subgoal of a query.

Example 2.4 Consider the query

$$ans(X, X, Z) \leftarrow r(X, Y, Z), p(a, Y) \quad (1)$$

which contains a join between the second columns of r and p, a selection on the first column of p, and the same variable in two columns of *ans*. Its rectified equivalent is

$$ans(X_1, X, Z) \leftarrow r(X, Y, Z), p(A, Y_1), , equal(X, X_1), equal(Y, Y_1), equal(A, a) \quad (2)$$

□ Notice that we treat the *equal* subgoal not as a built-in predicate, but as a source predicate. We call rules that obey these conditions *rectified* rules and the process that transforms any rule to a rectified rule *rectification*. We call the inverse procedure (that would give us rule 1 from rule 2) *de-rectification*.

In the next two sections we provide algorithms for deciding whether a query is expressible by a description and for solving the CBR problem.

[3]We refer to the EDB and IDB relations and their facts as *ordinary*, to distinguish them from facts of the *equal* relation.

3 Deciding query expressibility with p-Datalog descriptions

In this section we present an algorithm for query expressibility of p-Datalog descriptions. In doing that, we develop the techniques that will allow us in the next section to give an elegant and improved solution to the problem of answering queries using an infinite set of views [LRO96].

Our algorithm is an extension of the classic algorithm for deciding query containment in a Datalog program that appears in [RSUV89] (also see [Ull89]). The algorithm tries to identify one expansion of the p-Datalog program that is equivalent to our query. We next illustrate the workings of the algorithm with an example.

Example 3.1 Let us revisit the bibliographic source of previous examples. Assume that the source contains a table $books(Id, Author, Publisher)$, a word index on titles, $title_index(title_word, Id)$[4] and an author index $au_index(au_name, Id)$. Also assume that the query capabilities of the source are described by the following p-Datalog program:

$$\begin{array}{ll} ans(A,P) & \leftarrow books(Id,A,P), ind_1(Id_1), ind_2(Id_2), equal(Id,Id_1), equal(Id,Id_2) \\ ind_1(Id) & \leftarrow title_index(V,Id), equal(\$c,V) \\ ind_1(Id) & \leftarrow \epsilon \\ ind_2(Id) & \leftarrow au_index(V,Id), equal(V,\$c) \\ ind_1(Id) & \leftarrow \epsilon \end{array}$$

Let us consider the query Q'

$$ans(X,Y) \leftarrow books(Id,X,Y), title_index(Id,\texttt{Zen}), au_index(Id,\texttt{Smith})$$

First we produce its rectified equivalent

$$\begin{array}{l} Q' : ans(X,Y) \leftarrow books(Id,X,Y), title_index(Id_1,V_1), au_index(Id_2,V_2), equal(V_1,\texttt{Zen}), \\ \qquad\qquad\qquad equal(V_2,\texttt{Smith}), equal(Id,Id_1), equal(Id,Id_2) \end{array}$$

Apparently the above query is expressible by the description. Intuitively, our algorithm discovers expressibility by "matching" the Datalog program rules with the subgoals. In particular, the "matching" is done as follows: first we create a DB containing a "frozen fact" for every subgoal of the query. Frozen facts are derived by turning the variables into unique constants which will be denoted with a bar.

Moreover, we want to capture all the information carried by *equal* subgoals into the DB. If, for example, subgoals $equal(X,Y), equal(X,Z)$ exist in the query, we will generate "frozen" facts for all implicit equalities as well, *i.e.*, $equal(Y,X), equal(Y,Z)$ etc. In the interests of space and clarity, we will write $equal(X,Y,Z)$ to mean that all the previously mentioned facts are in the DB. The DB for our running example is then

$$\begin{array}{l} books(\bar{id},\bar{x},\bar{y}), title_index(\bar{id}_1,\bar{v}_1), au_index(\bar{id}_2,\bar{v}_2), equal(\bar{id},\bar{id}_1,\bar{id}_2), \\ equal(\bar{v}_1,\texttt{Zen}), equal(\bar{v}_2,\texttt{Smith}) \end{array}$$

We then evaluate the Datalog program on the DB, deriving more facts for the IDB's. In addition, we keep track of the set of frozen facts, called *supporting set*, that we used for deriving each fact. Here is the set of facts and supporting sets derived by a particular evaluation of the Datalog program.

[4] Id denotes ISBN

$$
\begin{array}{ll}
< ind_1(Id), & \{\} > \\
< ind_2(Id), & \{\} > \\
(1) ::< ans(\bar{x}, \bar{y}), & \{books(\bar{id}, \bar{x}, \bar{y}), equal(\bar{id}, \bar{id})\} > \\
< ind_1(\bar{id}_1), & \{title_index(\bar{id}_1, \bar{v}_1), equal(\bar{v}_1, \texttt{Zen})\} > \\
< ind_2(\bar{id}_2), & \{au_index(\bar{id}_2, \bar{v}_2), equal(\bar{v}_2, \texttt{Smith})\} > \\
(2) ::< ans(\bar{x}, \bar{y}), & \{books(\bar{id}, \bar{x}, \bar{y}), title_index(\bar{id}_1, \bar{v}_1), equal(\bar{v}_1, \texttt{Zen}), \\
& au_index(\bar{id}_2, \bar{v}_2), equal(\bar{v}_2, \texttt{Smith}), equal(\bar{id}, \bar{id}_1, \bar{id}_2)\} >
\end{array}
$$

Every *ans* fact that is identical to the frozen head of the client query "corresponds" to a query that contains the client query. Furthermore, we can derive the containing query from the <fact, supporting set> pair by translating "frozen" facts back into subgoals. In our running example, the two containing queries[5] correspond to (1) and (2). If the supporting set is identical to the DB that we started with (modulo redundant equality subgoals) then the "corresponding" query is equivalent to the client query. Indeed, the "corresponding" query to (2) is

$$ans(X, Y) \leftarrow books(Id, X, Y), title_index(Id, \texttt{Zen}), au_index(Id, \texttt{Smith})$$

which is equivalent (actually identical) to our given query. □

The algorithm starts by mapping the subgoals of the given query into "frozen" facts, such that every variable maps to a unique constant, thus creating the *canonical database* [RSUV89, Ull89] of the query, and then evaluates the p-Datalog program on it, trying to produce the "frozen" head of the query. Moreover, it keeps track of the different ways to produce the same fact; that is achieved by "annotating" each produced fact f with its *supporting* facts, *i.e.*, the facts of the canonical DB that were used in that derivation of f.

We next formalize the notion of the canonical database. A formal definition of supporting facts follows.

Definition: Let $Q : H \leftarrow G_1, \ldots, G_k, \ldots, E_1, \ldots, E_m$ be a rectified conjunctive query, where $G_1, \ldots, G_k$ are the ordinary subgoals and $E_1, \ldots, E_m$ are the equality subgoals. Select a mapping τ that assigns to every variable X of Q a unique "frozen" constant $\tau(X) = \bar{x}$ and is the identity mapping on constants and predicate names. This way we construct k "frozen" ordinary facts: $\tau(G_1), \ldots, \tau(G_k)$. We also construct m "frozen" facts of the EDB predicate *equal*: $\tau(E_1), \ldots, \tau(E_k)$. These m facts constitute an instance of the *equal* relation. We create additional *equal* facts so that we get the smallest set of *equal* facts that includes this instance and is an equivalence relation. All the constructed facts constitute the canonical DB of query Q. □

Notice that this DB contains two "kinds" of constants: "regular" constants and frozen constants.

Example 3.2 Consider the rectified query:

$$ans(Y) \leftarrow p(X, X_1), q(X_2, Y, Z), equal(X, X_1), equal(X_1, X_2), equal(X, X_3), equal(Z, c)$$

The canonical DB produced by this query is

$$
\begin{array}{l}
p(\bar{x}, \bar{x}_1), q(\bar{x}_2, \bar{y}, \bar{z}), equal(\bar{x}, \bar{x}_1), equal(\bar{x}_1, \bar{x}_2), equal(\bar{z}, c), equal(\bar{x}, \bar{x}), equal(\bar{x}_1, \bar{x}_1), \\
equal(\bar{x}_2, \bar{x}_2), equal(\bar{x}, \bar{x}_2), equal(\bar{x}_2, \bar{x}), equal(\bar{x}_1, \bar{x}), equal(\bar{x}_2, \bar{x}_1), equal(c, \bar{z}), \\
equal(\bar{z}, \bar{z}), equal(c, c)
\end{array}
$$

□

[5]The algorithm uses pruning to eliminate (1) from the output.

Shorthand notation: Before we proceed, let us formalize the shorthand notation introduced in Example 3.1. It is obvious that if the *equal* facts form an equivalence relation, the constants and frozen constants appearing in *equal* facts are divided in equivalence classes.

Let us look at the canonical DB of some query Q. If variables $X_1, \ldots, X_k$ appearing in the canonical DB belong to the same equivalence class, we replace all *equal* facts involving $X_1, \ldots, X_k$ by $equal(X_1, \ldots, X_k)$. For example, $equal(X_1, X_2, X_3)$ "stands for" all $equal(X_i, X_j), 1 \leq i, j \leq 3$.

The canonical DB produced by the query of Example 3.2 above can be written as

$$p(\bar{x}, \bar{x}_1), q(\bar{x}_2, \bar{y}, \bar{z}), equal(\bar{z}, c), equal(\bar{x}, \bar{x}_1, \bar{x}_2)$$

It is easy to see that

$$equal(Y_1, \ldots, Y_l) \text{ is a subset of } equal(X_1, \ldots, X_m) \text{ iff } \forall i \leq l, Y_i \in \{X_1, \ldots, X_k\}$$

Definition: Let h be an ordinary fact produced by an application of the p-Datalog rule

$$r : H \leftarrow G_1, \ldots, G_k, E_1, \ldots, E_m$$

of a p-Datalog description P on a canonical DB, and let μ be a mapping from the rule into the canonical DB such that $\mu(G_i), \mu(E_j) \in DB$ and $h = \mu(H)$. The set $\mathcal{S}_h$ of supporting facts of h, or *supporting set* of h, with respect to P, is the smallest set such that

- if G_i is an EDB subgoal, $\mu(G_i) \in \mathcal{S}_h$,
- if G_i is an IDB subgoal and $\mathcal{S}'$ is the set of supporting facts of $\mu(G_i)$, then $\mathcal{S}' \subseteq \mathcal{S}_h$,
- if E is the set of all $\mu(E_i) \in \mathcal{S}_h$, then the smallest set of equality facts that includes E and is an equivalence relation is included in $\mathcal{S}_h$.

The supporting set can be computed in time quadratic in the size of the rule. □

Let us notice that $\mathcal{S}_h$ is the set of leaves of a proof tree [Ull89] for h. We can further annotate the produced fact with the "id" of the rule used in its production, thus generating the whole proof tree for this fact.

Example 3.3 We can apply the rule

$$ans(X_1, Z_1) \leftarrow author(X_1, Z_1), publisher(Z_2, W), equal(Z_1, Z_2), equal(W, \$w)$$

on the following canonical DB

$$author(\bar{a}, \bar{b}), author(\bar{a}, \bar{a}), publisher(\bar{d}, \bar{f}), publisher(\bar{g}, \bar{h}), equal(\bar{b}, \bar{d}), equal(\bar{a}, \bar{g}),$$
$$equal(\bar{f}, \texttt{PrenticeHall})$$

to produce fact $ans(\bar{a}, \bar{b})$. The supporting set $\mathcal{S}$ is

$$\{author(\bar{a}, \bar{b}), publisher(\bar{d}, \bar{f}), equal(\bar{b}, \bar{d}), equal(\bar{f}, \texttt{PrenticeHall})\}$$

□

We next define the notions of *extended facts* and *extended canonical DB*:

Definition: An *extended fact* is a pair of the form $< h, \mathcal{S}_h >$, where h is a fact and $\mathcal{S}_h$ is the supporting set for h, with respect to some description P. Let Q be a rectified conjunctive query. The *extended canonical DB* of Q is a database of extended facts $< f, \{f\} >$, such that every f belongs in the canonical DB of Q. □

Referring to Example 3.3, the extended fact "associated" with our production of $ans(\bar{a},\bar{b})$ is

$$< ans(\bar{a},\bar{b}), \{author(\bar{a},\bar{b}), publisher(\bar{d},\bar{f}), equal(\bar{b},\bar{d}), equal(\bar{f},\texttt{PrenticeHall})\} >$$

We now introduce the notion of the *corresponding query* for a fact, that makes our intuition about the supporting set explicit.

Definition: Let $< h, \mathcal{S}_h >$ be an extended fact of the DB. Then, for every fact $g_i \in \mathcal{S}_h$, we can define a mapping ρ that is the identity on constants and predicate names and maps every frozen constant to the variable which it came from. It is easy to see that this mapping is well-formed. Moreover, it maps $\mathcal{S}_h$ into a query body and the fact h into a query head. The query $Q{:}\rho(h) \leftarrow \rho(g_1), \ldots, \rho(g_k)$ is called the *corresponding query* for extended fact $< h, \mathcal{S}_h >$. □

Intuitively, the corresponding query is an instantiated expansion of the rules of the description that can prove h and uses only source and equality predicates.

We finally define the notion of *minimal* containing expansions, following [PGGMU95].

Definition: A query Q_s is a containing query for query Q with respect to description program P, iff $Q \subseteq Q_s$ and Q_s is described by P. Moreover, Q_s is a *minimal* containing query iff there is no query Q'_s described by P such that $Q \subseteq Q'_s$ and $Q'_s \subset Q_s$. □

Intuitively, a minimal containing query is an expansion of the description program that most tightly contains the given query. Notice that there may be more than one minimal containing query for a given query and a given description program. Moreover, notice that a minimal containing query for Q contains the maximal number of (non-redundant) subgoals of Q among containing queries of Q.

Our algorithm produces the set of minimal containing queries, since these are the only expansions that could be equivalent to the given query. If that set is nonempty, then obviously there exists a containing query for Q with respect to P. Moreover, Q is expressible by P *iff* one of the minimal containing queries in the set is equivalent to Q.

The algorithm is presented in detail in Appendix A. We proceed to give results on its correctness and running time. Proofs are omitted because of space considerations. Before that, let us just demonstrate with an example why rectification is necessary.

Example 3.4 To illustrate why rectification is necessary in identifying the minimal containing queries, let us consider the query $ans(X) \leftarrow p(X,c)$ and the p-Datalog description[6] $ans(A) \leftarrow p(A,B)$. Evaluating the description on the canonical DB $\{p(\bar{x},c)\}$ (without rectification), would produce the extended fact $< ans(\bar{x}), \{p(\bar{x},c)\} >$.
The corresponding query is

$$ans(X) \leftarrow p(X,c)$$

which is not a correct minimal containing query, because it is not expressible (by Def. 2.1) by the given description. If on the other hand we use rectification, we get the canonical DB $\{p(\bar{x},\bar{y}), equal(\bar{y},c)\}$. Evaluating the description on it, we get the corresponding query

$$ans(X) \leftarrow p(X,Y)$$

which is indeed the minimal containing query for our given query with respect to our description.
□

Now we are ready to state some formal results about the algorithm.

[6]This is obviously the description of a source with a very simple query interface

Lemma 3.5 (*MiniMax*) Let Q be a query and P be a p-Datalog description. Let also h be the "frozen" head of the query.

- Q' is a containing query for Q with respect to P if and only if there exists some extended fact $< h, \mathcal{S}_h >$ such that the corresponding query to the extended fact is equivalent to Q'.
- If Q_h is the corresponding query to the fact h, then Q_h is a minimal containing query for Q if and only if $\mathcal{S}_h$ is *maximal*, *i.e.*, it contains the maximal number of frozen subgoals of Q.

Algorithm A.1 only generates extended facts with maximal supporting sets and it is exhaustive.

Corollary 3.6 The corresponding queries for the *ans* facts that are the output of Algorithm A.1 are the minimal containing queries of input query Q.

Therefore, we have can state the following:

Theorem 3.7 (Correctness) Algorithm A.1 terminates and produces the set of minimal containing queries of input query Q.
Now, we can decide whether Q is expressible by P:

Lemma 3.8 Q is expressible by P iff the set of supporting facts for some extended fact $< h, \mathcal{S}_h >$ of the frozen head h of Q is identical[7] to the canonical DB for Q.

Proof: See Appendix. □

The number of extended facts that can be generated per "real" fact is equal to the number of different supporting sets for the fact, *i.e.*, it is exponential in the size of the canonical DB. The number of facts is exponential in the size of the description, so we have the following:

Theorem 3.9 Algorithm A.1 produces an answer in time exponential to the size of the description and the size of the query.

Translation: Let us consider the case of a wrapper that receives a query. It is easy to see that we could extend Algorithm A.1 so that it annotates each fact not only with its supporting set, but also with its proof tree. The wrapper then can use the parse tree to perform the actual translation of the user query in source-specific queries and commands, by applying the translating actions that are associated with each rule of the description.

4 Answering Queries Using p-Datalog Descriptions

Mediators are faced with a different problem than wrappers: Given the descriptions for one or more wrappers, the mediator has to answer the user query by "issuing" only queries expressible by the wrapper descriptions. That is the Capabilities-Based Rewriting (CBR) problem [PGH96, HKWY96]. As we have said in previous sections, a source description defines the (possibly infinite) set of conjunctive queries answerable by the source. So, the CBR problem is equivalent to the problem of answering the user query using an infinite set of views [LRU96].

Our algorithm proceeds in two steps. The first step finds a finite set of expansions. The second step uses an algorithm for answering queries using views [LMSS95, Qia96] to combine some of these expansions to answer the query. The first step uses the Algorithm A.1 to generate a finite set of expansions (see Appendix A). We prove that if we can answer the query using any combination

[7] After de-rectification of both.

of expressible queries, then we can answer it using a combination of expansions in our finite set. In [LRU96], a solution is presented for the problem whose complexity is non-deterministic doubly exponential in the size of the query and the description. The solution is based on "signatures" for the expansions of the description, that divide the queries that are expressible by the description into equivalence classes. We will show that our solution is non-deterministic exponential in the size of the query and the description. Moreover, the proof of our solution is more intuitive and simpler.

Given a user query Q and a wrapper description P in p-Datalog, Algorithm A.1 produces all[8] the minimal containing queries of Q with respect to P. We can show that there is at most an exponential number of those:

Lemma 4.1 The output of Algorithm A.1 contains at worst an exponential number of minimal containing queries, whose length is at most linear to the size of the given user query.

Moreover, we can prove that these are the only queries expressible[9] by P that are "relevant" in answering Q.

Theorem 4.2 (**CBR**) Assume we have a query Q and a p-Datalog description P, and let $\{Q_i\}$ be the result of applying Algorithm A.1 on Q and P. There exists a rewriting Q' of Q, such that $Q' \equiv Q$, using any $\{Q_j | Q_j$ is expressible by $P\}$ *if and only if* there exists a rewriting Q'' , such that $Q'' \equiv Q$, using only $\{Q_i\}$.

Proof: See Appendix. □

The problem of finding an equivalent rewriting of a query using a finite number of views is known to be NP-complete in the size of the query and the view set [LMSS95] and there are known algorithms for solving it [LMSS95, Qia96]. Using the set $\{Q_i\}$ of minimal containing queries as input to one of these algorithms, we obtain a solution to the CBR problem for p-Datalog that is non-deterministic exponential, since $|\{Q_i\}|$ is exponential in the size of the p-Datalog description and the user query.

5 Expressive Power of p-Datalog

We have illustrated the use of p-Datalog programs as a source description language. In this section, we explore some limits of its description capabilities. Proofs for the results in this section are provided in Appendix C. It should be noted that although we focus here on the description of conjunctive queries, similar results hold when negation and disjunction are introduced.

Clearly, there are sets of conjunctive queries that cannot be described by any p-Datalog description. Moreover:

Lemma 5.1 There exist *recursive* sets of conjunctive queries that are not expressible by any p-Datalog description.

However, the practical question is whether there are recursive sets of conjunctive queries, that correspond to "real" sources, and cannot be expressed by p-Datalog programs. We show next that some common sources (intuitively the "powerful" ones) exhibit this behavior. Before we prove this result, we demonstrate the expressive abilities and limitations of p-Datalog.

Let us start with an observation: For every p-Datalog description program P, the arity of the result is exactly the arity of the *ans* predicate. This restriction is somewhat artificial, since we

[8] modulo variable renaming

[9] The corresponding queries Q_i, that are the output of Algorithm A.1, actually are *described by P*.

can define descriptions with more that one "answer" predicate. However, even in that case, a given program would still bound the arities of answers. Furthermore, a more serious bound is the number of variables that occur in any one of the rules of the program. We will see that this bound is imposing severe restrictions on the queries that can be expressed.

But first, if we bound the number of variables, we can show the following:

Theorem 5.2 Let k be some integer. Let $p_1, \ldots, p_m$ be the EDB predicates of a database. There exists a p-Datalog program P that describes all conjunctive queries with at most k variables[10] on this database.

As mentioned above, a fixed p-Datalog program bounds the arity of the results, but this bound is not the *only* cause of limitation. Even if we focus on arity-0 results, *i.e.*, queries that answer yes or no and do not provide data, p-Datalog is limited. The limitation is related to the number of variables. Let FO^k be the set of sentences of first order logic [AHV95] with at most k variables. Note that the same variable can be "reused" as much as wanted using quantification. The following relates the queries described by a p-Datalog program to formulas expressible in first-order logic with a bounded number of variables. It states that although one such query may use an arbitrary number of variables, with appropriate "reuse" only a bounded number of variables suffice.

Lemma 5.3 Let P be a Datalog program and k the maximum number of variables occurring in a rule of P. Then for each Q expressible by P, Q is equivalent to a query in FO^k (using only $\wedge$ and $\exists$).

The limitation on the number of variables of the program prohibits the description of the set of all conjunctive queries over a schema — a set that is supported by common powerful sources.

Theorem 5.4 Let the database schema $\mathbf{S}$ have a relation of arity at least two. For every p-Datalog description P over $\mathbf{S}$, there exists a boolean query Q over $\mathbf{S}$, such that Q is not expressible by P. (So, in particular, there is no p-Datalog description that could describe a source that can answer all conjunctive queries, even if we fix the arity of the answer.)
The theorem points out a rather serious limitation of p-Datalog descriptions.

6 The RQDL description Language

Given the limitations of p-Datalog for the description of powerful information sources, we are proposing the use of a more powerful query description language. RQDL (Relational Query Description Language) is a Datalog-based rule language used for the description of query capabilities. It was first proposed in [PGH96] and used for describing query capabilities of information sources. [PGH96] shows its advantages over Datalog when it is used for descriptions that are not schema specific, i.e., the description does not refer to specific relations or arities in the schema of the specific source. In this way the descriptions are more concise and they gracefully handle schema evolution.

In this paper we present a formal specification of *extended-RQDL*, which provably allows us to describe large sets of queries. For example, we can prove that the extended-RQDL (from now on, we will by default refer to the extended-RQDL as RQDL), unlike p-Datalog, can describe the set of all conjunctive queries. Furthermore, we reduce RQDL descriptions to terminating p-Datalog programs with function symbols. Consequently, the decision on whether a given conjunctive query is expressed by an RQDL description is reduced to deciding expressibility of the query by the resulting p-Datalog program.

Note, the reduction of RQDL to Datalog with function symbols is important because

[10] We disregard repeated variables in the head of the conjunctive queries, so we assume that the result predicate has arity at most k.

- It reduces the comparison between the expressive power of p-Datalog and RQDL to a comparison between Datalog and Datalog with function symbols.
- It reduces the decision procedure for expressibility to Algorithm A.1. That allows us to give a complete solution to the CBR problem for RQDL.

Subsections 6.1 and 6.2 demonstrate the use of RQDL for the description of source capabilities and define the syntax and semantics of RQDL. Section 7 describes the reduction of RQDL descriptions to p-Datalog programs with function symbols and Section 8 proceeds to give algorithms for query expressibility by RQDL description and for the CBR problem for RQDL descriptions.

6.1 Using RQDL for query description

To support schema independent descriptions, RQDL allows the use of *predicate tokens*[11] in place of the relation names. Furthermore, to allow tables of arbitrary arity and column names, RQDL provides special variables called *vector variables*, or simply vectors, that match with sets of relation attributes that appear in a query. Vectors can "carry" arbitrarily large sets of attributes. It is this property that eventually allows the description of large, interesting sets of conjunctive queries (like the set of all conjunctive queries).

Example 6.1 illustrates RQDL's ability to describe source capabilities without referring to a specific schema. Example 6.2 demonstrates an RQDL program that describes all conjunctive queries over any schema. Subsection 6.2 describes the formal syntax and semantics of RQDL. Before we go ahead with the examples, let us introduce some new notation.

Named Attributes in Conjunctive Queries: For notational convenience, we slightly modify the query syntax so that we can refer to the components of tuples by attribute names instead of column numbers. For example, assuming that the relation *book* has schema *book*(*title*, *isbn*), we will write *book* subgoals by explicitly mentioning the attribute names. So, instead of writing

$$ans() \leftarrow book(X, Z), equal(X, \texttt{DataMarts})$$

we will write

$$ans() \leftarrow book(title : X, isbn : Z), equal(X, \texttt{DataMarts})$$

The connection of this scheme to SQL syntax is evident. We will be using named attributes in the rest of this paper. Every predicate will then have a *set* of named attributes (and not a list of attributes).

Example 6.1 Consider a source that accepts queries that refer to exactly one relation and pose exactly one selection condition over the source schema.

$$ans() \leftarrow \$r(\vec{V}), \vec{V}[_A] \equiv X', equal(X', \$c)$$

The above RQDL description[12] describes, among others, the query

$$ans() \leftarrow books(title : X, isbn : Z), equal(X, \texttt{DataMarts})$$

because, intuitively, we can map $\$r$ to relation *books*, $\vec{V}$ to the set of attribute-variable pairs $\{title : X, isbn : Z\}$, X' to X, and $\$c$ to *DataMarts*. The *metapredicate* $\vec{V}[_A] \equiv X'$ declares that

[11] Predicate tokens belong to the same sort as tokens.

[12] Notice that both the RQDL descriptions and the queries are rectified.

the variable X' maps to one of the variables in the set of attr-variable pairs that $\vec{V}$ is mapped to, i.e., X' maps to one of the variables of the subgoal $\$r$. The variable $_A$ maps to the attribute name of the variable X' in $\vec{V}$. No condition is placed on $_A$ and hence X' can be either X or Z.

RQDL descriptions do not have to be completely schema independent. For example, let us assume that we can put a selection condition only on the title attribute of the relation. Then we modify the above RQDL description as follows:

$$ans() \leftarrow \$r(\vec{V}), \vec{V}[name] \equiv X', equal(X', \$c) \tag{3}$$

The replacement of $_A$ by *name* forces the selection condition to refer to the *name* attribute only. □

Next we present the RQDL description that describes all conjunctive queries over any schema.

Example 6.2

$$\begin{array}{lcl}
ans(\vec{V}_1) & \leftarrow & cond(\vec{V}), \vec{V}_1 \subseteq \vec{V} \\
cond(\vec{V}) & \leftarrow & \$p(\vec{V}_1), cond(\vec{V}_2), \vec{V} = \vec{V}_1 \cup \vec{V}_2 \\
cond(\vec{V}) & \leftarrow & \vec{V}[_P] \equiv X, equal(X, \$c), cond(\vec{V}) \\
cond(\vec{V}) & \leftarrow & \vec{V}[_P_1] \equiv X_1, \vec{V}[_P_2] \equiv X_2, equal(X_1, X_2), cond(\vec{V}) \\
cond(\vec{V}) & \leftarrow & \$p(\vec{V})
\end{array}$$

Given any rectified conjunctive query, the description above describes it. □

6.2 Formal Syntax and Semantics of RQDL

The full syntax of RQDL appears in Appendix D, Fig. 3. There are three *metapredicates*, and their argument list has to be of a specific type: We define

$$union(\vec{V}, \vec{V}_1, \vec{V}_2) \text{ to mean } \vec{V} = \vec{V}_1 \cup \vec{V}_2 \tag{4}$$

where $\vec{V}$ is a vector and $\vec{V}_1, \vec{V}_2$ can be vectors, or sets of attribute-variable pairs. $\vec{V}$ can only appear in the head of the rule (in addition to the *union* subgoal). We also define

$$set_item(\vec{V}, _P, X) \text{ to mean } \vec{V}[_P] \equiv X \tag{5}$$

and

$$set_item(\vec{V}, a, X) \text{ to mean } \vec{V}[a] \equiv X \tag{6}$$

which means that the variable of attribute named $_P$ (or a) in vector $\vec{V}$ is the same as X. a is a constant. $_P$ is an *attribute variable*, meaning it is a variable that can only bind to attribute names. X is a variable. We define

$$subset(\vec{V}, \vec{V}_1) \text{ to mean } \vec{V} \subseteq \vec{V}_1 \tag{7}$$

where $\vec{V}$ is a vector and $\vec{V}_1$ can be a vector or a set of attribute-variable pairs. $\vec{V}$ can only appear in the head of the rule (in addition to the *subset* subgoal). The intuition behind *subset* is that it allows us to do arbitrary projections.

In addition, the metapredicates must observe some binding pattern constraints. In particular, all vectors that appear in metapredicates must be safe as defined below:

- If a vector appears in an EDB or IDB subgoal then it is safe.
- If a vector $\vec{V}$ appears in a subgoal $union(\vec{V}, \vec{V_1}, \vec{V_2})$ and $\vec{V_1}$ and $\vec{V_2}$ are safe, then $\vec{V}$ is also safe.
- If a vector $\vec{V}$ appears in a subgoal $subset(\vec{V}, \vec{V_1})$ and $\vec{V_1}$ is safe, then $\vec{V}$ is also safe.

An RQDL description is a finite set of RQDL rules. The description semantics of RQDL are a generalization of the description semantics of p-Datalog, to account for the existence of vectors and metapredicates. We start by defining what is an *expansion* of an RQDL description.

Definition: Let P be an RQDL description with a particular IDB predicate ans. The set of *expansions* $\mathcal{E}_P$ of P is the smallest set of rules such that:

- each rule of P that has ans as the head predicate is in $\mathcal{E}_P$;
- if r_1: $p \leftarrow q_1, \ldots, q_n$ is in $\mathcal{E}_P$, r_2: $r \leftarrow s_1, \ldots, s_m$ is in P, and a substitution θ is the most general unifier of some q_i and r then the resolvent

$$\theta p \leftarrow \theta q_1, \ldots \theta q_{i-1}, \theta s_1, \ldots, \theta s_m, \theta q_{i+1}, \ldots, q_n$$

  of R_1 with R_2 using θ is in $\mathcal{E}_P$.

□

Unification: Unification extends to vectors in the following way:

1. a vector can be unified with another vector, yielding a vector;
2. a vector can unify with a set consisting of attribute-value or attribute-variable pairs, yielding that set; for example $p(\vec{V})$ can unify with $p(attr_1 : X, attr_2 : \texttt{John})$ yielding

$$p(attr_1 : X, attr_2 : \texttt{John})$$

In the presence of *set_item* subgoals referring to the unified vector, unification in the second case succeeds if the subgoal is made true.

Following the definition of description semantics of Section 2, we now define the description semantics of RQDL.

Definition: The set of *terminal expansions* $\mathcal{T}_P$ of P is the subset of all expansions $e \in \mathcal{E}_P$ containing only EDB predicates or predicate tokens in the body.

The set of *instantiated terminal expansions* $\mathcal{I}_\mathcal{P}$ of RQDL description P is the set of all (rectified) conjunctive queries $\tau(r)$, where r belongs to the set of terminal expansions of P and τ is a mapping of the RQDL rule r to a conjunctive query, that:

1. maps every token $\$c$ to a constant c. (Note, we consider relation names to be of constant type.)
2. maps every vector $\vec{V}$ to a list of attribute-variable pairs $[(a_1, X_1), \ldots, (a_n, X_n)]$ such that
   (a) after we replace every predicate subgoal $p(\vec{V})$ with $p(a_1 : X_1, \ldots, a_n : X_n)$ no variable appears in more than one predicate subgoals,
   (b) for every subgoal of the form $union(\vec{V}, \vec{V_1}, \vec{V_2})$, $\tau(\vec{V}) = \text{concat}(\tau(\vec{V_1}), \tau(\vec{V_2}))$,

(c) for every subgoal of the form $set_item(\vec{V}, a, X)$, $\tau(\vec{V})$ includes a pair (a, X),

(d) for every subgoal of the form $set_item(\vec{V}, _P, X)$, $\tau(\vec{V})$ includes a pair (a, X), for some a,

(e) for every subgoal of the form $subset(\vec{V}, \vec{V_1})$, τ maps $\vec{V}$ to a subset of $\tau(\vec{V_1})$.

3. and drops all metapredicate subgoals.

The set of *described* queries of an RQDL description P with "designated" predicate ans (when ans is understood), is the set of safe instantiated terminal expansions of P. □

If Q is a conjunctive query with head predicate ans and P is an RQDL description, we say that *Q is expressible by P*, if there exists Q' described by P, such that $Q \equiv Q'$.

Next we will discuss an efficient algorithm for deciding whether a query is expressible by a description. The algorithm is based on a reduction of both the query and the description into a simple standard schema which facilitates reasoning about relations and attribute names.

7 Reducing RQDL to p-Datalog with function symbols

Deciding whether a query is expressible by an RQDL description requires "matching" the RQDL description with the query. This is a challenging problem because vectors have to match with non-atomic entities, i.e., sets of variables, hence making matching much harder.

In [PGH96], where that problem is also identified, a brute force approach is used, where vectors actually match with sets during the derivation. Unfortunately, this approach soon leads to complicated problems which forced [PGH96] to restrict the applicability of matching algorithms to a subset of RQDL descriptions. A particularly tough problem is the existence of unsafe rules that have vectors in the head. A brute force approach may then derive extended facts where a vector is "half-specified", i.e., we know some of the attr-variable pairs that it should contain but not all of them. Note that [PGH96] is not applicable to RQDL descriptions that may exhibit this behavior.

In this section we present an algorithm that avoids these problems by reducing the problem of query expressibility by RQDL descriptions to the problem of query expressibility by p-Datalog *with function symbols*, i.e., we reduce the RQDL description into a corresponding description in p-Datalog with function symbols. The reduction is based on the idea that every database DB can be reduced into an equivalent database DB' such that the attribute names and relation names of DB appear in the data (and not the schema) of DB'. We call DB' a *standard schema database*. We then rewrite the query so that it refers to the schema of DB' (*i.e.*, the standard schema) and we also rewrite the description into a p-Datalog description *with function symbols* which refers to the standard schema as well.

Subsection 7.1 presents the conceptual reduction of a database into a standard schema database. Subection 7.2 presents the rewriting of queries and Subsection 7.3 presents the rewriting of RQDL descriptions. Each of the subsections starts with one or two examples and continues with a formal definition of the reduction which can be skipped at the first reading.

7.1 Reduction of any database to standard schema database

In order to reason with the relation names and attribute names of the queries, we conceptually reduce the original database into a standard schema database where the relation names and the attribute names appear as data and hence can be manipulated without the need of higher order

syntax. First we present a reduction example and then we formally define the reduction of a database into its standard schema counterpart.

Example 7.1 Consider the following database DB with schema $b(author, isbn)$ and $f(subject, isbn)$.

b	
author	*isbn*
Smith	123
Jones	345

f	
subject	*isbn*
Logic	123
Theology	345

The corresponding standard schema database DB' consists of two relations $tuple(table\ name, tuple\ id)$ and $attr(tuple\ id, attr\ name, value)$ which are common to all standard schema databases. In the running example DB' is

tuple	
table name	*tuple id*
b	b(Smith,123)
b	b(Jones,345)
f	f(Logic,123)
f	f(Theology,345)

attr		
tuple id	*attr name*	*value*
b(Smith,123)	author	Smith
b(Smith,123)	isbn	123
b(Jones,345)	author	Jones
b(Jones,345)	isbn	345
f(Logic,123)	subject	Logic
f(Logic,123)	isbn	123
f(Theology,345)	subject	Theology
f(Theology,345)	isbn	345

Notice above how we invented one tuple id for each tuple of the original database. □

Definition: Given a database DB, we say that the standard schema database corresponding to DB is the smallest database DB' such that

1. its schema is $tuple(table\ name, tuple\ id)$ and $attr(tuple\ id, attr\ name, value)$, and
2. for every tuple $t(a_1 : v_1, \ldots, a_n : v_n)$ in DB, there is a tuple $tuple(t, t(v_1, \ldots, v_n))$ in DB' and for every attribute $a_i, i = 1, \ldots, n$ there is a tuple $attr(t(v_1, \ldots, v_n), a_i, v_i)$ in DB'.

□

7.2 Reduction of queries to standard schema queries

The RQDL expressibility algorithm first reduces a given conjunctive query Q over some database DB into a corresponding query Q' over the standard schema database DB'. The reduction is correct in the following sense: the result of asking query Q' on DB' is equivalent, modulo tuple-id naming, to the reduction into standard schema of the result of Q on DB.

To illustrate the query reduction, let us consider a couple of examples. We first consider a boolean query Q over the schema of Example 7.1.

$$ans() \leftarrow b(author : X, isbn : S_1), f(subject : A, isbn : S_2), equal(S_1, S_2), equal(A, \texttt{Theology})$$

Query Q is reduced into the following query Q':

$$tuple(ans, ans()) \leftarrow tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), equal(S_1, S_2), attr(F, subject, A), equal(A, \texttt{Theology})$$

Notice that for every ordinary subgoal we introduce a *tuple* subgoal and invent a tuple id. For every attribute we introduce an *attr* subgoal. The tuple id for the result relation *ans* is simply *ans()* because the result relation has no attributes. When the query head has attributes, a single conjunctive query is reduced to a non-recursive Datalog program. For example, consider the following query that returns the authors and ISBNs of books if their subject is `Theology`.

$$ans(author : X, isbn : S_1) \leftarrow b(author : X, isbn : S_1), f(subject : A, isbn : S_2), equal(S_1, S_2), equal(A, \texttt{Theology})$$

This query is reduced to the following program Q' where the first rule defines the *tuple* part of the standard schema answer and the last two rules describe the *attr* part.

$$\begin{array}{lcl} tuple(ans, ans(X, S_1)) & \leftarrow & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), \\ & & equal(S_1, S_2), attr(B, author, X), attr(F, subject, A), \\ & & equal(A, \texttt{Theology}) \\ attr(ans(X, S_1), author, X) & \leftarrow & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), \\ & & equal(S_1, S_2), attr(B, author, X), attr(F, subject, A), \\ & & equal(A, \texttt{Theology}) \\ attr(ans(X, S_1), isbn, S_1) & \leftarrow & tuple(b, B), tuple(f, F), attr(B, isbn, S_1), attr(F, isbn, S_2), \\ & & equal(S_1, S_2), attr(B, author, X), attr(F, subject, A), \\ & & equal(A, \texttt{Theology}) \end{array}$$

In general, the reduction is accomplished by the following procedure:

Procedure 7.2 (Reduction) If Q's head is $ans(a_1 : V_1, \ldots, a_n : V_n)$, generate a program with $n+1$ rules such that

1. one rule has head $tuple(ans, ans(V_1, \ldots, V_n))$,
2. for every attribute $a_i, i = 1, \ldots, n$ there is a rule with head $attr(ans(V_1, \ldots, V_n), a_i, V_i)$, and
3. all rules have the same body which is constructed by the following steps:
   (a) For every subgoal of Q of the form $r(a_1 : X_1, \ldots, a_m : X_m)$, invent and associate to it a unique variable T. The variables such as T bind to tuple id's of the standard schema database and hence we call them *tuple id variables.*,
   (b) include in the standard schema query body the subgoal $tuple(r, T)$,
   (c) and for every attribute $a_i, i = 1, \ldots, m$ include in the standard schema query the subgoal $attr(T, a_i, X_i)$.
   (d) Add to that body all equality subgoals of the original query.

□

X_i can be a variable, a token or a constant. It is easy to see that under a few obvious constraints there exists the inverse reduction.

Next we show how we reduce RQDL descriptions into p-Datalog descriptions over standard schema databases.

7.3 Reduction of RQDL programs to Datalog programs operating on standard schema

In the previous sections we showed how schema information, i.e., relation and attribute names, becomes data in standard schema databases. Based on this idea, we will reduce RQDL descriptions into p-Datalog descriptions that do not use higher order features such as metapredicates and vectors. In particular, we "reduce" vectors to tuple identifiers. Intuitively, if a vector matches with the arguments of a subgoal, then the tuple identifier associated with this subgoal is enough for finding all the attr-variable pairs that the vector will match to. Otherwise, if a vector $\vec{V}$ is the result of a union of two other vectors $\vec{V_1}$ and $\vec{V_2}$, then we associate with it a new *constructed* tuple id, the function $u(T_1, T_2)$ where T_1 and T_2 are the tuple id's that correspond to $\vec{V_1}$ and $\vec{V_2}$. As we will see later, the reduction carefully produces a program which terminates despite the use of the u function.

Example 7.3 The description of Example 6.2 describes all boolean conjunctive queries. It reduces into the following p-Datalog description (with function symbols):

$$\begin{array}{ll} tuple(ans, ans(T)) \leftarrow cond(T) & (1) \\ cond(T) \leftarrow tuple(\$p, T_1), cond(T_2), valid(T, T_1, T_2) & \\ cond(T) \leftarrow attr(T, _P, X), equal(X, \$c), cond(T) & \\ cond(T) \leftarrow attr(T_1, _P_1, X_1), attr(T_2, _P_2, X_2), equal(X_1, X_2), cond(T) & \\ cond(T) \leftarrow tuple(\$p, T) & \end{array}$$

and $subset_flag_{(1)} = 1$.

The reduction of each rule is independent from the reduction of other rules. In the reduction of the first rule, notice that the vector variable $\vec{V}$ has been "replaced" by the variable T which matches with a tuple id. In the second rule, notice that we reduced $\vec{V}$ to T, which is "produced" by the predicate *valid*, given T_1 and T_2. *valid* is a predicate defined by the rules of Fig. 4 (see Appendix D), which have to be included in all reduced p-Datalog descriptions. *valid* constructs a new tuple id of a restricted form, that has "associated" with it all the attributes associated with T_1 or T_2. The role of *valid* is to "simulate" the union that it replaces, by not allowing generation of arbitrary u terms[13].

The intuition behind *valid* terms is the following: tuple id variables bind either to tuple ids or to constructed tuple ids, *i.e.*, u terms "built" from tuple ids. Assuming that there is a total order for the tuple ids of the standard schema database, $valid(T, T_1, T_2)$ creates a u term in which all tuple ids appear in sorted order, and none are repeated. In particular, $valid(T, u(t_2, u(t_3, t_4)), u(t_3, t_5))$ will bind T to $u(t_2, u(t_3, u(t_4, t_5)))$.

Finally, the description has to include the "standard" rules of Fig. 2, that make sure that all attributes of tuple with ids T_1 and T_2 are also attributes of tuples with id T, constructed from T_1, T_2.

$$\begin{array}{l} attr(T, _A, X) \leftarrow attr(T_1, _A, X), valid(T, T_1, T_2) \\ attr(T, _A, X) \leftarrow attr(T_2, _A, X), valid(T, T_1, T_2) \end{array}$$

Figure 2: Default rules for generation of *attr* tuples

In the reduction of the third rule of the description, notice that the metapredicate $\vec{V}[_P] \equiv X$, *i.e.*, $set_item(V, _P, X)$, is reduced to the predicate $attr(T, _P, X)$. □

[13]The analogy is that union includes attr-var pairs only once.

Formally, an RQDL description P is reduced to a p-Datalog description P' by the following steps:

1. Include in P' the rules of Figures 2 and 4.
2. Reduce each rule r of the description to p-Datalog with functions as follows:
   (a) Reduce predicates that do not involve vectors as described in subsection 7.2.
   (b) For each subgoal of the form $r(\vec{V})$ include in the reduced rule a subgoal $tuple(r, T)$. T is the *reduction* of $\vec{V}$.
   (c) For each subgoal of the form $set_item(\vec{V}, a, X)$, where a is a variable or a constant, include in the reduced rule the subgoal $attr(T, a, X)$, where T is the reduction of $\vec{V}$.
   (d) For each subgoal of the form $union(\vec{V}, \vec{V_1}, \vec{V_2})$, replace in the reduced rule all instances of $\vec{V}$ with T and include the subgoal $valid(T, T_1, T_2)$, where T_1 and T_2 are the reductions of $\vec{V_1}$ and $\vec{V_2}$.
   (e) For each subgoal of the form $subset(\vec{V}_1, \vec{V})$, let T_1 be the reduction of $\vec{V}_1$ and T be the reduction of $\vec{V}$. Replace T_1 by T in the rule where *subset* appears, set the *subset flag* for the rule to 1 (see below) and drop the *subset* subgoal.
   (f) If the head is of the form $ans(\vec{V})$ then reduce it to $tuple(ans, T)$.
   (g) If the head is of the form ans(attr-var set) then follow Procedure 7.2 to generate all the p-Datalog rules that r reduces to.

The intuition behind the *subset_flag* of a rule is as follows: Assume the existence of a subgoal $subset(\vec{V}_1, \vec{V})$ in rule r. As we have said earlier, $\vec{V}_1$ must appear in the rule head, so let the head of r be $p(\vec{V}_1)$, and $\vec{V}$ must appear in an ordinary subgoal, say $q(\vec{V})$. The subset subgoal means that the RQDL rule r describes all conjunctive queries whose head attribute set is any projection of the attribute set of relation q. In the reduction, we replace T_1 (the reduction of $\vec{V}_1$) by T (the reduction of $\vec{V}$), saying effectively that the attribute set of p must be the same as the attribute set of q. That's why we set a flag, the *subset flag*, to make sure we also consider described those conjunctive queries that include projections on q.[14]

Theorem 7.4 Let P be an RQDL description and P' its reduction in p-Datalog with functions. Let also DB be a canonical standard schema database of a query Q. Then P' applied on DB terminates.

Crux: It suffices to see that the generation of u terms cannot fall into an infinite loop, since no tuple id present in the database can appear twice in any constructed tuple id. □

The next section explains the semantics of p-Datalog with functions, and shows how to solve the CBR problem for RQDL using the algorithms developed for p-Datalog in Sections 3 and 4.

[14]Another way to handle the subset metapredicate is by defining an ordering among constructed tuple ids, *i.e.*by defining what $T_i < T_j$ means if T_i, T_j are not atomic values. Then $subset(\vec{V}_1, \vec{V})$ would just be reduced to $T_1 < T$, where T_1, T are correspondingly the reductions of $\vec{V}_1$ and $\vec{V}$.

8 Expressibility and CBR with RQDL descriptions

Let us start by clarifying the semantics of p-Datalog with functions. We will denote p-Datalog with functions with *p-Datalog*^{f}.

Let us note that the transformation of a conjunctive query to refer to the standard schema, per subsection 7.2, results in a set of standard schema queries with the same body. We illustrate the notion of expressiblity in RQDL with an example. Notice that there are now two "designated" predicates, the predicates $tuple$ and $attr$.

Example 8.1 Consider the query Q: $ans(a:X) \leftarrow books(au:X, titl:Y)$ and the description

$$\begin{array}{lcl} ans(a:X) & \leftarrow & \$r(au:X, titl:Y) \\ ans(b:Y) & \leftarrow & \$r(au:X, titl:Y) \end{array}$$

The canonical DB is

$$tuple(books, t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0, t_1, t_2)$$

The reduction of the description (after rectification) is

$$\begin{array}{lcl} tuple(ans, ans(X)) & \leftarrow & tuple(\$r, T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T, T_1), equal(T, T_2) \\ attr(ans(X), a, X) & \leftarrow & tuple(\$r, T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T, T_1), equal(T, T_2) \\ tuple(ans, ans(Y)) & \leftarrow & tuple(\$r, T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T, T_1), equal(T, T_2) \\ attr(ans(Y), b, Y) & \leftarrow & tuple(\$r, T), attr(T_1, au, X), attr(T_2, titl, Y), equal(T, T_1), equal(T, T_2) \end{array}$$

Notice that we didn't include the rules of Figures 2 or 4 (*valid* rules) in the reduced description, since the original description didn't contain any metapredicates.

If we run the Algorithm A.1 on the canonical DB, the following extended facts are produced:

$$\begin{array}{ll} (1) < tuple(ans, ans(x)), & \{tuple(books, t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0, t_1, t_2)\} > \\ (2) < attr(ans(x), a, x), & \{tuple(books, t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0, t_1, t_2)\} > \\ < tuple(ans, ans(y)), & \{tuple(books, t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0, t_1, t_2)\} > \\ < attr(ans(y), b, y), & \{tuple(books, t_0), attr(t_1, au, x), attr(t_2, titl, y), equal(t_0, t_1, t_2)\} > \end{array}$$

The output of the algorithm includes the exact two conjunctive queries (the corresponding queries to the extended facts (1) and (2)) that are the reduction of Q. We therefore say that Q is expressible by our description. □

Before presenting the theorem that states the condition for RQDL expressibility, let us make the following important observations:

Let Q be an RQDL query and let $\{Q_i, i \leq n\}$ be the set of standard schema queries it reduces to. Let H_i be the heads of those queries. As we pointed out in Section 7.2, all Q_i have the same body. Moreover, H_1 is of the form $tuple(ans, T)$, where T is a term that denotes a tuple id, and $H_i, i \neq 1$ are of the form $attr(T, c_i, X_i)$ *for the same* T. We call T the *query id*.

Finally, let us observe that the exact "value" of tuple ids is not important: their use is to identify components (*i.e.*, attributes) of the same relation. Therefore, we say that a reduced query Q in p-Datalog^f is expressible by a reduced p-Datalog^f description P if and only if there exists Q' equivalent to Q *up to tuple-id naming* that is described by P.

Theorem 8.2 A query Q is expressible by an RQDL description P if there exists a maximal set $\{Q'_i, i \leq n\}$ of queries described by the reduced description P', where all Q'_i have the same id, such that $Q'_i \equiv Q_i$, $\forall i$.[15] Maximal means that $\{Q'_i\}$ includes *all* described queries with that same query id.

[15] If the subset flag is on, then it could be $\{Q'_i, i \leq m\}$ with $m \geq n$.

Because of Theorems 7.4 and 8.2, we have seen that we can use Algorithm A.1 to answer the expressibility question in RQDL. The idea is to generate all possible extended facts for *tuple* and *attr* and then, as in Section 3, check whether (i) all and only the necessary "frozen" *tuple* and *attr* facts are produced, and have the same id, and (ii) their corresponding queries are equivalent to the Q_i's. For the algorithm to work properly, we need to change the definition of the supporting set of a fact: due to the reduction introduced in Sections 7.2 and 7.3, there is an implicit "connection" between a fact $tuple(const_1, T)$ and facts $attr(T, const_2, X)$, *i.e.*, between the *tuple* fact and the *attribute* facts that are created by the reduction. We make that connection explicit by modifying the definition of supporting set as follows:

Definition: Let h be an ordinary fact produced by an application of the p-Datalog^f rule

$$r : H \leftarrow G_1, \ldots, G_k, E_1, \ldots, E_m$$

of a (reduced) p-Datalog^f description P on a canonical DB, and let μ be a mapping from the rule into the canonical DB such that $\mu(G_i), \mu(E_j) \in DB$ and $h = \mu(H)$. The set $\mathcal{S}_h$ of supporting facts of h, or *supporting set* of h, with respect to P, is the smallest set such that

- if G_i is an EDB subgoal, $\mu(G_i) \in \mathcal{S}_h$,
- if G_i is an IDB subgoal and $\mathcal{S}'$ is the set of supporting facts of $\mu(G_i)$, then $\mathcal{S}' \subseteq \mathcal{S}_h$,
- if $tuple(c, t) \in \mathcal{S}_h$ for some[16] c and t, then for all c', x, if $attr(t, c', x)$ in the canonical DB, then $attr(t, c', x) \in \mathcal{S}_h$,
- if $attr(t, c, x) \in \mathcal{S}_h$ for some t, c and x, then
  1. There exists a $tuple(c', t)$ in the canonical DB for some c'. That fact $tuple(c', t) \in \mathcal{S}_h$.
  2. For all c', x, if $attr(t, c', x)$ in the canonical DB, then $attr(t, c', x) \in \mathcal{S}_h$.
- if E is the set of all $\mu(E_i) \in \mathcal{S}_h$, then the smallest set of equality facts that includes E and is an equivalence relation is included in $\mathcal{S}_h$.

□

Let us now consider the following example.

Example 8.3 If our RQDL description is

$$ans(\vec{V}) \leftarrow p(\vec{V})$$

then the query $ans(name : X) \leftarrow p(name : X, age : Y)$ is *not* expressible by our description. The reduction of the description is

$$\begin{array}{lcl} tuple(ans, T) & \leftarrow & tuple(p, T) \\ attr(T, _A, X) & \leftarrow & attr(T_1, _A, X), valid(T, T_2, T_3), equal(T_1, T_2) \\ attr(T, _A, X) & \leftarrow & attr(T_1, _A, X), valid(T, T_2, T_3), equal(T_1, T_3) \end{array}$$

and the reduction of the query (*i.e.*, the set $\{Q_i\}$) is

$$\begin{array}{lcl} tuple(ans, ans(X)) & \leftarrow & tuple(p, T), attr(T, name, X), attr(T, age, Y) \\ attr(ans(X), name, X) & \leftarrow & tuple(p, T), attr(T, name, X), attr(T, age, Y) \end{array}$$

[16] c and t can be frozen or regular constants.

The canonical DB is then

$$tuple(p, t_0), attr(t_1, name, \bar{x}), attr(t_2, age, \bar{y}), equal(t_0, t_1, t_2)$$

The extended facts produced by Algorithm A.1 are

$$\begin{array}{lll} (1) & < tuple(ans, t_0), & \{tuple(p, t_0), attr(t_1, name, \bar{x}), attr(t_2, age, \bar{y}), equal(t_0, t_1, t_2)\} > \\ (2) & < valid(t_0, t_0, t_0) & \{tuple(p, t_0), attr(t_1, name, \bar{x}), attr(t_2, age, \bar{y}), equal(t_0, t_1, t_2)\} > \\ (3) & < attr(t_0, name, \bar{x}) & \{tuple(p, t_0), attr(t_1, name, \bar{x}), attr(t_2, age, \bar{y}), equal(t_0, t_1, t_2)\} > \\ (4) & < attr(t_0, age, \bar{x}) & \{tuple(p, t_0), attr(t_1, name, \bar{x}), attr(t_2, age, \bar{y}), equal(t_0, t_1, t_2)\} > \end{array}$$

Even though both standard schema queries of the reduction are expressible by our reduced description, the original query as pointed out is not expressible by the RQDL description. That is because the only maximal set of described queries produced (consisting of the corresponding queries for (1),(3) and (4)) is larger than the set of reduced queries. □

Let us consider a more complicated example.

Example 8.4 The following source can accept queries that perform a join between relation q with any other relation over any set of attributes.

$$\begin{array}{lll} ans(\vec{V}) & \leftarrow & cond(\vec{V}) \\ cond(\vec{V}) & \leftarrow & q(\vec{V}_1), union(\vec{V}, \vec{V}_1, \vec{V}_2), cond(\vec{V}_2) \\ cond(\vec{V}) & \leftarrow & set_item(\vec{V}, _P_1, X_1), set_item(\vec{V}, _P_2, X_2), equal(X_1, X_2), cond(\vec{V}) \\ cond(\vec{V}) & \leftarrow & \$r(\vec{V}) \end{array}$$

The reduction of the description, after rectification, is

$$\begin{array}{lll} tuple(ans, T) & \leftarrow & cond(T) \\ cond(T) & \leftarrow & tuple(q, T_1), cond(T_2), valid(T, T_3, T_4), equal(T_1, T_3), equal(T_2, T_4) \\ cond(T) & \leftarrow & attr(T, _P_1, X_1), attr(T_1, _P_2, X_2), equal(X_1, X_2), cond(T_2), \\ & & equal(T, T_1), equal(T, T_2) \\ cond(T) & \leftarrow & tuple(\$r, T) \\ attr(T, _A, X) & \leftarrow & attr(T_1, _A, X), valid(T, T_2, T_3), equal(T_1, T_2) \\ attr(T, _A, X) & \leftarrow & attr(T_1, _A, X), valid(T, T_2, T_3), equal(T_1, T_3) \end{array}$$

plus the rules in Fig. 4 (see Appendix D).
The user query submitted to the source is the following:

$$ans(name : X, lastname : X, age : Z) \leftarrow q(name : X, age : Z), s(lastname : X)$$

which produces the extended canonical DB

$$tuple(q, t_0), attr(t_1, name, \bar{x}), attr(t_2, age, \bar{z}), tuple(s, t_3), attr(t_4, lastname, \bar{x}_1), equal(t_0, t_1, t_2), \\ equal(t_3, t_4), equal(\bar{x}, \bar{x}_1)$$

The standard schema reduction of the user query is

$$\begin{array}{lll} tuple(ans, ans(X, Z)) & \leftarrow & tuple(q, Q), tuple(s, S), attr(Q_1, name, X), attr(Q_2, age, Z) \\ & & attr(S_1, lastname, X_1), equal(Q, Q_1, Q_2), equal(S, S_1), equal(X, X_1) \\ attr(ans(X, Z), name, X) & \leftarrow & tuple(q, Q), tuple(s, S), attr(Q_1, name, X), attr(Q_2, age, Z) \\ & & attr(S_1, lastname, X_1), equal(Q, Q_1, Q_2), equal(S, S_1), equal(X, X_1) \\ attr(ans(X, Z), lastname, X) & \leftarrow & tuple(q, Q), tuple(s, S), attr(Q_1, name, X), attr(Q_2, age, Z) \\ & & attr(S_1, lastname, X_1), equal(Q, Q_1, Q_2), equal(S, S_1), equal(X, X_1) \\ attr(ans(X, Z), age, Z) & \leftarrow & tuple(q, Q), tuple(s, S), attr(Q_1, name, X), attr(Q_2, age, Z) \\ & & attr(S_1, lastname, X_1), equal(Q, Q_1, Q_2), equal(S, S_1), equal(X, X_1) \end{array}$$

Running Algorithm A.1 on the canonical DB produces the following extended facts[17]:

$< valid(u(t_0,t_4),t_0,t_4),$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4)\} >$

$< cond(t_4),$ $\{tuple(s,t_3), attr(t_4,lastname,\bar{x}_1), equal(t_3,t_4)\} >$

$< cond(u(t_0,t_4)),$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4)\} >$

$< cond(u(t_0,t_4)),$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >$

$(1) < tuple(ans,u(t_0,t_4)),$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >$

$< valid(u(t_0,t_4),u(t_0,t_4),u(t_0,t_4)),$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >$

$(2) < attr(u(t_0,t_4),name,\bar{x})$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >$

$(3) < attr(u(t_0,t_4),lastname,\bar{x}_1)$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >$

$(4) < attr(u(t_0,t_4),age,\bar{z})$ $\{tuple(q,t_0), attr(t_1,name,\bar{x}), attr(t_2,age,\bar{z}), tuple(s,t_3),$
$attr(t_4,lastname,\bar{x}_1), equal(t_0,t_1,t_2), equal(t_3,t_4), equal(\bar{x},\bar{x}_1)\} >$

The maximal set of described queries with query id $u(t_0,t_4)$ (corresponding to (1),(2),(3) and(4)) is equal to the set of the standard schema queries that are the reduction of the user query. Therefore, the user query is expressible by our RQDL description, by Theorem 8.2. □

8.1 The CBR problem for RQDL

Before attempting to solve the Capabilities-Based Rewriting problem for CBR, let us make the following observations: Algorithm A.1 produces *tuple* and *attr* extended facts with maximal supporting sets[18]. If there exists an extended fact $< attr(t, attr_name, x), S_t >$ in the result, then there also exists an extended fact $< tuple(r,t), S_t >$ for some table r.

We solve the CBR problem for a given query in two steps:

- We generate the set of relevant described queries from the output of the Algorithm A.1, by "glueing" together the *tuple* and *attr* subgoals that have the same supporting set. In other words, we create the corresponding standard schema queries for the extended facts and then do the inverse reduction on the sets of those that have the same id and body (thus ending up with some queries on the original schema). These are the relevant queries of the description with respect to the given query.
- When we have the given query (over some schema) and a number of relevant queries (or views) over the same schema, we can apply an answering queries using views algorithm [Qia96, LMSS95] on that problem.

The complexity of this procedure is non-deterministic exponential in the input size.

[17]We are only showing some of the extended facts that could be produced, for the sake of brevity.

[18]From Theorem 3.7 and Lemma 3.5.

9 Related Work

The different and limited query capabilities of information sources are an important problem for integration systems. In this section we discuss the approaches taken by various systems and we also discuss some theoretical work in this area.

[PGGMU95] suggested a grammar-like approach for describing query capabilities and [LRU96] used a Datalog with tokens for the same purpose. These works are focused on showing how we can compute a query Q given a capabilities description P. The algorithm presented in [PGGMU95] only applies to specific classes of descriptions. We already mentioned that we improved upon the result of [LRU96] for the problem of answering a query using an infinite number of views. Moreover, our paper also studies RQDL, which is more powerful than p-Datalog, and also gives expressiveness results.

RQDL was proposed by [PGH96] to allow capabilities descriptions that are not schema specific. In this paper we show that it is more expressive than p-Datalog and present complete algorithms for query expressibility and CBR. The Information Manifold [LRO96] focuses on the capabilities description of sources found on the Web; hence it does not consider recursion. The expressive power of its capabilities-describing mechanism is strictly less than p-Datalog.

The DISCO system [TRV95] describes the capabilities of the sources using context-free grammars appropriately augmented with actions. DISCO enumerates plans initially ignoring limited wrapper capabilities. It then checks the queries that appear in the plans against the wrapper grammars and rejects the plans containing unsupported queries. DISCO's strategy can be much more expensive than doing capabilities-based rewriting, which ensures that the queries emitted to the wrappers are indeed answerable by the source.

The Garlic system [HKWY96] combines capabilities-based rewriting with cost-based optimization. The assumption is made that all the variables mentioned in a query are always available by the wrapper. This compromises the expressiveness of the description language but greatly simplifies the proposed algorithm. It is also interesting that capabilities descriptions are given in terms of plans supported by the wrappers. Additional assumptions are made at this point regarding the class of plans that can be described.

Finally, RQDL's handling of constructed tuple ids is based on a use of Skolem functions that is close to the ideas in [Mai86, KL89]

10 Conclusions and Future Work

We discussed the problems of (i) describing the query capabilities of sources and (ii) using the descriptions for source wrapping and mediation. We first considered a Datalog variant, called p-Datalog, for describing the set of queries accepted by a wrapper. We also provide algorithms for solving (i) the expressibility and (ii) the CBR problems. The first algorithm decides whether a given query is equivalent to one of the queries described by a p-Datalog program. This algorithm is used by the wrapper. The second algorithm is run by the mediators and it finds out if a given query can be computed using queries which are expressible by a p-Datalog program.

We then study the expressive power of p-Datalog. We reach the important result that p-Datalog can *not* describe the query capabilities of certain powerful sources. In particular, we show that there is no p-Datalog program that can describe all conjunctive queries over a given schema. Indeed, there is no program that describes all boolean conjunctive queries over the schema. A direct consequence of our result is that p-Datalog can not model a fully-fledged relational DBMS.

We subsequently describe and extend RQDL, which is a provably more expressive language

than p-Datalog. The extra power is mainly a result of *vector variables* which can match to sets of attributes of arbitrary length. However, the existence of vector variables makes very hard a brute force implementation of mediator and wrapper algorithms using RQDL. We get around this problem by providing a reduction of RQDL descriptions into p-Datalog augmented with function symbols. Using this reduction we discuss complete algorithms for solving the expressibility and the CBR problem.

We have focused exclusively on conjunctive queries. We plan to extend our work to non-conjunctive queries, *i.e.*, queries involving aggregates and negation. Combining a CBR algorithm with cost-based query optimization also presents interesting challenges.

Acknowledgements

We are grateful to Serge Abiteboul for his help in formalizing the presentation of p-Datalog and the proof of Theorem 5.4. We also thank Jeff Ullman for many fruitful discussions and Luis Gravano for comments on a previous draft of this paper.

References

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[ASU87] A. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1987.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[HKWY96] Laura Haas, Donald Kossman, Edward Wimmers, and Jun Yang. An optimizer for heterogeneous systems with non-standard data and search capabilities. *Special Issue on Query Processing for Non-Standard Data, IEEE Data Engineering Bulletin*, 19:37–43, December 1996.

[KL89] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf.*, pages 134–46, Portland, OR, June 1989.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, pages 95–104, 1995.

[LRO96] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, 1996.

[LRU96] A. Levy, A. Rajaraman, and J.D. Ullman. Answering queries using limited external processors. In *Proc. PODS Conf*, pages 227–37, 1996.

[Mai86] D. Maier. A logic for objects. In J. Minker, editor, *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, Washington, DC, USA, August 1986.

[PGGMU95] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, and J. Ullman. A query translation scheme for the rapid implementation of wrappers. In *Proc. DOOD Conf.*, pages 161–86, 1995. Available via ftp at `db.stanford.edu` file `/pub/papakonstantinou/1995/querytran.ps`.

[PGH96] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. In *Proc. PDIS Conf.*, pages 170–181, 1996. Available via ftp at `db.stanford.edu` file `/pub/papakonstantinou/1995/cbr-extended.ps`.

[Qia96] Xiaolei Qian. Query folding. In *Proc. ICDE*, pages 48–55, New Orleans, USA, 1996.

[RSU95] A. Rajaraman, Y. Sagiv, and J. Ullman. Answering queries using templates with binding patterns. In *Proc. PODS Conf.*, pages 105–112, 1995.

[RSUV89] R. Ramakrishnan, Y. Sagiv, J.D. Ullman, and M.Y. Vardi. Proof tree transformations and their applications. In *Proc. PODS Conf*, pages 172–182, 1989.

[TRV95] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of DISCO. Technical report, INRIA, 1995.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*. Computer Science Press, New York, NY, 1989.

A Expressibility Algorithm for p-Datalog

Algorithm A.1

Input

Minimized [Ull89] (non-rectified) conjunctive query Q of the form $H \leftarrow G_1, G_2, \ldots, G_k$, where the head subgoal H is of the form $ans(X_1, \ldots, X_n)$.
(non-rectified) p-Datalog description P.

Output

A set of minimal containing queries.

Method

Rectify P and Q
Construct the extended canonical DB of Q
Apply the rules of P to the facts in DB to generate *all possible extended facts* using bottom up evaluation [Ull89] modified in the following ways:

1. populate IDB relations with extended facts, *i.e.*, if fact h is produced by the rule, compute $\mathcal{S}_h$ and then enter $< h, \mathcal{S}_h >$ in the database iff
   - $< h, \mathcal{S}_h >$ is not already in the database and
   - No $< h, \mathcal{S}'_h >$ where $\mathcal{S}_h \subseteq \mathcal{S}'_h$ is present in the DB.
2. when a new fact $< h, \mathcal{S}_h >$ is added to the DB, delete from the DB all facts of the form $< h, \mathcal{S}'_h >$, where $\mathcal{S}'_h \subset \mathcal{S}_h$.
3. if a rule is unsafe, *i.e.*, some distinguished variables do not appear in the rule body, simply leave those variables in the produced fact.

In the end:

4. if $< h, \mathcal{S}_h >$ is an extended fact, h is an *ans* fact and h contains variables, delete the extended fact.

% If an *ans* fact has nonground variables, then its expansion is unsafe.
% and is not included in the description of the program.

5. de-rectify the resulting extended facts, and the query Q.

□

The treatment of unsafe rules is the same as in generalized magic sets [Ull89].

The corresponding queries for the facts in the *ans* relation are the minimal containing queries that constitute the output of the algorithm.

B Proofs for Theorems of Sections 3 and 4

In this section, we restate and prove some of the theorems of Sections 3 and 4.
Lemma 3.8 Q is expressible by P *if and only if* the set of supporting facts for some extended fact $< h, \mathcal{S}_h >$ of the frozen head h of Q is identical to the canonical DB for Q[19].

Proof: (Sketch)

IF: It is obvious from the way the "corresponding" query is defined, that if DB $\equiv \mathcal{S}_h$, then the corresponding query is equivalent to Q.

ONLY IF: The output contains *minimal* containing queries, *i.e.*, there is no expansion that is a "tighter fit" than the queries in the output. If for every $\mathcal{S}_h$, there exists some fact in the canonical DB that is not in that $\mathcal{S}_h$ set, then the corresponding query (that is a minimal containing query) cannot be equivalent to Q. The reason for that is that Q is minimized, and minimization is unique up to isomorphism, so all subgoals (*i.e.*, all facts in the canonical DB) are necessary. □

Theorem 4.2 (**CBR**) Assume we have a query Q and a p-Datalog description P, and let $\{Q_i\}$ be the result of applying Algorithm A.1 on Q and P. There exists a rewriting Q' of Q, such that $Q' \equiv Q$, using any $\{Q_j | Q_j$ is expressible by $P\}$ *if and only if* there exists a rewriting Q'' , such that $Q'' \equiv Q$, using only $\{Q_i\}$.

Proof: (sketch) The *if* direction is trivial. For the *only if*: It must be that $Q \subseteq Q_j$ [LMSS95]. Since Q_j is expressible by P, Q_j are containing queries. But $\{Q_i\}$ contains all the minimal containing queries of Q with respect to P by Theorem 3.7, so for any Q_j there exists some "corresponding" Q_i such that $Q_i \subseteq Q_j$. That means that there exists a containment mapping [Ull89] from Q_j to Q_i. Let Q':$Q_{j_1}, \ldots, Q_{j_k}, \ldots, Q_{j_m}$ be the rewritten query. If we replace each Q_{j_k} with its "corresponding" Q_{i_k} identified above, then Q'':$Q_{i_1}, \ldots, Q_{i_m}$ is also equivalent to Q. In proof:

- there exists a containment mapping [Ull89] from Q'' to Q, since $Q \subseteq Q_i$ for all i.
- there exists a containment mapping from Q to Q' and from Q' to Q'', and therefore also from Q to Q''.

Therefore, by the containment mapping theorem [CM77], Q'' and Q are equivalent. That completes the proof of the theorem. □

C Proofs for Theorems of Section 5

Lemma 5.1 There exist *recursive* sets of conjunctive queries that are not expressible by any p-Datalog description.

Proof: The proof is based on a simple complexity argument. As we have seen in the previous section, the decision procedure for the description semantics of p-Datalog is exponential. Therefore, any recursive set of conjunctive queries with a membership function that is super-exponential is not expressible by any p-Datalog description. □

Theorem 5.2 Let k be some integer. Let $p_1, \ldots, p_m$ be the EDB predicates of a database. There exists a p-Datalog program P that describes all conjunctive queries with at most k variables[20] on this database.

[19] After de-rectification of both.

[20] We disregard repeated variables in the head of the conjunctive queries, so we assume that the result predicate has arity at most k.

Proof: (sketch) We show the construction for $k = 3$ and for the case where $p_1, \ldots, p_m$ are each predicates of arity two. The program P that can describe all conjunctive queries is the following:

$$\begin{aligned}
ans_3(X_i, X_j, X_l) &\leftarrow temp(X_1, \ldots, X_k), \forall i, j, l \leq k & (8)\\
ans_2(X_i, X_j) &\leftarrow ans_3(X_i, X_j, X_j), \forall i, j \leq k & (9)\\
ans_1(X_i) &\leftarrow ans_2(X_i, X_i), \forall i \leq k & (10)\\
ans_0() &\leftarrow ans_1(X) & (11)\\
temp(X_1, \ldots, X_k) &\leftarrow p_l(X_i, X_j), temp(X_1, \ldots, X_k) \;\; \forall l \leq m, \; \forall i, j \leq k & (12)\\
temp(X_1, \ldots, X_k) &\leftarrow p_l(X_i, \$c), temp(X_1, \ldots, X_k) \;\; \forall l \leq m, \; \forall i \leq k & (13)\\
temp(X_1, \ldots, X_k) &\leftarrow p_l(\$c, X_j), temp(X_1, \ldots, X_k) \;\; \forall l \leq m, \; \forall j \leq k & (14)\\
temp(X_1, \ldots, X_k) &\leftarrow p_l(\$c_1, \$c_2), temp(X_1, \ldots, X_k) \;\; \forall l \leq m & (15)\\
temp(X_1, \ldots, X_k) &\leftarrow \epsilon & (16)
\end{aligned}$$

where $X_1, \ldots, X_k$ are distinct variables. □

Lemma 5.3 Let P be a Datalog program and k the maximum number of variables occurring in a rule of P. Then for each $Q \lhd P$, Q is equivalent to a query in FO^k (using only $\wedge$ and $\exists$).

Proof: (sketch) Let $x_1, \ldots, x_k$ be the variables appearing in the rules of description P. Also, let

$$Q' : ans(u_1) \leftarrow p_1(u_2), p_2(u_3), \ldots, p_n(u_n)$$

be in $descr(P)$ such that $Q \equiv Q'$. We will show that Q' is equivalent to a first order sentence with only k variables.

The proof is by induction on the number of resolution steps used to construct a rule. If Q' is a rule of P, then the claim is true. Otherwise, when doing a step of the resolution, let q_i be the literal that is unified with some rule head. Then, the variables not used in q_i can be reused existentially quantified for the extra variables in the rule. □

Theorem 5.4 Let the database schema **S** have a relation of arity at least two. For every p-Datalog description P over **S**, there exists a boolean query Q over **S**, such that Q is not expressible by P. (So, in particular, there is no p-Datalog description that could describe a source that can answer all conjunctive queries, even if we fix the arity of the answer.)

In order to prove this, we first need to prove the following lemma:

Lemma C.1 Let a database consist of a binary relation G that contains no self loops. The question "is there a k-clique in G" can be expressed by a conjunctive query (with k variables) but is not in FO^{k-1}.

Proof: (sketch) The question is clearly expressed by the following query:

$$\begin{aligned}
ans() \leftarrow\; & G(x_1, x_2), \ldots, G(x_1, x_k), \ldots,\\
& G(x_i, x_1), \ldots, G(x_i, x_{i-1}), G(x_i, x_{i+1}), \ldots, G(x_i, x_k)\\
& G(x_k, x_1), \ldots, G(x_k, x_{k-1})
\end{aligned}$$

This query cannot be expressed [AHV95] by an FO^{k-1} formula, as can be shown by playing an Ehrenfeucht-Fraissé game (see [AHV95]), on the following two structures: G_1, a k-clique without self-loop and G_2, a k-1 clique without self-loop. □

Now we are ready to prove Theorem 5.4.

Proof: Let **S** (without loss of generality) contain the binary predicate G. Suppose such a description P exists. Let k be the maximum number of variables in a rule of P. Then each conjunctive query expressible with P is in FO^k by Lemma 5.3. But then the $k+1$ clique without self-loop is not in P. □

D RQDL Grammar and *valid* Rules

The following table contains the complete syntax of RQDL.

(0) $\langle \mathit{description} \rangle$	::=	$(\langle \mathit{ans_query\ template} \rangle \mid \langle \mathit{rule\ template} \rangle)*$
(1) $\langle \mathit{ans_query\ template} \rangle$	::=	$\mathbf{ans}() \leftarrow \langle \mathit{subgoal\ list} \rangle$
(2) $\langle \mathit{rule\ template} \rangle$	::=	$\langle \mathit{predicate\ name} \rangle\ (\ \langle \mathit{arguments} \rangle\ ) \leftarrow \langle \mathit{subgoal\ list} \rangle$
(3) $\langle \mathit{subgoal\ list} \rangle$	::=	$\langle \mathit{subgoal} \rangle\ (,\ \langle \mathit{subgoal} \rangle)*$
(4) $\langle \mathit{subgoal\ list} \rangle$	::=	$\langle \epsilon \rangle \% \mathit{subgoal\ list\ may\ be\ emtpy}$
(5) $\langle \mathit{subgoal} \rangle$	::=	$\langle \mathit{predicate} \rangle\ (\ \langle \mathit{arguments} \rangle\ ) \quad \% \mathit{predicate}$
(6) $\langle \mathit{subgoal} \rangle$	::=	$\langle \mathit{metapredicate\ name} \rangle\ (\ \langle \mathit{arguments} \rangle\ ) \quad \% \mathit{metapredicate}$
(7) $\langle \mathit{subgoal} \rangle$	::=	$\langle \mathit{token} \rangle\ (\ \langle \mathit{arguments} \rangle\ ) \quad \% \mathit{token\ as\ predicate}$
(8) $\langle \mathit{arguments} \rangle$	::=	$\langle \mathit{argument} \rangle (, \langle \mathit{argument} \rangle)*$
(9) $\langle \mathit{argument} \rangle$	::=	$\langle \mathit{vector} \rangle \mid \langle \mathit{variable} \rangle \mid \langle \mathit{identifier} \rangle \mid \langle \mathit{token} \rangle$
(10) $\langle \mathit{predicate\ name} \rangle$	::=	$\langle \mathit{identifier} \rangle \mid \langle \mathit{token} \rangle$
(11) $\langle \mathit{metapredicate\ name} \rangle$	::=	**union**|**set_item**|**subset**
(12) $\langle \mathit{nonterminal\ name} \rangle$	::=	$\langle \mathit{identifier} \rangle$
(13) $\langle \mathit{token} \rangle$	::=	$\$\langle \mathit{identifier} \rangle$

Figure 3: RQDL syntax

The next figure contains the rules that define the predicate *valid* (see subsection 7.3).

$$\begin{array}{l}
valid(u(T_1,T_2),T_1,T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2 \\
valid(u(T_2,T_1),T_1,T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1 \\
valid(T,T,T) \leftarrow tuple(N_1,T) \\
valid(u(T_1,T), u(T_1,T_1'), T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2, valid(T,T_1',T_2) \\
valid(u(T_2, u(T_1,T_1')), u(T_1,T_1'), T_2) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1 \\
valid(u(T,T_1), u(T,T_1), T) \leftarrow tuple(N,T) \\
valid(u(T_1, u(T_2,T_2')), T_1, u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2 \\
valid(u(T_2,T), T_1, u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1, valid(T,T_1,T_2') \\
valid(u(T,T_1), T, u(T,T_1)) \leftarrow tuple(N,T) \\
valid(u(T_1, u(T_2,T)), u(T_1,T_1'), u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_1 < T_2, valid(T,T_1',T_2') \\
valid(u(T_2, u(T_1,T)), u(T_1,T_1'), u(T_2,T_2')) \leftarrow tuple(N_1,T_1), tuple(N_2,T_2), T_2 < T_1, valid(T,T_1',T_2') \\
valid(u(T,T'), u(T,T_1), u(T,T_2)) \leftarrow tuple(N,T), valid(T',T_1,T_2)
\end{array}$$

Figure 4: Default rules for the generation of valid $u-terms$.