On the Optimization of Recursive Relational Queries: Application to Graph Queries

Author names omitted for double blind reviewing

ABSTRACT

Graph databases have received a lot of attention as they are particularly useful in many applications such as social networks, life sciences and the semantic web. Various languages have emerged to query graph databases, many of which embed some form of recursion which reveals essential for navigating in graphs. The relational model has benefited from a huge body of research in the last half century and that is why many graph databases either rely on (or have adopted the techniques of) relational based query engines. Since its introduction, the relational model has seen various attempts to extend it with recursion and it is now possible to use recursion in several SQL or Datalog based database systems. The optimization of recursive queries remains, however, a challenge. We propose μ-RA, a variation of the Relational Algebra equipped with a fixpoint operator for expressing recursive relational queries. μ-RA can notably express unions of conjunctive regular path queries. Leveraging the fact that this fixpoint operator makes recursive terms more amenable to algebraic transformations, we propose new rewrite rules. These rules makes it possible to generate new query execution plans, that cannot be obtained with previous approaches. We present the syntax and semantics of μ-RA, and the rewriting rules that we specifically devised to tackle the optimization of recursive queries. We report on practical experiments that show that the newly generated plans can provide significant performance improvements for evaluating recursive queries over graphs.

1 INTRODUCTION

The expressive power of query languages has been greatly improved with the introduction of recursion. Recursive queries are, for instance, very useful in data integration since expressive ontologies use recursion [27]. Graph databases are another example where recursion is particularly useful for expressing navigation along paths connecting nodes in the graph. For this purpose, graph query languages often include constructs such as Regular Path Queries (RPQs) [26], and various extensions such as Conjunctions of them (CRPQs) and Union of CRPQs (UCRPQs) [18, 19, 25, 46]. For instance, the query language SPARQL 1.1 [40] introduced Property Paths, and language proposals such as OpenCypher [32, 51] and G-core [11] also include the possibility of expressing recursive paths. SPARQL's Property Paths revealed crucial for extracting information from RDF data structures such as those found in social networks, life sciences and transportation networks. However, recursive path queries are notoriously known to be much harder to optimize and evaluate than non-recursive ones [50, 65]. In practice, even with datasets of modest sizes, the benchmarking work found in [14] notices that "*all tested systems either failed on the majority of these* [recursive] *queries or had to be manually terminated after unexpectedly long running times.*" A major difficulty is to find an appropriate way to execute the query, a task frequently referred to in the literature as finding an appropriate Query Execution Plan (QEP). For example, let us consider the following query:

```
?x isLocatedIn+ ?y
Emmy_Noether worksAt ?x
```
Q_{ex}

In a graph, this query retrieves all pairs of nodes ?x, ?y such that Emmy Noether worked for ?x and ?x is located in ?y, or is located in a place that is located in ?x, *etc.*. Formally, isLocatedIn+ indicates that it is the *transitive closure* of isLocatedIn. Different QEPs exist for executing Q_{ex}. For instance, the following two are considered by the approach that we develop in this paper:

- A first QEP, named $\mathcal{P}_1$, corresponds to computing first the transitive closure of isLocatedIn and then joining it with the ?x solution of Emmy_Noether worksAt ?x.
- Another QEP, noted $\mathcal{P}_2$, would rather start by computing the set of solutions for Emmy_Noether worksAt ?x. Then, these results are joined with the set of pairs ?x,?y solutions of ?x isLocatedIn ?y. The resulting pairs of nodes are all solutions of Q_{ex}. Finally, iteratively, for each pair ?x,?y in the set of solutions one can find the ?y' such

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that `?y isLocatedIn ?y'` and add `?x,?y'` to the set of solutions.

The QEP $\mathcal{P}_2$ is generally more efficient than $\mathcal{P}_1$. Each pair of nodes (?x,?y) processed in $\mathcal{P}_2$ is a solution of Q_{ex}. Therefore the total running time of $\mathcal{P}_2$ is linear in the number of solutions multiplied by the maximal degree of nodes in the graph. In contrast, in $\mathcal{P}_1$, the transitive closure of `isLocatedIn` always needs to be fully computed. This closure can contain, in the worst case, a number of elements which is quadratic in the number of `isLocatedIn`-labeled edges; and even in the best case, the transitive closure contains at least all the `isLocatedIn`-labeled edges. In the YAGO dataset [31, 59], that contains millions of entities and facts extracted from Wikipedia, there are only 16 solutions for Q_{ex} whereas there are millions of `isLocatedIn`-labeled edges. $\mathcal{P}_2$ is thus expected to be by far more efficient than $\mathcal{P}_1$.

Let us now consider a slightly more sophisticated example with the following query (taken from [5]), still intended to be executed against the YAGO dataset:

`?x hasChild/livesIn/isLocatedIn+/dealsWith+ Japan` $\quad Q_2$

The variable `?x` denotes any graph node representing a person. Q_2 retrieves all such nodes that are connected to a particular node labeled "`Japan`" through a path which must satisfy a regular path expression over edge labels in the graph. The regular path expression is recursive because it includes the subexpression R=`isLocatedIn+/dealsWith+` in which the operator "+" stands for the transitive closure. Again, there are several ways to evaluate Q_2 for retrieving all possible values for `?x`. In particular, there exists different QEPs for R corresponding to radically different manners of executing Q_2. We briefly describe below three QEPs: $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$ that are considered by the new approach that we develop:

- Plan $\mathcal{P}_3$ consists in first computing the two transitive closures `isLocatedIn+` and `dealsWith+` of the relations `isLocatedIn` and `dealsWith`, respectively, and then joining the results. One pitfall of this evaluation plan is that the sets to be joined might be very large (due to e.g. numerous locations in the database and the large size of the set of pairs connected by "`isLocatedIn+`"), even though the overall query finally retrieves only few results.
- Early works on recursive query optimization [7, 8] already proposed how to push filters (and projections) as close to the sources as possible, even through recursive terms. Plan $\mathcal{P}_3$ can be optimized using this technique: the constant "`Japan`" can be "pushed inside the computation of `dealsWith+`". This corresponds to plan $\mathcal{P}_4$ where the evaluation starts from `?t dealsWith Japan`, then iteratively adds `dealsWith` steps on the left. Once this computation is finished, the results are joined with `isLocatedIn` and then, once again, iterative `isLocatedIn` steps are performed on the left. One advantage of $\mathcal{P}_4$ compared to $\mathcal{P}_3$ is that $\mathcal{P}_4$ processes each node at most twice while the whole transitive closures of `dealsWith` and `isLocatedIn` can be considerably larger (quadratic in the worst case).
- Another completely different way to evaluate R, noted plan $\mathcal{P}_5$, consists in not computing any transitive closure but, instead, computing first the composed relation "`isLocatedIn/dealsWith`" and then recursively navigating in the graph either hopping on the left with `isLocatedIn` or hopping on the right by `dealsWith` to retrieve nodes. Notice that this slightly more general form of recursion is not a transitive closure of any relation.

The plan $\mathcal{P}_3$ will generally be slower than $\mathcal{P}_4$ or $\mathcal{P}_5$, even when using smart algorithms for computing transitive closures [9, 20, 56]. On some instances $\mathcal{P}_4$ will be much more efficient (if, for instance, `?x dealsWith+ Japan` has few solutions) and on some other instances $\mathcal{P}_5$ will be faster (if, for instance, there are few solutions of `dealsWith/isLocatedIn`). This is the case when evaluating Q_2 over the YAGO graph (that has more than 62 million of edges): $\mathcal{P}_5$ is faster by a factor of more than 200x compared to other plans.

We investigate the problem of computing such query evaluation plans automatically.

Contribution. We introduce a theory, which is an extension of Codd's classical relational algebra, for the purpose of automatically obtaining efficient evaluation plans for recursive queries. Specifically, we introduce a fixpoint operator "μ" in the relational algebra for denoting recursive terms in an algebraic manner. This fixpoint operator can express transitive closures as well as slightly more general forms of recursion. It makes recursive terms more amenable to transformations. We take advantage of this for introducing new rewriting optimization rules. These rules allow generating new execution plans for recursive queries, that are beyond reach using previous approaches. We demonstrate empirically with a prototype implementation that these new plans can provide significant performance gains in recursive graph query evaluation compared to previous approaches.

Outline. We first review related works on the topic in § 2. Then in § 3, we introduce the syntax and semantics of μ-RA: a variation of the relational algebra (RA) equipped with a fixpoint operator. § 4 describes properties of μ-RA and demonstrates that our fixpoint can be rewritten (to push filters, projections and joins inside of the fixpoints) and that two (or more) fixpoints can sometimes be merged into a unique fixpoint. We also explain why our approach is able to generate efficient plans that were beyond reach. As an application, we present how recursive graph queries translate into μ-RA in § 5. We then report on practical experiments in § 6, in which we benchmark a prototype implementation with state-of-the-art systems for evaluating recursive graph queries. For reading purposes, we present only proof sketches of our main theorems; the full proofs being available in [12].

2 RELATED WORK

We first present the main approaches that have been proposed to evaluate recursive queries, and we explain why our approach can be more efficient, in particular for the specific kind of recursive queries expressed as UCRPQs.

2.1 The relational model

The relational model [24] introduced by Codd in 1970 has become the de facto paradigm for querying data banks. The design of the prevailing database query language SQL has been heavily influenced by the Relational Algebra (RA).

One of the main interest of the RA (and of SQL) is that it allows programmers to express the data they are interested in without specifying the way to retrieve it [47], instead they rely on a relational query engine to find an efficient way to process queries. Most relational query engines thus rely on a optimization process where the query to be processed is rewritten into a query that is semantically equivalent but more efficiently processed. For this operation many optimizers first rewrite the input query into multiple equivalent queries and then use a cost estimation to select the estimated best among those. Each of the different queries obtained by the rewriting process is a possible QEP and the set of such considered queries is thus called a *Plan Space*.

2.2 Recursive queries & expressive power

Soon after the introduction of RA, the work found in [8] noticed that the RA lacks the possibility of expressing recursive queries. Several formalisms have been introduced as attempts to fill this gap. We now briefly review the most closely related of these various formalisms (see e.g. [2, 17] for complete surveys).

The α-extended RA [7] extends the RA with a recursive facility noted α. The operator α is a transitive closure operator; given an α-extended RA term R defining a binary relation $\mathcal{R}$, $\alpha(R)$ represents its transitive closure, $\mathcal{R}^+$. In terms of expressive power over graphs, this corresponds to UCRPQs. It is for instance not capable of querying the path expression a^n/b^n which queries paths whose labels are first as then bs with exactly the same number of a and b.

A more powerful way of extending the RA is the LFP-RA. LFP-RA is the RA extended with a "least fixpoint" construct. Given a LFP-RA term R parametric in a relation X, the *least fixpoint* of R over X, noted $\mu(X = R)$, is obtained as the limit of a sequence $(X_i)_{i\in\mathbb{N}}$ where $X_0 = \emptyset$ and X_{i+1} is computed by adding to X_i the results of evaluating R when the relation X is filled with X_i.

This LFP-RA formalism has the same expressive power and the same data complexity as Datalog with stratified negation, for which evaluation is known to lie in PTime. A well-studied fragment of this LFP-RA corresponds to limiting fixpoints so that all recursions are "linear" (in a sense that we will define in Definition 6). This restriction makes the expressive power drop to linear Datalog [6], which is strictly between the expressive power of α-extended RA and the expressive power of Datalog. For instance, all UCRPQs as well as the non-regular path expression a^n/b^n are expressible in linear Datalog; however, the path expression asking for all paths containing exactly as many a labels as b labels *in any order* is not. *The μ-RA that we use in this paper is a variant of this restricted LFP-RA and it has the same expressive power as linear Datalog.*

Another way of extending the RA is the WHILE language (see [2]). This language is known to be at least as expressive as LFP-RA and the inclusion is strict (unless PTime=PSpace).

2.3 Optimization of recursive RA

Since Datalog is more popular and since the recursive RAs can generally be translated into Datalog, most of the work on optimizing recursive queries has been performed on Datalog. We present in this subsection two notable exceptions and we will review in the next section the optimization of Datalog programs.

In 1979, a first line of work [8] proposed a restricted LFP-RA. The authors showed that some optimizations can be performed on fixpoints, such as pushing selections into these fixpoints (similar to our theorem 1). One drawback of their method is that it produces complex terms and handles only some types of fixpoints (it is more restricted than our μ-RA). Furthermore, even if the authors conjecture that it is sometimes possible to push projections (similar to theorem 5) they do not provide an effective criterion for this.

Using the (unrestricted) LFP-RA, other authors [43] provided in 1990 a general framework of optimization. Their method works on System Graphs (SG) and computes a fixpoint of filters that is safe to apply recursively. Their work also provides an effective criterion to push projections. We believe our approach is more straightforward, and it is also more general: they do not deal with conjunctions (our theorems 3 and 4) and thus cannot find plan $\mathcal{P}_2$ for Q_{ex} nor find plan $\mathcal{P}_5$ for Q_2.

2.4 Datalog

The term Datalog has been coined at the end of the 70s to designate the fragment of logic programming restrained to data. Datalog is a language supporting recursive queries.

2.4.1 *Magic Sets.*

One of the most well-known optimization algorithms for Datalog is the "Magic Sets" algorithm (see [16, 53]). The syntax of Datalog vastly differs from the RA, but the effect of Magic Sets algorithm is very similar to pushing some type of selections and projections. The idea of the magic set is to compute, for each datalog relation, the set of "contexts" where this term will be evaluated. For instance, in the translation of `?a dealsWith+ Japan`, the magic set method can sometimes detect that, on the recursive use of `dealsWith`, the right side is always Japan, and it will not compute the full transitive closure `dealsWith+`.

2.4.2 Right and left linear programs. There are many different ways to translate transitive closures in Datalog. For the transitive closure R^+, one translation would use that R^+ is either a path composed of a single R or the concatenation of a path R and a path R^+. Such a translation is called right linear because R^+ is computed by adding R paths on the left. Another translation would be left-linear: R^+ is then either a path R or the concatenation of R^+ with R. As noticed before [49], given a Datalog term t computing a binary relation $P(x, y)$, the Magic Set algorithm is able to push filters on the right side (the y) only if t is a right-linear program; and conversely, it can only push filters on the left side (the x) when t is left-linear. The authors of [49] thus proposed an automated way to "reverse" right-linear programs into left-linear (and vice-versa). This reversal can then be used by a query optimizer in combination with the Magic Set algorithm.

2.4.3 Demand Transformation. The Demand-driven Transformation, or Demand Transformation, is a recent improvement [60] over the Magic Sets. The idea is similar to the Magic Set: it pushes filters to avoid computing some "useless" facts. It has been proven that Demand Transformation always beats Magic Sets; however, it still suffers from some of the problems of Magic Sets. In particular, Demand Transformation is also sensitive to whether programs are left or right linear; and on examples containing the concatenation of two transitive closures (such as $\texttt{isLocatedIn}^+/\texttt{dealsWith}^+$ in Q_2), the query execution plan computing first the concatenation `isLocatedIn/dealsWith` and then recursively adding `isLocatedIn` on the left or `dealsWith` on the right will not be considered by the Datalog engine, even after Demand Transformation and program reversals.

If a Datalog engine uses a combination of Magic Sets or Demand Transformation and reversals then it will consider plan $\mathcal{P}_2$ for Q_{ex}, and plan $\mathcal{P}_4$ for Q_2; but it will not be able to consider plan $\mathcal{P}_6$ for Q_2.

2.5 SQL

Since its 1999 version, the SQL standard supports recursive queries via the Common Table Expressions (CTE). SQL is more expressive than our μ-RA as it allows, e.g., arithmetic and aggregation, but if we restrict SQL to its core our proposed μ-RA is not very different from SQL equipped with recursive CTE and therefore the best plans offered by SQL 1999 are not very different from ours. However what is lacking is the possibility for SQL to optimize terms with recursive CTE. Not all vendors support CTE or recursive CTE and those who do support CTE tend to consider CTE as optimization barriers. There are exceptions (such as DB2) but these vendors use a technique inspired by the Magic Set technique invented for Datalog.

2.6 Ad-hoc evaluation of UCRPQs

Up to now, we have only compared systems operating on languages strictly more expressive than UCPRQs. If we are only interested in UCRPQs or RPQs there are others systems and we now present the main ones.

2.6.1 Automata. One way to evaluate UCRPQ is to translate individual RPQs to automata, run them, and then union-join the individual results. This automata-based method is clearly not optimal for UCRPQs as the various RPQs are considered individually and therefore the constraints on one RPQ cannot be used on the others but the automata is sometimes not optimal even on a single RPQ. Indeed, let us suppose that we are considering the regular expression $(a/b/c)^+$, the automata approach will start by computing the solution to the path a, then to the path a/b, then $a/b/c/c$ and recursively restart (i.e. in a computational form we have $R_{i+1} = ((R_i/a)/b)/c)$. The paper introducing Waveguide [64] notes that this method forces the associativity of the computation and do not test some associations such as computing $R_{i+1} = (R_i/a)/(b/c)$ in which we precompute paths matching (b/c) and which can, in some cases, be much more efficient, when e.g. there are only a handful of solutions to the path b/c.

2.6.2 α-extended RA. The paper introducing Waveguide [64] also notes that the α-extended RA is suboptimal because it forces the associativity of the computation. The translation of the RPQ $(a/b/c/)^+$ in the α-extended RA will include a term $\alpha(t)$ where t computes $(a/b/c)$. Therefore the computation will compute once the whole $(a/b/c)$ then its transitive closure which is not always optimal when for instance the left side is bounded.

2.6.3 Waveguide. The Waveguide paper [64] introduced a new technique to evaluate RPQs that mixes ideas from automata and from the α-extended RA. Their idea is that the "interesting" plans to evaluate RPQ one can either start on the left of the RPQ and try to match the right part, or do the opposite (start on the right part and match the left one) or start in the middle and go both way. For instance on Q_2 they will try all the plans that we express. However, since they are focusing on single RPQ on a conjunction of RPQ such as Q_{ex} Waveguide will not take advantage of the constraint on `?x` and will materialize the full relation `owns+`. Moreover, on a query `?a dealsWith+ ?b, ?b isLocatedIn+ ?c`, our approach will have a single fixpoint in which the number of mappings treated is exactly the number of solutions, while their approach will compute and then join the full transitive closures `dealsWith+` and `isLocatedIn+`.

The authors of Waveguide extended their work in two short papers. The first one describes Tasweet, a system focusing on disjunction of RPQs and the second one presents Wireframe, a tool focusing on conjunctions of RPQs. Wireframe computes a "query spanning tree" to decide an order in which evaluate RPQs and then relies on Waveguide for individual RPQs. Tasweet improves on Waveguide by noticing that given a set of disjunctive RPQs some of the computation can be shared. Both of these works suffer from the some of same limitations of Waveguide, for instance in Wireframe the constraints on one node can be used to constrain the evaluation of other RPQs but the evaluation of various RPQs cannot be interleaved. Furthermore these tools are limited to conjunction or union of RPQs.

3 THE μ-EXTENDED REL. ALGEBRA

We present our variation of the domain-independent relational algebra, equipped with a fixpoint, which we call μ-RA (μ-extended relational algebra). We first recall some usual definitions. Then we present our syntax, types and semantics that are adapted from the RA for our new construct.

3.1 Data model

Our data model is the same as for the classical relational algebra: we consider *relations* which are sets of *mappings* (also called tuples, or lines) which associate *column names* to *values*. Formally, we assume the following constants:

- $\mathfrak{V}$ an infinite set of *values*;
- $\mathfrak{C}$ an infinite set of *column names*;
- $\mathfrak{R}$ an infinite set of *relation names*.

Definition 1. *A* mapping *or tuple is a partial function $m\colon \mathfrak{C} \to \mathfrak{V}$ whose domain is finite. If $dom(m) = \{c_1, \dots, c_n\}$, m can also be seen as the set $\{c_1 \to m(c_1), \dots, c_n \to m(c_n)\}$.*

Definition 2. *Two mappings m_1 and m_2 are* compatible*, noted $m_1 \sim m_2$, when $\forall c \in dom(m_1) \cap dom(m_2)$, $m_1(c) = m_2(c)$. If m_1 and m_2 are compatible, we define $m_1 + m_2 : dom(m_1) \cup dom(m_2) \to \mathfrak{V}$ by:*

$$(m_1 + m_2)(c) = \begin{cases} m_1(c) & \text{if } c \in dom(m_1) \\ m_2(c) & \text{if } c \in dom(m_2) \end{cases}$$

If we see mappings as sets, this corresponds to their union.

Definition 3. *A* relation *is a finite set of mappings which share the same domain. We call this common domain the* type *of the relation. We do not consider datatypes (all values are in the single domain $\mathfrak{V}$): a type is just a set of column names.*[1] *The empty relation is considered compatible with all types.*

Relations represent data. A relational database is a finite set of named relations (also called tables). We represent such a database as a triple $(\mathcal{R}, \Gamma, D)$ where: $\mathcal{R} \subset \mathfrak{R}$ is the set of relation names; Γ, the database schema, associates relation names to relation types; and D, the database body, associates relation names to actual relations. The body must be consistent with the schema: for any $R \in \mathcal{R}$, $D(R)$ has type $\Gamma(R)$.

3.2 Syntax of μ-RA terms

Our algebra μ-RA is mainly a variation of the relational algebra, with the addition of a fixpoint operator μ inspired from the μ-calculus [45]; it operates on relations. The terms represent queries and are built from relation variables and operations; given a mapping from variables to relations (representing a database body), a term can be evaluated and yields another relation (the solution of the query). Evaluation of terms is described in § 3.3.

3.2.1 Filters. The standard selection operation $\sigma_{\mathfrak{f}}$, which operates on a relation by keeping only a subset of its mappings, depends on a *filter* $\mathfrak{f}$ indicating which mappings are to be kept. This filter can be seen as a function from mappings to booleans. To keep things focused, we do not detail here a syntax for filters, but we assume that for any filter $\mathfrak{f}$ we can compute a set $FC(\mathfrak{f})$ of column names such that the result of $\mathfrak{f}(m)$ depends only on $\{c \to m(c) \mid c \in FC(\mathfrak{f})\}$.

3.2.2 Terms. The core syntax of terms is defined in Fig. 3. The base terms are relation variables X and constants $|c \to v|$ (representing a single mapping with a singleton domain). Two relations can be combined with the classical relational operators $\cup$, $\bowtie$ and $\triangleright$. One relation can be filtered using the classical selection operation $\sigma_{\mathfrak{f}}$ where $\mathfrak{f}$ is a filter. The rename operator $\rho_a^b(\cdot)$ renames the column a into b. Less classically, the anti-projection $\widetilde{\pi}_a(\cdot)$ (or column dropping) removes column a. The projection operator $\pi_{p_1,\dots,p_n}(\varphi)$ can be expressed in terms of $\widetilde{\pi}(\cdot)$ provided we know the type of φ: if φ has type $t = \{p_1, \dots, p_n, a_1, \dots, a_k\}$ we have $\pi_{p_1,\dots,p_n}(\varphi)$ equivalent to $\widetilde{\pi}_{a_1}\left(\dots \widetilde{\pi}_{a_k}(\varphi)\right)$. Our choice of anti-projection will allow us to extend the domains of subterms without changing the projections, as in $\widetilde{\pi}_a(\varphi) \bowtie \psi \to \widetilde{\pi}_a(\varphi \bowtie \psi)$ when a is not in the type of ψ.

Finally, we introduce the fixpoint term $\mu(X = \varphi)$ representing a recursive query. In this term, there are some additional restrictions on φ, which will be detailed in § 3.5. The result R of this operation is a fixpoint in the sense that evaluating

[1] We do this for simplicity; datatypes could be added by replacing column names with pairs of a column name and a datatype without changing our theory much.

$$
\begin{array}{rcl}
[\![\varphi_1 \bowtie \varphi_2]\!]_V &=& \{m_1 + m_2 \mid m_1 \in [\![\varphi_1]\!]_V \wedge m_2 \in [\![\varphi_2]\!]_V \wedge m_1 \sim m_2\} \\
[\![\varphi_1 \triangleright \varphi_2]\!]_V &=& \{m \in [\![\varphi_1]\!]_V \mid \forall m' \in [\![\varphi_2]\!]_V\ \neg(m' \sim m)\} \\
[\![\widetilde{\pi}_a\,(\varphi)]\!]_V &=& \left\{\{c \to v \in m \mid c \neq a\} \,\middle|\, m \in [\![\varphi]\!]_V\right\} \\
[\![\rho^b_a\,(\varphi)]\!]_V &=& \left\{\{c \to v \in m \mid c \neq a\} \cup \{b \to v \mid a \to v \in m\} \,\middle|\, m \in [\![\varphi]\!]_V\right\} \\
[\![\mu(X = \varphi)]\!]_V &=& [\![X]\!]_{V[X/U_\infty]} \quad \text{where } U_0 = \emptyset,\ U_{i+1} = U_i \cup [\![\varphi]\!]_{V[X/U_i]}, \text{ and } U_\infty = \bigcup_{n \in \mathbb{N}} U_i
\end{array}
\qquad
\begin{array}{rcl}
[\![\varphi_1 \cup \varphi_2]\!]_V &=& [\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V \\
[\![|c \to v|]\!]_V &=& \{\{c \to v\}\} \\
[\![X]\!]_V &=& V(X) \\
[\![\sigma_{\mathfrak{f}}\,(\varphi)]\!]_V &=& \{m \mid m \in [\![\varphi]\!]_V \wedge \mathfrak{f}(m) = \top\}
\end{array}
$$

Figure 1: Semantics of μ-RA

$$
\Gamma \vdash |c \to v| : c
\qquad
\frac{\Gamma(X) = t}{\Gamma \vdash X : t}
\qquad
\frac{\Gamma \vdash \varphi_1 : t \qquad \Gamma \vdash \varphi_2 : t}{\Gamma \vdash \varphi_1 \cup \varphi_2 : t}
\qquad
\frac{\Gamma \vdash \varphi_1 : t_1 \qquad \Gamma \vdash \varphi_2 : t_2}{\Gamma \vdash \varphi_1 \bowtie \varphi_2 : t_1 \cup t_2}
\qquad
\frac{\Gamma \vdash \varphi_1 : t_1 \qquad \Gamma \vdash \varphi_2 : \_}{\Gamma \vdash \varphi_1 \triangleright \varphi_2 : t_1}
$$

$$
\frac{\Gamma \vdash \varphi : t \qquad FC(\mathfrak{f}) \subseteq t}{\Gamma \vdash \sigma_{\mathfrak{f}}\,(\varphi) : t}
\qquad
\frac{\Gamma \vdash \varphi : t \qquad a \in t \qquad b \notin t}{\Gamma \vdash \rho^b_a\,(\varphi) : (t \setminus \{a\}) \cup \{b\}}
\qquad
\frac{\Gamma \vdash \varphi : t \qquad a \in t}{\Gamma \vdash \widetilde{\pi}_a\,(\varphi) : t \setminus \{a\}}
\qquad
\frac{\Gamma \cup \{X \to t\} \vdash \varphi : t}{\Gamma \vdash \mu(X = \varphi) : t}
$$

Figure 2: Typing rules for μ-RA

$$
\begin{array}{lllr}
\varphi & ::= & & \text{term} \\
 & & X & \text{relation variable} \\
 & | & |c \to v| & \text{constant} \\
 & | & \varphi_1 \cup \varphi_2 & \text{union} \\
 & | & \varphi_1 \bowtie \varphi_2 & \text{join} \\
 & | & \varphi_1 \triangleright \varphi_2 & \text{antijoin} \\
 & | & \sigma_{\mathfrak{f}}\,(\varphi) & \text{filtering} \\
 & | & \rho^b_a\,(\varphi) & \text{renaming} \\
 & | & \widetilde{\pi}_a\,(\varphi) & \text{anti-projection} \\
 & | & \mu(X = \varphi) & \text{fixpoint}
\end{array}
$$

Figure 3: Grammar of μ-RA

φ with X bound to R must yield R again. The restrictions we add ensure that this fixpoint exists and can be computed iteratively. We consider μ as a variable binder, yielding the standard notions of *free* and *bound* variable occurrences:

Definition 4. *In a term φ, all occurrences of a variable X which appear in a subterm of the form $\mu(X = \psi)$ are* bound. *All other occurrences of X are* free.

As will be clear from the semantics, bound variables can be renamed, as usual, without changing the meanings of the terms. We can thus assume for simplicity that all bound variables are different from each other and from free variables.

3.3 Semantics

In μ-RA, relation variables X are used to denote both references to a database relation and a recursive relation. In a full query, the two are distinguished by the fact that database references appear as free variables, whereas recursion variables are bound by μ; but in a subterm, we do not need to distinguish the two. In all cases, the semantics of a term φ depends on an *environment* V which maps all free variables of φ to relations.

The semantics is defined in Fig. 1, where $[\![\varphi]\!]_V$ designates the result of evaluating φ in the environment V. This result is defined recursively from the results of evaluating the subterms. The initial environment for evaluating the whole term is a database body D, but in evaluating $\mu(X = \varphi)$, the recursive calls use different environments where the recursion variable X is given a value: the notation $V[X/S]$ represents the environment V altered by mapping X to S.

3.4 Type System

Given a schema Γ for a set of relation variables $\mathcal{R}$, we can infer types for terms whose free variables are in $\mathcal{R}$. The typing judgement $\Gamma \vdash \varphi : t$ means that when evaluated in an environment conforming to the schema Γ, φ will yield a relation of type t.

Our type system is defined by the rules on Fig. 2; it is quite straightforward. The only difficulty is to infer a type for fixpoints (last rule), since the rule does not give an explicit way to guess the value of t. However, this rule is still operational. Indeed, we can start typing φ with an unknown type for X (a type variable). During the typing, the value of this type variable can get constrained (typically, if $\cup$ is used it forces the two types to be equal). Then when we finish there are three possibilities: either t does not exist (the constraints are incompatible with each other), meaning the whole term is not typable; or t is entirely determined and we computed it; or the constraints are not enough to determine t entirely, meaning that the term is actually typable with different values for t. In the last case, it means that the result of the fixpoint term is always empty. Indeed, the type of a relation is the common domain of all its mappings; it is unique unless the relation is empty.

Given a database schema Γ, we write $\mathcal{F}[\Gamma]$ for the set of well-typed terms in Γ (i. e. the terms φ such that $\Gamma \vdash \varphi : t$ holds for some t).

Proposition 1. *Given a database $(\mathcal{R}, \Gamma, D)$ and $\varphi \in \mathcal{F}[\Gamma]$, if $\Gamma \vdash \varphi : t$ then the relation $[\![\varphi]\!]_D$ has type t.*

Example 1. *Let us suppose that we want to compute the transitive closure of R of type $\{a, b\}$. The closure is captured by the term $\mu\Big(X = R \cup \widetilde{\pi}_m \left(\rho^m_b (R) \bowtie \rho^m_a (X)\right)\Big)$ of type $\{a, b\}$.*

Indeed, the fixpoint should have the type of its constant part which is R of type $\{a, b\}$. Then we can check the type of the non-constant part: $\rho^m_b (R)$ has type $\{a, m\}$ and is joined with $\rho^m_a (X)$ of type $\{b, m\}$. The result has type $\{a, b, m\}$ but the m column is discarded by the $\widetilde{\pi}_m (\dots)$.

3.5 Restrictions on fixpoints

Our syntax for the μ-RA is very general and comprises some counter-intuitive fixpoints (e.g. mixing negation and negation) and some types of fixpoints that are hard to optimize (e.g. non linear & mutually recursive). We present restrictions on fixpoints that are needed for the validity of our rewrite rules and of most of our propositions and theorems.

In the sequel, we suppose that all fixpoints abide the restrictions that we present here. This does not mean, however, that our method can not be applied on general terms: given a general term φ that contains a subterm ψ, if ψ abides the restrictions then we can apply our rewrite rules on ψ.

3.5.1 Properties of fixpoints.

Definition 5. *Given a term φ, we say that φ is* constant *in X when X is not a free variable of φ.*

Definition 6. *A fixpoint term $\mu(X = \varphi)$ is said:*

- positive *when for all subterms $\varphi_1 \triangleright \varphi_2$ of φ, φ_2 is constant in X;*
- linear *when for all subterms of φ of the form $\varphi_1 \bowtie \varphi_2$ or $\varphi_1 \triangleright \varphi_2$, either φ_1 or φ_2 is constant in X;*
- mutually recursive *when there exists a subterm $\mu(Y = \psi)$ of φ with X free in ψ.*

Proposition 2. *If $\mu(X = \varphi)$ of type t is linear, positive and non mutually recursive then the function $f(S) = [\![\varphi]\!]_{V[X/S]}$ (for S a set of mappings of type t) is such that:*

$$f(S) = f(\emptyset) \cup \bigcup_{x \in S} f(\{x\})$$

and thus f has a fixpoint with $[\![\mu(X = \varphi)]\!]_V = f^\infty(\emptyset)$.

Proof sketch. We prove by induction on the subterms ξ of φ where X is free in ξ that $[\![\xi]\!]_{V[X/S]} = [\![\xi]\!]_{V[X/\emptyset]} \cup \bigcup_{x \in S} [\![\xi]\!]_{V[X/\{x\}]}$. By linearity, such a ξ can only be combined with a constant term and by positivity, it cannot be negated. □

3.5.2 Expressivity of restricted fixpoints.

These restrictions do have an effect on the expressivity of our language. We can show (see appendix B) that μ-RA has, at least, the expressive power of inflationary-Datalog$^\neg$ (Datalog with inflationary semantics and negation). When we restrict fixpoints to be positive and non mutually recursive then our language has exactly the expressive power of stratified-Datalog$^\neg$. Finally with all our restrictions, our language has exactly the expressive power of linear datalog (see § 2.2). This fragment, however, does contain a lot of interesting queries: for instance, the next section presents how to translate UCRPQ into μ-RA (with the restrictions).

As mentioned previously, our method can be applied on general terms but, for the sake of simplicity, in the sequel we will only consider the fragment *rest-μ-RA* of μ-RA containing only terms where all the fixpoints are **linear, positive and non mutually recursive**.

3.6 Decomposed fixpoints

Once our terms are restricted, we see that fixpoints can actually be decomposed into a strictly *recursive* part and a *constant* part. This decomposition will be later useful for expressing some of our rewrite rules.

Definition 7. *Given a term φ linear and positive in X, we say that φ is recursive in X when $rec(\varphi, X) = \top$ with rec defined as:*

$$\begin{array}{rcl}
rec(\varphi_1 \cup \varphi_2, X) & = & rec(\varphi_1, X) \land rec(\varphi_2, X) \\
rec(\varphi_1 \bowtie \varphi_2, X) & = & rec(\varphi_1, X) \lor rec(\varphi_2, X) \\
rec(\varphi_1 \triangleright \varphi_2, X) & = & rec(\varphi_1, X) \\
rec(\sigma_{\mathfrak{f}} (\varphi), X) & = & rec(\varphi, X) \\
rec(\widetilde{\pi}_a (\varphi), X) & = & rec(\varphi, X) \\
rec(\rho^b_a (\varphi), X) & = & rec(\varphi, X) \\
rec(\mu(Y = \varphi), X) & = & \bot \\
rec(X, Y) & = & X = Y \\
rec(|c \to v|, X) & = & \bot
\end{array}$$

Being recursive or constant (def. 5) are syntactical properties. However the two following propositions give a semantic interpretation of those syntactical properties.

Lemma 1. *Let φ be a term.*

- *If φ is recursive in X then for all V, $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$.*
- *If φ is constant in X, then φ does not depend on X, i.e. for all S and V, $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$.*

Definition 8. *A fixpoint term $\mu(X = \kappa \cup \psi)$ is said* decomposed *when κ is constant in X and ψ is recursive in X.*

Example 2. *The term* $\mu\Big(X = R \cup \widetilde{\pi}_m \left(\rho_b^m (R) \bowtie \rho_a^m (X)\right)\Big)$ *of Example 1 is a* decomposed fixpoint*: R is constant and* $\widetilde{\pi}_m \left(\rho_b^m (R) \bowtie \rho_a^m (X)\right)$ *is recursive in X.*

Proposition 3. *A fixpoint term $\mu(X = \varphi)$, linear, positive and non mutually recursive can be rewritten to either: an empty term, a term φ with one less fixpoint or a decomposed fixpoint.*

Remark 1. *The type of a decomposed fixpoint term is the type of its constant part.*

In fact, for decomposed fixpoints the last typing rule of Fig. 2 can be replaced by the following:

$$\frac{\Gamma \vdash \kappa : t \qquad \Gamma \cup \{X \to t\} \vdash \psi : t}{\Gamma \vdash \mu(X = \kappa \cup \psi) : t}$$

which removes the need to guess t or use a type variable.

4 GENERATING NEW QUERY PLANS

4.1 Motivation

The traditional RA has rewrite rules and the optimization of RA queries is usually done by rewriting to a (estimated) more efficient term using rewrite rules. In this section, we discuss properties of *rest-μ*-RA which allow us to introduce new rewrite rules, specific to terms with fixpoints. These rules are an addition to the classical rewrite rules of the RA, which are all valid on *rest-μ*-RA as well.

We first describe our four new rewrite rules informally; then in the following subsections we discuss the conditions under which these rules are valid.

4.1.1 Pushing filters into fixpoints:

$$\sigma_{\mathfrak{f}} \left(\mu(X = \varphi)\right) \to \mu(X = \sigma_{\mathfrak{f}} (\varphi))$$

(see Theorem 1). This always reduces the amount of mappings manipulated by the fixpoint. This rule is typically useful on *e.g.* RPQs such as $R^+(a, b)$ where a is a constant to force the evaluation to only compute the (a, b) where b is reachable from this a and not compute the whole R^+ before filtering the pairs with the wrong a.

4.1.2 Pushing joins into fixpoints:

$$\mu(X = \varphi) \bowtie \psi \to \mu(X = \varphi \bowtie \psi)$$

(see Theorem 3). This can also lower the number of mappings solutions of the fixpoint. This rewrite rule can be used on RPQs such as R_1^+/R_2 where the naive translation would compute the whole relations R_1^+ and R_2 before joining them. With this rewrite rule, it would start from R_1/R_2 and at each iteration prepend a R_1. This would be typically useful when R_1/R_2 is smaller than just R_1 (e.g. the right side of R_2 might be a constant).

4.1.3 Merging fixpoints:

$$\mu(X = \varphi \cup \psi(X)) \bowtie \mu(X = \kappa \cup \xi(X)) \\ \to \mu(X = \varphi \bowtie \kappa \cup \psi(X) \cup \xi(X))$$

(see Theorem 4). This limits the number of fixpoints but also reduces the amount of mappings. This rewrite rule can be used on RPQs of the form R_1^+/R_2^+ similarly to the rule above: we compute R_1/R_2 and then at each step we either prepend R_1 or append R_2. This rule is also useful at the UCRPQ level, for instance to compute the combination of $R_1^+(a, b)$,$R_2^+(a, c)$,$R_3^+(a, d)$. By applying our rewrite rule twice, we obtain a term that computes $R_1(a, b)$, $R_2(a, c)$, $R_3(a, d)$ and at each iteration replace in the quadruplet (a, b, c, d) either b with b' (with $R_1(b, b')$) or c with c' (with $R_2(c, c')$), or d with d' (with $R_3(d, d')$).

4.1.4 Pushing antiprojections into fixpoints: $\widetilde{\pi}_p \left(\mu(X = \varphi)\right) \to \mu\Big(X = \widetilde{\pi}_p (\varphi)\Big)$ (see Theorem 5). This can reduce the number of mappings since as the value of the removed column is ignored, several mappings might get merged. For instance, on RPQs of the form (R_1^+/R_2) where the right value is discarded.

We now discuss formally the conditions which allow these rewritings. All the proofs are in [12].

4.2 Image of a variable through a term

Given a term φ linear and positive in a variable X, we can compute a set of derivations from X for φ. And if w is a mapping then each mapping of $[\![\varphi]\!]_{V[X/\{w\}]}$ either belongs to $[\![\varphi]\!]_{V[X/\emptyset]}$ or is obtained using one of those derivations.

If we do not have $\sigma_{\mathfrak{f}} \left(\mu(X = \varphi)\right) \equiv \mu(X = \sigma_{\mathfrak{f}} (\varphi))$ in general, it is because some mappings solution of $\mu(X = \varphi)$ might not pass the filter but still be useful to create mappings (with the fixpoint iteration) passing the filter condition. The study of those derivations will allow us to infer that e.g., in some circumstances, not passing the filter condition is something that will stay invariant by one iteration of the fixpoint, in which case we will have $\sigma_{\mathfrak{f}} \left(\mu(X = \varphi)\right) = \mu(X = \sigma_{\mathfrak{f}} (\varphi))$.

4.2.1 Logical framework.

Definition 9. *The set of derivations $d(\varphi, X)$ is:*

$$\begin{array}{lcl}
d(\varphi_1 \cup \varphi_2, X) & = & d(\varphi_1, X) \cup d(\varphi_2, X) \\
d(\varphi_1 \triangleright \varphi_2, X) & = & d(\varphi_1, X) \\
d(\varphi_1 \bowtie \varphi_2, X) & = & d(\varphi_1, X) \cup d(\varphi_2, X) \\
d(\rho_a^b (\varphi), X) & = & \{p \circ (b \to a, a \to \bot) \mid p \in d(\varphi, X)\} \\
d(\widetilde{\pi}_a (\varphi), X) & = & \{p \circ (a \to \bot) \mid p \in d(\varphi, X)\} \\
d(\sigma_{\mathfrak{f}} (\varphi), X) & = & d(\varphi, X) \\
d(\mu(Y = \varphi), X) & = & \emptyset \\
d(X, X) & = & \{()\} \text{ (a singleton identity)} \\
d(X, R) & = & \emptyset \\
d(|c \to v|, X) & = & \emptyset
\end{array}$$

Where $\circ$ represents the composition and $(a_1 \to b_1, \ldots, a_n \to b_n)$ represents the function that maps each a_i to its b_i and every *other column name to itself. Note that this definition manipulates functions with an infinite domain but the domain where they do not coincide with the identity is finite and they are thus computable.*

Example 1 followup. In our previous example, X appears only once and thus there is only one derivation that maps $a \to \bot, m \to \bot$ and everything else to itself. In particular b is mapped to itself.

Lemma 2. *Let w be a mapping and φ a term linear, positive and non mutually recursive in X. For all $m \in [\![\varphi]\!]_{V[X/\{w\}]}$ either $m \in [\![\varphi]\!]_{V[X/\emptyset]}$ or there exists $p \in d(\varphi, X)$ such that for all $c \in dom(w)$:*

$$\Big(p(c) = \bot\Big) \vee \Big(p(c) \notin dom(w)\Big) \vee \Big(m(c) = w\big(p(c)\big)\Big)$$

Definition 10. *Given a term φ linear and positive in a variable X, we define the* stabilizer *of X in φ as the following set of column names: $stab(\varphi, X) = \{c \in \mathfrak{C} \mid \forall p \in d(\varphi, X)\ p(c) = c\}$*

Lemma 3. *Given a fixpoint term $\mu(X = \varphi) \in \mathcal{F}[\Gamma]$ of type t and a mapping of type t, $w \in [\![\mu(X = \varphi)]\!]_V$ if and only if we can find a* lineage *$w_0, \ldots, w_n$ for w, that is $w_0, \ldots, w_n$ such that $w_0 \in [\![\varphi]\!]_{V[X/\emptyset]}$ and $w_{i+1} \in [\![\varphi]\!]_{V[X/\{w_i\}]}$.*

Furthermore for all lineages $w_0, \ldots, w_n$ and all $c \in t \cap stab(\varphi, X)$, we have $w_0(c) = w(c)$.

4.2.2 Application to rewrite rules.

Theorem 1 (Pushing filters). *Let $\mu(X = \varphi)$ be a fixpoint term, V an environment and f a filter condition with $FC(f) \subseteq stab(\varphi, X)$. Then we have $[\![\sigma_f(\mu(X = \varphi))]\!]_V = [\![\mu(X = \sigma_f(\varphi))]\!]_V$, and if $\mu(X = \varphi)$ can be decomposed into $\mu(X = \kappa \cup \psi)$, we also have $[\![\sigma_f(\mu(X = \kappa \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\kappa) \cup \psi)]\!]_V$.*

Proof sketch. This is a consequence of lemma 3: we can filter the lineage on w or on w_0 but they have equal values on $FC(f)$ and by definition of FC, $eval(f, w_0) = eval(f, w)$. □

Theorem 2 (Pushing anti-joins). *Let $\mu(X = \varphi)$ be a fixpoint term, V an environment and ψ a term of type $t \subseteq stab(\varphi, X)$ (we suppose that X is not a free variable of ψ). Then we have $[\![\mu(X = \varphi) \triangleright \psi]\!]_V = [\![\mu(X = \varphi \triangleright \psi)]\!]_V$, and if $\mu(X = \varphi)$ can be decomposed into $\mu(X = \kappa \cup \xi)$, we also have $[\![\mu(X = \kappa \cup \xi) \triangleright \psi]\!]_V = [\![\mu(X = \kappa \triangleright \psi \cup \xi)]\!]_V$.*

Proof sketch. The anti join will act in very similar way to a filter since ψ is constant in X. Lineages $w_0 \ldots w_n$ will preserve the property to be compatible with one of the elements of $[\![\psi]\!]_V$. □

4.3 Adding columns to fixpoints

In § 4.2 we have seen that the set of derivation describe what columns stay unchanged after one iteration of the fixpoints. We now study what happens when we change the type of a fixpoint by, e.g. adding or removing one column.

4.3.1 Logical framework.

Recall from Proposition 3 that, in *rest-μ*-RA, all fixpoints can be decomposed into the form $\mu(X = \kappa \cup \psi)$ with κ constant and ψ recursive in X. In this section, we only consider fixpoints rewritten in such a way, and we use the typing rule for decomposed fixpoints described in Remark 1.

So, let us assume a fixpoint $\varphi = \mu(X = \kappa \cup \psi)$ such that $\Gamma \vdash \varphi : t$. t is thus the type of κ. Suppose we change the constant part κ in such a way that it has now type $t' \neq t$. We want to ensure that the columns that t' adds or removes with respect to t play no role in the recursive computation ψ. The first requirement is of course that the modified fixpoint stays typable (we must have $\Gamma \vdash \mu(X = \kappa' \cup \psi) : t'$), but it is not enough because the columns might play a role in an antijoin, which does not show in the type. We thus introduce the following predicate:

Definition 11. *We say that a column $c \in \mathfrak{C}$ can be* added *to or removed from a term $\psi \in \mathcal{F}[\Gamma]$ recursive in X when $add(\psi, X, c) = \top$ holds, with add defined as:*

$$\begin{array}{lcl}
add(\varphi_1 \cup \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\
add(\varphi_1 \bowtie \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\
add(\varphi_1 \triangleright \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\
add(\rho_a^b(\varphi), X, c) & = & add(\varphi, X, c) \wedge c \notin \{a, b\} \\
add(\widetilde{\pi}_a(\varphi), X, c) & = & add(\varphi, X, c) \text{ when } c \neq a \\
add(\widetilde{\pi}_c(\varphi), X, c) & = & X \notin free(\varphi) \\
add(\sigma_f(\varphi), X, c) & = & add(\varphi, X, c) \wedge c \notin FC(f) \\
add(\mu(Y = \varphi), X, c) & = & add(\varphi, X, c) \\
add(R, X, c) & = & c \notin \Gamma(R) \text{ when } X \neq R \\
add(X, X, c) & = & \top \\
add(|c' \to v|, X, c) & = & c \neq c'
\end{array}$$

Lemma 4. *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t, let $c \in (\mathfrak{C} \setminus t)$ that can be added to ψ, and w a mapping of type t. We note $w(v) = w \cup \{c \to v\}$.*

If $\forall R \in \mathcal{R}, c \notin \Gamma(R)$, then we have:

- $c \in stab(\psi, X)$
- $\Gamma \cup \{X \to t \cup \{c\}\} \vdash \psi : t \cup \{c\}$
- $[\![\psi \bowtie |c \to v|]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\{w(v)\}]}$

Proof sketch. The first point can be proved inductively by definition of $stab$ and add. The second point is a consequence of the first. The third point can be proved by induction on the size of ψ. □

4.3.2 Application to rewrite rules.

Theorem 3 (Pushing joins). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ and $\varphi \in \mathcal{F}[\Gamma]$ (with $X \notin free(\varphi)$) a term of type t_φ such that:*

(1) $t_\varphi \subseteq stab(\psi, X)$
(2) $\forall c \in t_\varphi \setminus t_\kappa\ add(\psi, X, c)$

Then we have $\Gamma \vdash \mu(X = \kappa \bowtie \varphi \cup \psi) : t_\varphi \cup t_\kappa$ with for all V compatible with Γ:

$$[\![\varphi \bowtie \mu(X = \kappa \cup \psi)]\!]_V = [\![\mu(X = \kappa \bowtie \varphi \cup \psi)]\!]_V$$

Proof sketch. First we prove that $\psi \in \mathcal{F}[\Gamma \cup \{X \to t_\kappa \cup t_\varphi\}]$ by iterating Lemma 4.1. Then for each lineage $w_0, \ldots, w_n$ of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ and each v compatible with w_0 we can build a lineage $(w_0 + v), \ldots, (w_n + v)$ (by iteration on Lemma 4), which proves $w + v \in [\![\mu(X = \varphi \bowtie \kappa \cup \psi)]\!]_V$.

The reverse direction is proven the same manner. □

Example 3. *If we want to compute $R_1^+(x,y) \wedge R_2(y,z)$, the naive translation would compute R_1^+ and then join with R_2. But our approach also considers the plan where we start from x, y, z such that $R_2(y,z) \wedge R_1(x,y)$ and then will discover new x by a fixpoint: $\mu\Big(X = R_1 \bowtie R_2 \cup \widetilde{\pi}_c \left(\rho_x^c(X) \bowtie \rho_y^c(R_1)\right)\Big)$*

Theorem 4 (Merging fixpoints). *Given two decomposed fixpoints $\mu(X = \kappa_1 \cup \psi_1)$ and $\mu(X = \kappa_2 \cup \psi_2)$ of types t_1 and t_2 such that:*

(1) $t_1 \cap t_2 \subseteq stab(\psi_2, X, C_2) \cap stab(\psi_1, X, C_1)$
(2) $\forall c \in t_1 \setminus t_2 \; add(\psi_2, X, c)$
(3) $\forall c \in t_2 \setminus t_1 \; add(\psi_1, X, c)$

then we have: $[\![\mu(X = \kappa_1 \cup \psi_1) \bowtie \mu(X = \kappa_2 \cup \psi_2)]\!]_V = [\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$.

Proof sketch. The forward direction is easy: given two lineages $w_0^1, \ldots, w_n^1$ and $w_0^2, \ldots, w_m^1$ (for both $[\![\mu(X = \kappa_i \cup \psi_i)]\!]_V$) we can build a lineage $(w_0^1 + w_0^2) \ldots (w_0^1 + w_m^2) \ldots (w_n^1 + w_m^2)$.

The converse direction is more difficult but we can de-interlace the lineages and create two lineages, one for each $[\![\mu(X = \kappa_i \cup \psi_i)]\!]_V$ □

Example 4. *If we want to compute $R_1^+(x,y) \wedge R_2^+(y,z)$, the naive translation would compute both R_1^+ and R_2^+. But our approach also considers the plan where we start from x, y, z such that $R_2(y,z) \wedge R_1(x,y)$ and then will discover new x or new z by a fixpoint: $\mu(X = R_1 \bowtie R_2 \cup \psi)$ with $\psi = \widetilde{\pi}_c \left(\rho_x^c(X) \bowtie \rho_y^c(R_1)\right) \cup \widetilde{\pi}_c \left(\rho_z^c(X) \bowtie \rho_y^c(R_2)\right)$.*

Theorem 5 (Pushing antiprojections). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ. Let $b \in \mathfrak{C}$ be such that $add(\psi, X, b)$. Then:*

$$[\![\widetilde{\pi}_b(\mu(X = \kappa \cup \psi))]\!]_V = [\![\mu(X = \widetilde{\pi}_b(\kappa) \cup \psi)]\!]_V$$

Proof sketch. This property can be proven via lineages similarly to the proofs of the other theorems. □

5 GRAPH QUERY TRANSLATIONS

μ-RA can be used to model queries over directed graphs with labeled edges. We assume that the set of values $\mathfrak{V}$ gathers both vertices and edge labels. The graph can then be represented as a pair $(\mathcal{V}, \mathcal{E})$ with $\mathcal{V} \subset \mathfrak{V}$ denoting the set of vertices and $\mathcal{E} \subset \mathcal{V} \times \mathfrak{V} \times \mathcal{V}$ denoting the set of edges. This can be modeled as a relational database with two relations V and E representing these two sets, with the schema $\Gamma = \{V \to \{\mathtt{src}\}, E \to \{\mathtt{src}, \mathtt{l}, \mathtt{trg}\}\}$ where src, l, trg stand respectively for *source*, *label*, *target*[2].

Regular path queries (RPQs) [3, 22, 26, 30, 48] provide a basic construct used in graph query languages. An RPQ makes it possible to express a path connecting graph nodes by the means of regular expressions over edge labels. We consider a set W of query variables, and a set $K \subseteq \mathfrak{V}$ of vertex labels (constants). The general syntax of an RPQ is $r(x,y)$ where $x \in W \cup K$ is connected to $y \in W \cup K$ by the regular path expression r defined as follows:

$$\begin{array}{rrr} r \quad ::= & & \\ & v & \text{a single edge label} \\ | & r_1/r_2 & \text{concatenation} \\ | & r_1|r_2 & \text{alternative} \\ | & r^{-1} & \text{reverse} \\ | & r^+ & \text{transitive closure} \end{array}$$

For example, the sample query Q_2 given in the introduction is an RPQ. The basic component for translating graph queries into μ-RA is the translation of the regular path expressions r. We translate any r into a set of μ-RA terms ϕ, representing one or more alternative translations, in such a way that the result of evaluating any of those ϕ on the graph database is the set of all mappings $\{\mathtt{src} \to v_1, \mathtt{trg} \to v_2\}$ such that the sequence of labels in the path from v_1 to v_2 matches r. For this purpose, we define a translation function $(\!|\cdot|\!)$ that compiles any r into μ-RA, as follows:

$$\begin{array}{rcl}
(\!|v|\!) & = & \{\widetilde{\pi}_1(\sigma_{1=v}(E))\} \\
(\!|r_1/r_2|\!) & = & \left\{\widetilde{\pi}_m\left(\rho_{\mathtt{trg}}^m(\phi_1) \bowtie \rho_{\mathtt{src}}^m(\phi_2)\right) \mid \phi_1 \in (\!|r_1|\!) \wedge \phi_2 \in (\!|r_2|\!)\right\} \\
(\!|r_1|r_2|\!) & = & \{\phi_1 \cup \phi_2 \mid \phi_1 \in (\!|r_1|\!) \wedge \phi_2 \in (\!|r_2|\!)\} \\
(\!|r^{-1}|\!) & = & \left\{\rho_m^{\mathtt{src}}\left(\rho_{\mathtt{src}}^{\mathtt{trg}}\left(\rho_{\mathtt{trg}}^m(\phi)\right)\right) \mid \phi \in (\!|r|\!)\right\} \\
(\!|r^+|\!) & = & \left\{\mu\Big(X = \phi \cup \widetilde{\pi}_m\left(\rho_{\mathtt{trg}}^m(\phi) \bowtie \rho_{\mathtt{src}}^m(X)\right)\Big) \mid \phi \in (\!|r|\!)\right\} \cup \\
 & & \left\{\mu\Big(X = \phi \cup \widetilde{\pi}_m\left(\rho_{\mathtt{src}}^m(\phi) \bowtie \rho_{\mathtt{trg}}^m(X)\right)\Big) \mid \phi \in (\!|r|\!)\right\}
\end{array}$$

Notice that for r^+ we have two equivalent translated terms. This is because we can choose to rename src to m either in r or in X (and correspondingly trg to m in the other), depending on the direction we want to follow when recursively navigating the graph. We want to keep track of both translations, as this has impact on the plan space generation[3]. This is the reason why $(\!|r|\!)$ returns a set of terms and not a single term. This translation of regular path expressions constitutes the main component used for translating graph query languages such as UCRPQs.

[2]The choice of the column name src for V is arbitrary (trg could have been chosen instead), it simply avoids some renamings).

[3]This is because rewrite rules presented in § 4 apply differently over each initial translation, generating two different plan spaces, and we want to explore the union of the two.

A CRPQ is of the form $H \leftarrow C$, where the query head H is a non-empty set of vertex variables to be extracted by the query, and C is a conjunction of RPQs that describes how those vertex variables are connected to other vertex variables or to constants. More formally:

- H is of the form $(z_1, \ldots, z_m)$ with arity $m > 0$ (we do not consider boolean queries)
- C is a conjunction of the form $r_1(x_1, y_1), \ldots, r_n(x_n, y_n)$ where:
  - $x_1, y_1, \ldots x_n, y_n \in W \cup K$
  - each r_i is an RPQ (as defined previously)
  - for each $0 < k \leq m$ we have $z_k \in \{x_1, y_1, \ldots x_n, y_n\} \setminus K$.

UCRPQs extend CRPQs with union at top level. They have the syntax $H \leftarrow C_1 \cup \ldots \cup C_n$ in which each disjunct C_i is a conjunction as defined previously. We translate a UCRPQ into μ-RA as follows: $(\!|H \leftarrow C_1 \cup C_2 \ldots \cup C_n|\!) = (\!|C_1|\!)_H \cup (\!|C_2|\!)_H \ldots \cup (\!|C_n|\!)_H$ with:

$(\!|C|\!)_H = \{\phi_1 \bowtie \phi_2 \bowtie \ldots \bowtie \phi_n \mid (\varphi_1, \varphi_2, \ldots, \varphi_n) \in \texttt{combine}\,(C)_H\}$

$$\texttt{combine}\,(r_1(x_1, y_1), \ldots, r_n(x_n, y_n))_H = (\!|r_1(x_1, y_1)|\!)_H \times \ldots \times (\!|r_n(x_n, y_n)|\!)_H$$

$$(\!|r(x,y)|\!)_H = \left\{\Pi\left(\theta^x_{\texttt{src}}\left(\theta^y_{\texttt{trg}}(\varphi)\right)\right)_H \mid \varphi \in (\!|r|\!)\right\}$$

$$\theta^x_c(\varphi) = \begin{cases} \rho^x_c(\varphi) & \text{for } c \in W \\ \sigma_{c=x}(\varphi) & \text{for } x \in K \end{cases} \quad \text{in which}$$

$\Pi(\varphi)_H = \widetilde{\pi}_{x_1}(\widetilde{\pi}_{x_2 \ldots}(\widetilde{\pi}_{x_n}(\varphi)))$ where $x_1, x_2, \ldots, x_n$ occur in φ but not in H (we keep only columns corresponding to selected variables).

Fig. 4 presents an optimized plan ($\mathcal{P}_5$) of the introductory query `?x isLocatedIn+/dealsWith+ Japan`, in which fixpoints have been merged, with the transformations of § 4.

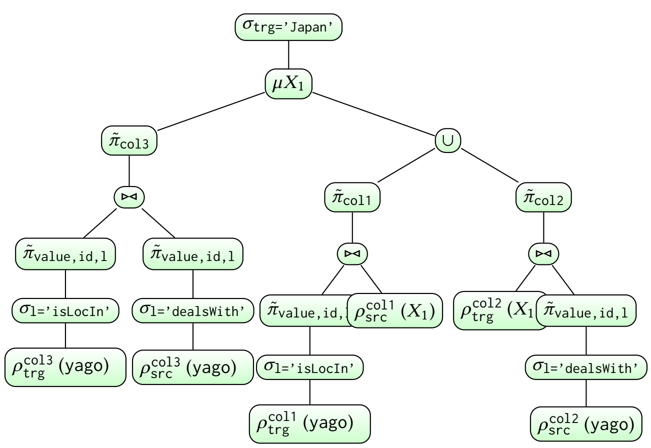

Figure 4: QEP $\mathcal{P}_5$ in which fixpoints have been merged.

6 EXPERIMENTS

6.1 Prototype

We implemented a query optimizer prototype based on the previously described μ-RA. This prototype extends well-known algebraic optimization rules with the new rules for recursive expressions described in § 4.1. The prototype parses queries written in UCRPQ notation and compiles them into μ-RA terms. μ-RA terms are then optimized by applying the extended set of optimization rules using a Volcano-style term enumeration strategy [35]. The prototype implements a cost model à la Selinger et al. used to select an estimated best plan from the plan space, based on estimated evaluation costs [55]. The prototype also translates μ-RA terms into SQL (some rewritten terms also use PL/pgSQL, but this is not the case of initial translations). The translation of the estimated best plan is then passed to PostgreSQL for evaluation. Our prototype thus extends PostgreSQL with the exploration of new plans not considered in the standard implementation, that are likely to be more efficient for recursive queries.

6.2 Methodology

For shedding light on the practical interest of computing richer plan spaces, we experimentally compared the performance of graph query evaluation with and without our optimizations. Our methodology is twofold: first, we tested a set of realistic queries evaluated over a real-world dataset; and we also used the gMark benchmark [14] to randomly generate UCRPQs with corresponding synthetic datasets.

6.2.1 Queries over the YAGO dataset. We use a cleaned version of the real world dataset YAGO2s [31], that we have preprocessed in order to remove duplicate triples and keep only triples with existing and valid identifiers. After preprocessing, we obtain a table of YAGO facts with 83 predicates and 62,643,951 rows (graph edges). We collected third-party queries previously considered against this popular dataset from the literature: we collected 7 RPQs from [5], 2 from [64] and 2 from [39]. Since all of them are RPQs, we supplemented them with additional CRPQs to illustrate more complex forms of recursion. Fig. 5 presents the 20 queries tested with the YAGO dataset.

6.2.2 Generated queries over synthetic datasets, using gMark. We gathered queries generated by gMark [14], which are available on the gMark open source repository [15]. We filtered them to retain only queries in which at least one recursion was present. For queries generated with an empty head, we replaced the empty head by a head containing all the variables occurring in the body (so as to test a more complex polyadic variant of the query instead of its simple boolean counterpart). This provided us with 12 recursive queries for the "UniProt" gMark scenario, that we evaluated

```
?x        ←  ?x isMarriedTo/livesIn/IsL+/dw+ Argentina          Q1
?x        ←  ?x hasChild/livesIn/IsL+/dw+ Japan                 Q2
?x        ←  ?x influences/livesIn/IsL+/dw+ Sweden              Q3
?x        ←  ?x livesIn/IsL+/dw+ United_States                  Q4
?x        ←  ?x hasSuccessor/livesIn/IsL+/dw+ India             Q5
?x        ←  ?x hasPredecessor/livesIn/IsL+/dw+ Germany         Q6
?x        ←  ?x haa/livesIn/IsL+/dw+ Netherlands                Q7

?x        ←  ?x IsL+/dw+ United_States                          Q8
?x        ←  ?x (actedIn/-actedIn)+ Kevin_Bacon                 Q9

?area     ←  wce -type/(IsL+/dw|dw) ?area                       Q10
?person   ←  ?person isMarriedTo+/owns/IsL+|owns/IsL+ USA       Q11

?a,?b     ←  ?a IsL+/dw ?b                                      Q12
?a,?b     ←  ?a IsL+/dw+ ?b                                     Q13
?a,?b,?c  ←  ?a wasBornIn/IsL+ ?b, ?b isConnectedTo+ ?c         Q14
?a,?b,?c  ←  ?a (IsL|isConnectedTo)+ ?b, ?a wasBornIn ?c        Q15
?a,?b,?c  ←  ?a wasBornIn/IsL+ Japan, ?b isConnectedTo+ ?c      Q16
?a        ←  ?a IsL+/(isConnectedTo|dw)+ Japan                  Q17
?a,?c     ←  ?a IsL+ Japan, ?a isConnectedTo+ ?c                Q18
?a        ←  ?a IsL+/IsL Japan                                  Q19
?a        ←  ?a IsL+/isConnectedTo+/dw+ Japan                   Q20
```

Figure 5: Queries for the YAGO dataset[a].

[a] $Q_1 \dots Q_7$ are taken from [5], Q_8, Q_9 from [64], and Q_{10}, Q_{11} from [39]. "isL" stands for "IsLocatedIn", "dw" for "dealsWith", "haa" for "hasAcademicAdvisor", and "wce" for "wikicat_Capitals_in_Europe".

on a gMark-generated graph instance having 76,707 edges. We also carried out similar tests with the "Shop" gMark scenario for which we report on the evaluation of 14 queries over a synthesized graph instance with 209,789 edges. Table 1 presents the sizes of the considered datasets.

Dataset Origin	Predicates	Edges	Nodes
YAGO 2.5 (cleaned) [31]	83	62,643,951	42,832,856
gMark-Shop [14]	81	209,789	135,737
gMark-Uniprot [14]	7	76,707	21,130

Table 1: Dataset Statistics.

6.3 Tested Systems

We compared the query evaluation performance of the newly obtained recursive query plans with state-of-the-art systems implementing previously known approaches for recursive queries, namely:

- system **P**: the popular PostgreSQL open-source relational database [52, 58] implementing SQL with recursive views.
- system **P**': our prototype extending the PostgreSQL system with our optimizations;
- system **V**: the Virtuoso graph column store [28] which is backed by a relational database, and that implements the SPARQL 1.1 language [40] (with property paths);
- system **L**: a modern engine implementing Datalog [61];
- system **N**: a native graph database implementing the openCypher graph query language [38, 51].

6.4 Experimental setup

Set semantics. Most systems implement both bag and set semantics, and use bag semantics by default. However, their implementation of recursion differs significantly, which causes some systems to retrieve more or less duplicates when compared to others. Therefore, we use set semantics.

Timeout. For each tested system, we set a maximum time of 30 min of computation for each query (after 30 min, we stop the computations and consider that this particular query evaluation is not feasible with the given system)[4].

Specific limitations. Some systems have inherent limitations or require specific configuration. System **N** supports recursion only around atomic patterns (e.g. `isLocatedIn+` is supported but `(actedIn/-actedIn)+` is not) so it can theoretically evaluate only a limited number of the considered queries. System **V** requires a maximum number of query results to be set. We increased the default bound of 5,000 to 10,000 so that it can retrieve the complete set of answers for most queries considered.

*Initial comparison baseline for system **P**.* A given graph query translates into several possible μ-RA forms, even before applying our rewrite rules (see § 5). For a fair comparison with system **P**, we use the internal cost estimation of **P** to discriminate them[5]. Thus, for a given query, the time reported for system **P** is the elapsed time for evaluating the query translation that **P** has itself chosen among other equivalent initial translations.

Reported metrics. We report on query evaluation performance, excluding (i) time spent for data preparation (e.g. for loading or computing indexes) and (ii) for query optimization (e.g. for generating the plan space – a task which can benefit from extensive research and various techniques found in the literature [36, 37, 44, 57], and which is beyond the scope of this paper). We thus mainly concentrate on query evaluation times. We also report on the number of query answers returned by each system using set semantics.

6.5 Results

6.5.1 Queries over the YAGO dataset. Fig. 6 shows the time spent with each system for evaluating each query of Fig. 5. The time scale is logarithmic. Whenever no time is reported for a given query and system, this means that either the system crashed within the first 30 minutes of computation for that query, or the computations were stopped after 30 minutes. In both cases, the evaluation of the query is considered unfeasible with that system.

[4] Experiments have been conducted on a server with 128 GB of RAM, 2 Intel Xeon E5-2630 v4 CPUs (2.20 GHz, 20 cores each) and 66 TB of 7200 RPM hard disk drives (SAS, RAID 5), running Ubuntu 16.04 LTS, Docker 18.09.7, and Docker images for each tested system, using the latest official public Docker image when available

[5] Specifically, for each initial translation, we get the 'total cost' value returned by a call to the "EXPLAIN" statement of system **P**. We then retain the translation of minimum estimated cost.

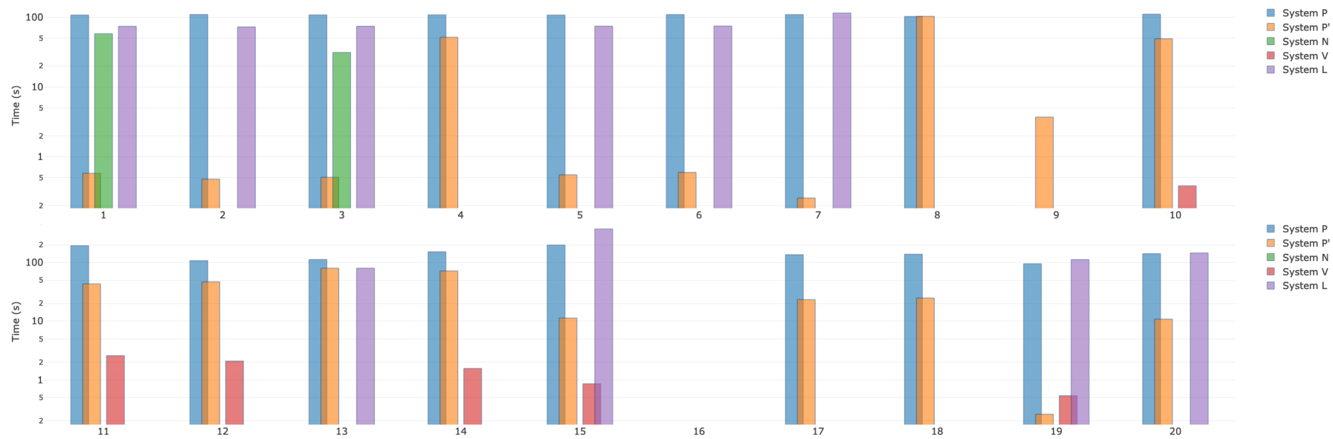

Figure 6: Elapsed time for evaluating queries of Fig. 5 over the YAGO real-world dataset.

Coverage. First, we observe significant discrepancies between the number of queries that each system has been able to answer. System **N** answered only 2 queries. System **V** answered 6 queries only. In comparison, system **P** answered many more queries: all but 2. System **P'** answered all queries but one (query 16) which was also unfeasible for other systems. The number of feasible queries for each system is summarized in Table 2. Fig. 7 presents the number of retrieved results (in logarithmic scale) for each query and each system. All the systems agree on the number of query results (using set semantics), except **V** with its bound on the number of results set to 10,000 (which is still not enough to retrieve the 2.4 million of results of query 12 retrieved by other systems).

	N	V	L	P	P'
Feasible queries	2	6	10	18	19
Unfeasible queries	18	14	10	2	1

Table 2: YAGO queries treated by each system.

Performance for feasible queries. For the few YAGO queries it covers, system **V** outperformed the other systems, except for of query 19 where it is outperformed by **P'**. We also observe that **P'** always performed more efficiently than **P**, in several cases by an order of magnitude. This suggests that the new plans computed by our approach sometimes represent much more efficient alternatives. In the case of query 9, the new plan selected even makes query answering possible whereas it is unfeasible with other systems. Table 3 summarizes the feasibility and performance gains brought by **P'** in comparison to other systems.

	P	N	V	L
Q_1	184.3	99.8	∞	127.0
Q_2	227.8	∞	∞	151.4
Q_3	212.2	61.5	∞	145.7
Q_4	2.1	∞	∞	∞
Q_5	194.4	∞	∞	134.8
Q_6	182.6	∞	∞	125.1
Q_7	424.3	∞	∞	446.7
Q_8	1.0	∞	∞	∞
Q_9	∞	∞	∞	∞
Q_{10}	2.2	∞	0.01	∞

	P	N	V	L
Q_{11}	4.5	∞	0.06	∞
Q_{12}	2.3	∞	0.04	∞
Q_{13}	1.4	∞	∞	1.0
Q_{14}	2.1	∞	0.02	∞
Q_{15}	17.6	∞	0.08	33.0
Q_{16}	⊥	⊥	⊥	⊥
Q_{17}	5.8	∞	∞	∞
Q_{18}	5.6	∞	∞	∞
Q_{19}	368.0	∞	2.07	433.9
Q_{20}	13.0	∞	∞	13.4

Table 3: Speedup with P' for YAGO queries[a].

[a] A speedup $x > 1$ means query evaluation is x times faster, $x < 1$ means slower, $x = 1$ means no speedup, ⊥ means that the query remains unfeasible, and ∞ denotes cases where a formerly unfeasible query becomes feasible.

6.5.2 *Generated queries over synthetic datasets.*

Fig. 8 and 9 show the times spent in evaluating recursive queries of the "UniProt" and the "Shop" gMark scenarios [14], respectively. No results are reported for system **N** because it supports none of the randomly generated queries (each one contains at least one form of recursion such as `(a/b)+` which is not supported by **N**). No query was feasible for system **L** neither (each query timed out even with a maximum time of 2 hours per query). System **V** performed well on the synthetic datasets. It is the most efficient system for 3 queries of the "UniProt" scenario and 4 queries of the "Shop" scenario. In the remaining majority of cases however, it is outperformed by systems **P** and **P'** (either in terms of feasibility or in terms of performance). System **V** can evaluate one query of the "Shop" scenario which is unfeasible for other systems, but on each scenario, systems **P** and **P'** make it possible to evaluate 3 more queries (unfeasible with **V**).

A few cases clearly illustrate the interest of the rich plan space explored by system **P'**. For instance for "uniProt" queries 3, 5, 6, 8, and 12, and "Shop" queries 2, 9, 10, and 13 of system **P'** performs much more efficiently thanks to the selected plan which was not present in the plan space of system **P**. At least 2 cases also illustrate that there exists room for improvement of the cost estimation function in charge of picking the best estimated plan in the search space. This is the case for

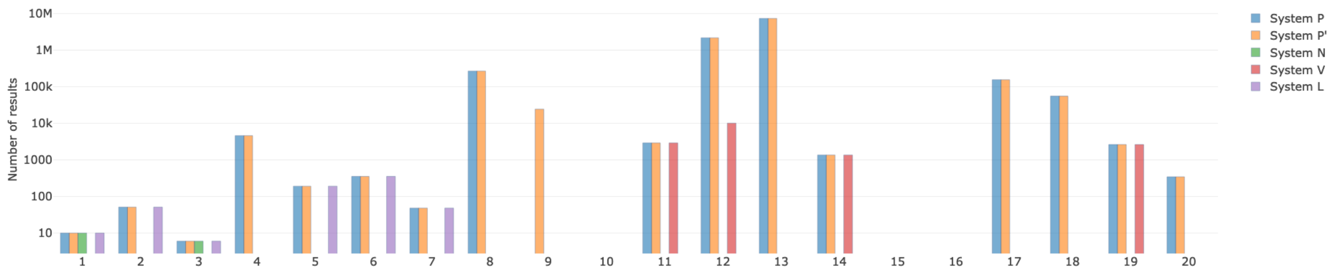

Figure 7: Number of results retrieved (with set semantics) by each query of Fig. 5 on the YAGO dataset.

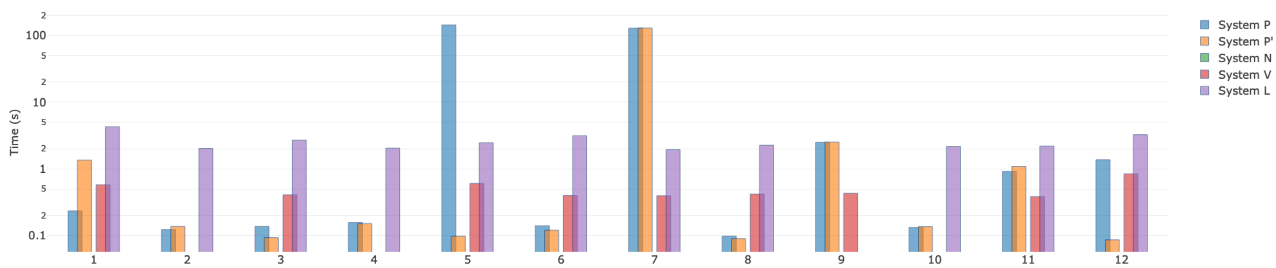

Figure 8: Time spent for evaluating recursive queries of the synthetic "UniProt" gMark scenario.

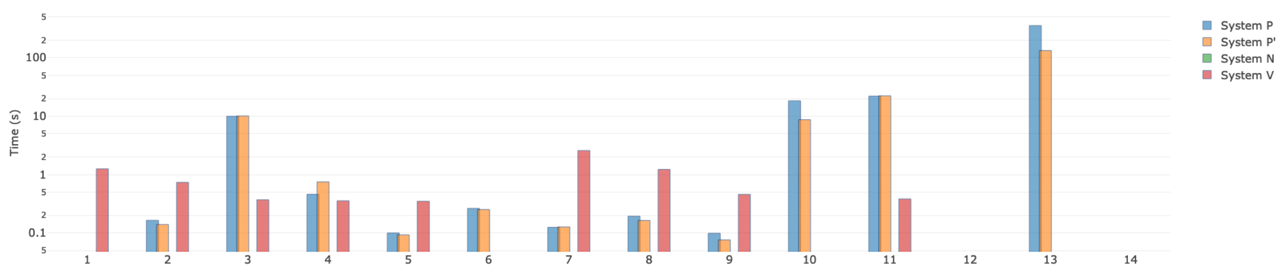

Figure 9: Time spent for evaluating recursive queries of the synthetic "Shop" gMark scenario.

uniProt query 1 and Shop query 4 for instance, where the plan picked by system **P**' reveals less efficient than the plan of system **P**. The term-picking function should thus ideally favor the initial plan, also present in the search space.

Overall, experiments suggest that the new plans computed by our approach can offer significant gains in computation time when evaluating recursive queries over graphs. They also suggest that some of the newly computed plans can allow to realize certain query evaluations which used to be unfeasible. This illustrates the benefits of the exploration of richer plan spaces made possible by μ-RA.

7 CONCLUSION

We propose a variation of the classical relational algebra extended with a fixpoint operator, which is useful for capturing recursive terms and for facilitating their transformations. We propose new rewriting rules for recursive terms. These new rules are compatible and compositional with existing rules for optimizing the core of the relational algebra. The extended set of optimization rules makes it possible to compute new query evaluation plans beyond reach with previous approaches. Our approach can be used within any mainstream database management system that implements SQL with recursion, either by adding the new rules inside the query optimizer, or as a preprocessing stage not requiring to modify the system's internals. Experiments with a prototype implementing such a preprocessing phase for the PostgreSQL system suggest that the new query plans can be useful for evaluating much more efficiently recursive queries over graphs. Interesting perspectives for future work include studying further extensions with aggregation and user-defined functions.

REFERENCES

[1] Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch, and Richard Cyganiak. 2017. Linking Open Data cloud diagram. http://lod-cloud.net/

[2] Serge Abiteboul, Richard Hull, and Victor Vianu (Eds.). 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. 1996. The Lorel Query Language for Semistructured Data. *Journal on Digital Libraries* 1, 1 (1996). http://ilpubs.stanford.edu:8090/162/

[4] Serge Abiteboul and Victor Vianu. 1991. Datalog Extensions for Database Queries and Updates. *J. Comput. Syst. Sci.* 43, 1 (Aug. 1991), 62–124. https://doi.org/10.1016/0022-0000(91)90032-Z

[5] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H. Chignell. 2017. TASWEET: Optimizing Disjunctive Path Queries in Graph Databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. OpenProceedings.org, 470–473. https://doi.org/10.5441/002/edbt.2017.47

[6] Foto Afrati, Manolis Gergatsoulis, and Francesca Toni. 2003. Linearisability on datalog programs. *Theoretical Computer Science* 308, 1 (2003), 199 – 226. https://doi.org/10.1016/S0304-3975(02)00730-2

[7] R. Agrawal. 1988. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering* 14, 7 (July 1988), 879–885. https://doi.org/10.1109/32.42731

[8] Alfred V. Aho and Jeffrey D. Ullman. 1979. Universality of Data Retrieval Languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. ACM, New York, NY, USA, 110–119. https://doi.org/10.1145/567752.567763

[9] Alexander Alexandrov, Georgi Krastev, and Volker Markl. 2019. Representations and Optimizations for Embedded Parallel Dataflow Languages. *ACM Trans. Database Syst.* 44, 1, Article 4 (Jan. 2019), 44 pages. https://doi.org/10.1145/3281629

[10] Faisal Alkhateeb and Jérôme Euzenat. 2014. Constrained regular expressions for answering RDF-path queries modulo RDFS. *IJWIS* 10, 1 (2014), 24–50. https://doi.org/10.1108/IJWIS-05-2013-0013

[11] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1421–1432. https://doi.org/10.1145/3183713.3190654

[12] Anonymous. 2019. Full proofs for SIGMOD2020 submission 48. https://github.com/asigmod/SIGMOD2020submission48

[13] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 629–638. https://doi.org/10.1145/2187836.2187922

[14] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869. https://doi.org/10.1109/TKDE.2016.2633993

[15] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2019. gMark: a domain and query language independent framework. https://github.com/graphMark/gmark

[16] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '86)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/6012.15399

[17] Francois Bancilhon and Raghu Ramakrishnan. 1988. An amateur's introduction to recursive query processing strategies. In *Readings in Artificial Intelligence and Databases*. Elsevier, 376–430.

[18] Pablo Barcelo, Diego Figueira, and Leonid Libkin. 2012. Graph Logics with Rational Relations and the Generalized Intersection Problem. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS '12)*. IEEE Computer Society, Washington, DC, USA, 115–124. https://doi.org/10.1109/LICS.2012.23

[19] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4, Article 31 (Dec. 2012), 46 pages. https://doi.org/10.1145/2389241.2389250

[20] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Morgan and Claypool publishers.

[21] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *PVLDB* 11, 2 (2017), 149–161. https://doi.org/10.14778/3149193.3149196

[22] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. 1996. A Query Language and Optimization Techniques for Unstructured Data. *SIGMOD Rec.* 25, 2 (June 1996), 505–516. https://doi.org/10.1145/235968.233368

[23] Ashok K. Chandra. 1981. Programming Primitives for Database Languages. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '81)*. ACM, New York, NY, USA, 50–62. https://doi.org/10.1145/567532.567537

[24] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

[25] Mariano P. Consens and Alberto O. Mendelzon. 1990. GraphLog: A Visual Formalism for Real Life Recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '90)*. ACM, New York, NY, USA, 404–416. https://doi.org/10.1145/298514.298591

[26] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*. ACM, New York, NY, USA, 323–330. https://doi.org/10.1145/38713.38749

[27] Oliver M Duschka, Michael R Genesereth, and Alon Y Levy. 2000. Recursive query plans for data integration. *The Journal of Logic Programming* 43, 1 (2000), 49–73.

[28] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8. http://sites.computer.org/debull/A12mar/vicol.pdf

[29] M. Fernandez and D. Suciu. 1998. Optimizing regular path expressions using graph schemas. In *Proceedings 14th International Conference on Data Engineering*. 14–23. https://doi.org/10.1109/ICDE.1998.655753

[30] Daniela Florescu, Alon Levy, and Alberto Mendelzon. 1998. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Rec.* 27, 3 (Sept. 1998), 59–74. https://doi.org/10.1145/290593.290605

[31] Max Planck Institute for Informatics and Telecom ParisTech University. 2019. YAGO: A high-quality knowledge base. https://www.mpi-inf.mpg.de/yago-naga/yago/

[32] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[33] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International*

Conference on Management of Data (SIGMOD '18). ACM, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[34] Georges Gardarin and Christophe de Maindreville. 1986. Evaluation of Database Recursive Logic Programs As Recurrent Function Series. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. ACM, New York, NY, USA, 177–186. https://doi.org/10.1145/16894.16872

[35] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[36] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *Data Engineering Bulletin* 18 (1995).

[37] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 209–218. http://dl.acm.org/citation.cfm?id=645478.757691

[38] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*. 520–523. https://doi.org/10.5441/002/edbt.2018.62

[39] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling Kleene: Fast Property Paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, Article 14, 7 pages. https://doi.org/10.1145/2484425.2484443

[40] Steve Harris and Andy Seaborne. 2013. SPARQL 1.1 Query Language, W3C Recommendation. https://www.w3.org/TR/sparql11-query/

[41] Olaf Hartig and Giuseppe Pirrò. 2017. SPARQL with property paths on the Web. *Semantic Web* 8, 6 (2017), 773–795. https://doi.org/10.3233/SW-160237

[42] Maurice AW Houtsma and Peter MG Apers. 1992. Algebraic optimization of recursive queries. *Data & Knowledge Engineering* 7, 4 (1992), 299–325.

[43] Michael Kifer and Eliezer L. Lozinskii. 1990. On Compile-time Query Optimization in Deductive Databases by Means of Static Filtering. *ACM Trans. Database Syst.* 15, 3 (Sept. 1990), 385–426. https://doi.org/10.1145/88636.87121

[44] Donald Kossmann and Konrad Stocker. 2000. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Trans. Database Syst.* 25, 1 (March 2000), 43–82. https://doi.org/10.1145/352958.352982

[45] Dexter Kozen. 1983. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.* 27 (1983), 333–354. https://doi.org/10.1016/0304-3975(82)90125-6

[46] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (March 2016), 53 pages. https://doi.org/10.1145/2850413

[47] Erik Meijer and Gavin M. Bierman. 2011. A co-relational model of data for large shared data banks. *Commun. ACM* 54, 4 (2011), 49–58. https://doi.org/10.1145/1924421.1924436

[48] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (Dec. 1995), 1235–1258. https://doi.org/10.1137/S009753979122370X

[49] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. 1989. Efficient Evaluation of Right-, Left-, and Multi-linear Rules. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD '89)*. ACM, New York, NY, USA, 235–242. https://doi.org/10.1145/67544.66948

[50] Van-Quyet Nguyen and Kyungbaek Kim. 2017. Estimating the Evaluation Cost of Regular Path Queries on Large Graphs. In *Proceedings of the Eighth International Symposium on Information and Communication Technology (SoICT 2017)*. ACM, New York, NY, USA, 92–99. https://doi.org/10.1145/3155133.3155160

[51] Online. 2019. http://www.opencypher.org.

[52] Online. 2019. The PostgreSQL system: http://www.postgresql.org.

[53] Domenico Saccà and Carlo Zaniolo. 1986. On the Implementation of a Simple Class of Logic Queries for Databases. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS '86)*. ACM, New York, NY, USA, 16–23. https://doi.org/10.1145/6012.6013

[54] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory (ICDT '10)*. ACM, New York, NY, USA, 4–33. https://doi.org/10.1145/1804669.1804675

[55] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*. 23–34. https://doi.org/10.1145/582095.582099

[56] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1135–1149. https://doi.org/10.1145/2882903.2915229

[57] Michael Stillger and Myra Spiliopoulou. 1996. Genetic Programming in Database Query Optimization. In *Proceedings of the 1st Annual Conference on Genetic Programming*. MIT Press, Cambridge, MA, USA, 388–393. http://dl.acm.org/citation.cfm?id=1595536.1595591

[58] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. 2019. The implementation of POSTGRES. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 519–559. https://doi.org/10.1145/3226595.3226639

[59] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*. 697–706. https://doi.org/10.1145/1242572.1242667

[60] K Tuncay Tekle and Yanhong A Liu. 2011. More efficient datalog queries: subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 661–672.

[61] Jeffrey D. Ullman. 1988. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA.

[62] Jacopo Urbani, Ceriel JH Jacobs, and Markus Krötzsch. 2016. Column-Oriented Datalog Materialization for Large Knowledge Graphs.. In *AAAI*. 258–264.

[63] Nikolay Yakovets, P Godfrey, and J Gryz. 2013. Evaluation of SPARQL property paths via recursive SQL. 1087 (01 2013).

[64] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. WAVEGUIDE: Evaluating SPARQL Property Path Queries.. In *EDBT*. 525–g528.

[65] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2016. Query Planning for Evaluating SPARQL Property Paths. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1875–1889. https://doi.org/10.1145/2882903.2882944

A PROOFS FOR SECTION 3 (THE μ-EXTENDED REL. ALGEBRA)

Proposition (1). *Given a database $(\mathcal{R}, \Gamma, D)$ and $\varphi \in \mathcal{F}[\Gamma]$, if $\Gamma \vdash \varphi : t$ then the relation $[\![\varphi]\!]_D$ has type t.*

Proof. Let Γ be compatible with V. The property is thus true for relation variables; it is also true for constants and by induction unions, joins, antijoins, filters, duplication or removal of columns. This leaves us with fixpoints.

Suppose $\Gamma \vdash \mu(X = \varphi) : t$. The empty set of mappings is compatible with t, thus $[\![\varphi]\!]_{V[X/\emptyset]}$ is compatible with t by induction, and thus by further induction we have $[\![\mu(X = \varphi)]\!]_V$ compatible with t. □

Proposition (2). *If $\mu(X = \varphi)$ of type t is linear, positive and non mutually recursive then the function $f(S) = [\![\varphi]\!]_{V[X/S]}$ (for S a set of mappings of type t) is such that:*

$$f(S) = f(\emptyset) \cup \bigcup_{x \in S} f(\{x\})$$

and thus f has a fixpoint with $[\![\mu(X = \varphi)]\!]_V = f^\infty(\emptyset)$.

Proof. We will first prove by induction on the size of terms the following property: given a valid term φ, for all $S \neq \emptyset$ we have $\forall m \in [\![\varphi]\!]_{V[X/S]}\ \exists w_m \in S\ m \in [\![\varphi]\!]_{V[X/\{w_m\}]}$.

- Using lemma 1 the property is clearly true for terms φ such that X is not free in φ. And the only relation variable where X appears free is X. For X the property trivially holds (with $w_m = m$).
- For unary operators $\varphi \in \{\rho_a^b(\varphi_1), \widetilde{\pi}_a(\varphi_1)$ we have $m \in [\![\varphi]\!]_{V[X/S]}$ implies the existence of $m' \in [\![\varphi_1]\!]_{V[X/S]}$ such that m is the image of m' through this operator. By the induction hypothesis, for m' there is w such that $m' \in [\![\varphi_1]\!]_{V[X/\{w_{m'}\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w_{m'}\}]}$
- For a join operator $\varphi = \varphi_1 \bowtie \varphi_2$ we have by linearity that φ is constant in X for some i (let us note $\bar{i} = 3 - i$). If $\varphi_{\bar{i}}$ is also constant in X then φ is constant in X so we can refer to the first item.
  Otherwise, $m \in [\![\varphi_{\bar{i}}]\!]_{V[X/S]}$ implies the existence of $m_1 \in [\![\varphi_1]\!]_{V[X/S]}$ and $m_2 \in [\![\varphi_2]\!]_{V[X/S]}$ such that $m = m_1 + m_2$. By induction, there exists $w_{m_{\bar{i}}}$ such that $m_{\bar{i}} \in [\![\varphi_{\bar{i}}]\!]_{V[X/\{w_{m_{\bar{i}}}\}]}$. For i we have $[\![\varphi_i]\!]_{V[X/S]} = [\![\varphi_i]\!]_{V[X/\emptyset]} = [\![\varphi_i]\!]_{V[X/\{w_{m_{\bar{i}}}\}]}$ which means that in any case $m_1 \in [\![\varphi_1]\!]_{V[X/\{w_{m_i}\}]}$, $m_2 \in [\![\varphi_2]\!]_{V[X/\{w_{m_i}\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w_{m_i}\}]}$.
- For the term $\varphi = \varphi_1 \triangleright \varphi_2$ with any mapping $m \in [\![\varphi]\!]_{V[X/S]}$ is built using at least one mapping m_1 from $[\![\varphi_1]\!]_{V[X/S]}$. By induction, we have w such that $m_1 \in [\![\varphi]\!]_{V[X/\{w\}]}$. But X does not appear free in φ_2, thus $[\![\varphi_2]\!]_{V[X/S]} = [\![\varphi_2]\!]_{V[X/\{w\}]}$ and thus $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\{w\}]}$.
- For the term $\varphi = \varphi_1 \cup \varphi_2$, $m \in [\![\varphi]\!]_{V[X/S]}$ implies $m \in [\![\varphi_1]\!]_{V[X/S]}$ or $m \in [\![\varphi_2]\!]_{V[X/S]}$. By induction we have w such that $m \in [\![\varphi_1]\!]_{V[X/\{w\}]}$ or $m \in [\![\varphi_2]\!]_{V[X/\{w\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w\}]}$.
- Given the term $\mu(Y = \varphi)$ we have $\mu(Y = \varphi))$ constant in X and thus the result by lemma 1.

□

Lemma (1). *Let φ be a term.*

- *If φ is recursive in X then for all V, $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$.*
- *If φ is constant in X, then φ does not depend on X, i.e. for all S and V, $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$.*

Proof. For the constant part the result is trivial by induction.

For the recursive part, we also work by induction. It is true for base relations (constants cannot be recursive) but we need to be careful because the subterms of a recursive term can be non-recursive. By the definition of *rec* this can only happen for $\varphi_1 \bowtie \varphi_2$, but one of the φ_i has to be recursive and since the join with an empty set leads to an empty set the result holds by induction. □

Proposition (3). *A fixpoint term $\mu(X = \varphi)$, linear, positive and non mutually recursive can be rewritten to either: an empty term, a term φ with one less fixpoint or a decomposed fixpoint.*

Proof. Given a fixpoint $\mu(X = \varphi)$ we can always decompose φ into a Constant part C and a Recursive part R (possibly empty).

The idea is to prove by induction on φ that it is true for φ where X is linear, positive and non-mutually recursive:

- For a term φ constant in X the result is clear ($R = \emptyset$, $C = \varphi$).
- For X, $R = X$, $C = \emptyset$.

- For a unary operator $f(\varphi) \in \{\rho_a^b(\varphi), \widetilde{\pi}_a(\varphi), \sigma_{\dagger}(\varphi)\}$, we have φ that can be decomposed into R_φ, C_φ the solution is $f(R_\varphi), f(C_\varphi)$ (where $f(s)$ represents $f(s)$ if s is a term and $\emptyset$ otherwise).
- For a join $\varphi_1 \bowtie \varphi_2$, let us suppose by symmetry that φ_2 is constant in X; then φ_1 can be decomposed into R, C and the result is $R \bowtie \varphi, C \bowtie \varphi$.
- For an antijoin, the same argument as the one for joins works.
- For unions the results for subterms can be merged.
- Fixpoints are constant in X by the non mutual recursion hypothesis.

□

B DATALOG & μ-RA EXPRESSIVE POWERS

In this section we present how to translate various Datalog into μ-RA. The results presented here are not at the heart of our work and most of them are already known in the literature (with very similar statements and with similar proofs, see e.g. [4] or [2] regarding Datalog and the while^{+} language).

The only novelty of this proof relies in the proof that the linearity of *rest-μ*-RA actually reduces the expressive power. However to understand why we need to present a translation from Datalog to μ-RA and back. We will therefore not rely on formal proofs but we will build some intuition and provide examples.

B.1 Datalog with only one IBD

We recall in this section that datalog programs can always be transformed to programs that have only one recursive rule and one output rule (this is exercise 14.17 of the alice book [2]).

B.1.1 Step 1: the n-aryfication. Given a Datalog program P, we can always modify P so that all rules in P are n-ary for some n. To do that we simply take n to be the maximal arity over all the rules and extend all the rules with a constant c to match this arity.

For instance:

```
Path(1,2).
Access(1).
Access(X) :- Access(Y), Path(X,Y)
```

can be made 3-ary in the following way:

```
Path(1,2,c).
Access(1,c,c).
Access(X,c,c) :- Access(Y,c,c), Path(X,Y,c)
```

B.1.2 Step 2: one rule datalog. Given a Datalog program P, we can always modify P so that there is only one recursive rule and one "output" rule in P. The idea is to first convert P into a n-ary program P' (for some n) then creates a unique $n+1$ rules that takes as its first argument the name of the rule. For instance, our running example becomes:

```
Rec(path,1,2,c).
Rec(access,1,c,c).
Rec(access,X,c,c) :- Rec(access,Y,c,c), Rec(path,X,Y,c).
Output(X) :- Rec(access,X,c,c).
```

B.2 From a derivation rule to μ-RA

It is a well-known fact that non-recursive datalog and the relational algebra coincide (see e.g. chapter 14 of the alice book [2]). Given a production $head(\bar{Y}) : -body_1(\bar{X}_1), \ldots, body_k(\bar{X}_k)$ we can translate $body_1(\bar{X}_1), \ldots, body_k(\bar{X}_k)$ using $k-1$ joins between each $body_i$, renames to rename arguments of $body_i$, antiprojections to remove existential variables, and filters for constants. Finally we use joins with constants for the constants of the head and renames for the variables.

For instance, if we translate the Datalog IBD *Rec* into a term *Rec* that has 4 columns (a_1, a_2, a_3 and a_4) the translation of the body

```
Rec(access,X,c,c) :- Rec(access,Y,c,c), Rec(path,X,Y,c).
```

is $\rho_{a2}^{Y}(\widetilde{\pi}_{a_1}(\widetilde{\pi}_{a_3}(\widetilde{\pi}_{a_4}(\sigma_{a1=access \wedge a3=c \wedge a4=c}(Rec))))) \bowtie \widetilde{\pi}_{a_1}\left(\widetilde{\pi}_{a_4}\left(\rho_{a3}^{Y}\left(\rho_{a2}^{X}\left(\sigma_{a1=path \wedge a4=c}(Rec)\right)\right)\right)\right)$

The whole translation is (using *body* to denote the above term):

$\rho_X^{a2} (\widetilde{\pi}_Y (body)) \bowtie |a3 \rightarrow c| \bowtie |a4 \rightarrow c| \bowtie |a1 \rightarrow access|$

B.3 From inflationary Datalog$^\neg$ to μ-RA

Given an inflationary-Datalog$^\neg$ program P that, *w.l.o.g.*, has recursive rule *Rec* and one output rule *Output* we can translate *Rec* to a fixpoint of the form $\mu(Rec = \varphi_1 \cup \dots \varphi_k)$ where each φ_i corresponds to one derivation of the rule *Rec*. Finally we translate each production of Output into a term ψ (where Rec is replaced by the fixpoint above) and we generate a term that is the union of all these ψ.

Given our initial example we have the term O (here we cut the translation to ease the reading):

$$
\begin{array}{lcr}
B_1 & = & |a_1 \rightarrow path| \bowtie |a_2 \rightarrow 1| \bowtie |a_3 \rightarrow 2| \bowtie |a_4 \rightarrow c| \\
B_2 & = & |a_1 \rightarrow access| \bowtie |a_2 \rightarrow 1| \bowtie |a_3 \rightarrow c| \bowtie |a_4 \rightarrow c| \\
B_3 & = & \rho_X^{a_3} \left(\widetilde{\pi}_Y \left(\rho_{a2}^Y (\widetilde{\pi}_{a_1} (\widetilde{\pi}_{a_3} (\widetilde{\pi}_{a_4} (\sigma_{a1=access \wedge a3=c \wedge a4=c} (Rec))))) \bowtie \rho_{a3}^Y \left(\rho_{a2}^X \left(\widetilde{\pi}_{a_1} \left(\widetilde{\pi}_{a_4} \left(\sigma_{a1=path \wedge a4=c} (Rec)\right)\right)\right)\right)\right)\right) \\
B_4 & = & \mu(Rec = B_1 \cup B_2 \cup B3) \\
O & = & \widetilde{\pi}_{a_1} \left(\widetilde{\pi}_{a_3} \left(\widetilde{\pi}_{a_4} \left(\sigma_{a_1=access} (B)_4\right)\right)\right)
\end{array}
$$

The semantics does coincide with inflationary-Datalog$^\neg$ because the formula $B_1 \cup B_2 \cup B_3$ captures the "immediate consequence" of the Datalog program.

B.4 From stratified Datalog to μ-RA

In a stratified Datalog program, each rule can be indexed with an integer n such that a negation of a rule indexed by k can only appear in the production rule of a term indexed with $k' > k$.

In the case of a stratified Datalog program, merging all the rules into one will break the stratification. The trick here is to operate stratum by stratum and translate the stratum i into a rule Rec_i. The resulting program will have one rule per stratum.

Just like in the inflationary case, each stratum i can be translated into a unique fixpoint $\mu(X_i = \varphi_i)$. The production rules of the stratum i can only reference to a Rec_j where $j \leq i$. We translate Rec_i into X_i and the Rec_j into $\mu\left(X_j = \varphi_j\right)$. Note that each φ_i can contain several occurrences of Rec_j with $j < i$ and that makes the translation exponential but all the fixpoints do are non mutually recursive and positive.

Let us consider the following example (already stratified):

```
Path(...) an EDB

Access_1(0).
Access_1(X) :- Access_1(Y),Path(Y,X)

Access_2(1).
Access_2(X) :- Access_2(Y),Path(Y,X), not Access_1(Y)
```

We translate Path into a term $\mu(X_0 = \varphi_0)$ (despite the fixpoint φ_0 is actually not recursive as Path is an EDB). Then we translate $Access_1$:

$$\mu\left(X_1 = |a_1 \rightarrow 0| \cup \rho_{a_2}^{a_1} (\widetilde{\pi}_{a_1} (X_1 \bowtie Path))\right)$$

Then we translate $Access_2$ (using $Access_1$ to denote the term above) :

$$\mu\left(X_2 = |a_1 \rightarrow 1| \cup \rho_{a_2}^{a_1} (\widetilde{\pi}_{a_1} (X_2 \bowtie Path \triangleright Access_1))\right)$$

B.5 From linear Datalog to *rest-μ*-RA

Given a linear Datalog program, we can use the stratified translation. In the resulting term each φ_i is composed of $\psi_1 \cup \dots \psi_k$ where each of the ψ_j corresponds to a linear production rule and thus contains at most one occurrence of X_i therefore our μ-RA term is also linear (in addition to be recursive and positive as proven by the stratified translation). All in all, our term does belong to *rest-μ*-RA.

B.6 From *rest*-μ-RA to linear Datalog

This direction is actually very simple once we know how to translate a term to a Datalog program, we just need to check that the resulting term is actually linear. To translate terms into Datalog, we work bottom-up associating each subterm φ to a Datalog rule. Datalog rules have columns that are indexed (there is a first column, a second, a third, etc.) while μ-RA has column names. To handle this discrepancy, we suppose that we have calculated the type of each term (i.e. we compute a set of column names), then we order column names (any total order on the column names can be used).

The only difficulty here is the language of filters, in *rest*-μ-RA we actually impose no restriction on the filter conditions; for the translation we suppose that only the equality is used.

We thus recursively create production rules for each Datalog predicate $s_\varphi(\bar{T})$ corresponding the each term φ $s_\varphi(\bar{T})$ (where $\bar{T}$ is the ordered set of columns of the type of φ).

- For $\varphi = \varphi_1 \bowtie \varphi_2$ we create a rule for the join: $s_\varphi(\bar{T}) \leftarrow s_1(\bar{T}_1), s_2(\bar{T}_2)$.
- For $\varphi = \varphi_1 \cup \varphi_2$ we have two production rules, one for each φ_i: $s_\varphi(\bar{T}) \leftarrow s_{\varphi_i}(\bar{T}_i)$.
- For $\varphi = \varphi_1 \triangleright \varphi_2$ we create the rule $s_\varphi \leftarrow s_{\varphi_1}(\bar{T}_1), \neg s_{\varphi_2}(\bar{T}_2)$.
- For $\varphi = \sigma_{a=b}(\varphi')$ we create the rule $s_\varphi(\bar{T}_1, b, \bar{T}_2) \leftarrow s_{\varphi'}(\bar{T}_1, b, \bar{T}_2)$ if we suppose that the ordered type of φ' is $\bar{T}_1, a, \bar{T}_2$
- For $\varphi = \widetilde{\pi}_p(\varphi')$ we create the rule $s_\varphi(\bar{T}_\varphi) \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$
- For $\varphi = \rho_a^b(\varphi')$ we create the rule $s_\varphi(\bar{T}') \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$ where $\bar{T}'$ is $\bar{T}_{\varphi'}$ where we inserted a a in the place of where b will be stored.
- For $\varphi = \mu(X = \varphi')$ we create the rule $s_X(\bar{T}) \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$.
- For $\varphi = X$ we create the rule $s_X(\bar{T}) \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$.

Since the *rest*-μ-RA term is linear we can see that each production rule contain at most one subgoal that is recursive with the head.

C PROOFS FOR SECTION 4 (GENERATING NEW QUERY PLANS)

Lemma (2). *Let w be a mapping and φ a term linear, positive and non mutually recursive in X. For all $m \in [\![\varphi]\!]_{V[X/\{w\}]}$ either $m \in [\![\varphi]\!]_{V[X/\emptyset]}$ or there exists $p \in d(\varphi, X)$ such that for all $c \in dom(w)$:*

$$\Big(p(c) = \bot\Big) \vee \Big(p(c) \notin dom(w)\Big) \vee \Big(m(c) = w\big(p(c)\big)\Big)$$

Proof. Let w and $m \in [\![\varphi]\!]_{V[X/\{w\}]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$. By induction:

- Since m exists, X can only be free in φ (no $|c \to v|$, no $Y \neq X$, no fixpoints).
- For a relation X, the result is clear.
- For a union we have i such that $m \in [\![\varphi_i]\!]_{V[X/\{w\}]}$ and thus the result.
- For a join $\varphi_1 \bowtie \varphi_2$ let us suppose by symmetry that φ_1 is not constant in X and that φ_2 is. We have $m_1 \in [\![\varphi]\!]_{V[X/\{w\}]}$, $d_1 \in d(\varphi_1, X) \subseteq d(\varphi_1 \bowtie \varphi_2, X)$ (with inductive hypothesis) and $m_2 \in [\![\varphi]\!]_{V[X/\emptyset]}$.
  For each $c \in dom(w)$ we either have $d_1(c) = \bot$ or $d_1(c) \notin dom(w)$ or $m_1(c) = w(d_1(c))$ and thus $m(c) = w(d_1(c))$. Note that $m_2(c)$ might or might not be defined but if $(d_1(c) \neq \bot) \wedge (p(c) \in dom(w))$ then $m_1(c)$ is also defined and $m(c) = m_1(c)$.
- For an antijoin or a filter the result is clear.
- For a column rename or removal, the definition of d makes it work. Let us note $\lambda(\varphi)$ the term, we have $m \in [\![\lambda(\varphi)]\!]_{V[X/\{w\}]}$ that implies $m' \in [\![\varphi]\!]_{V[X/\{w\}]}$ and $d' \in d(\varphi, X)$ with the property. And $d' \circ \lambda$ works.

□

Lemma (3). *Given a fixpoint term $\mu(X = \varphi) \in \mathcal{F}[\Gamma]$ of type t and a mapping of type t, $w \in [\![\mu(X = \varphi)]\!]_V$ if and only if we can find a* lineage *$w_0, \ldots, w_n$ for w, that is $w_0, \ldots, w_n$ such that $w_0 \in [\![\varphi]\!]_{V[X/\emptyset]}$ and $w_{i+1} \in [\![\varphi]\!]_{V[X/\{w_i\}]}$.*

Furthermore for all lineages $w_0, \ldots, w_n$ and all $c \in t \cap stab(\varphi, X)$, we have $w_0(c) = w(c)$.

Proof. Let $w \in [\![\mu(X = \varphi)]\!]_V$ and let n minimal such that $w \in U_n$ (as defined by the semantic). By iterating proposition 2 we $w_0, \ldots, w_n = w$ as expected.

Conversely if we have such $w_0, \ldots, w_n = w$ then clearly $w \in [\![\mu(X = \varphi)]\!]_V$.

Now, by Lemma 2, for each $0 \leq i \leq n-1$, the mappings w_i and w_{i+1} there is $p \in d(\varphi, X)$ such that for all $c \in stab(\varphi, X) \cap t$, $w_{i+1} = w_i(p(c) = w_i(c)$. By iteration so does w_0 and w. □

Theorem (1 Pushing filters). *Let $\mu(X = \varphi)$ be a fixpoint term, V an environment and f a filter condition with $FC(f) \subseteq stab(\varphi, X)$. Then we have $\llbracket \sigma_{\mathfrak{f}}(\mu(X = \varphi)) \rrbracket_V = \llbracket \mu(X = \sigma_{\mathfrak{f}}(\varphi)) \rrbracket_V$, and if $\mu(X = \varphi)$ can be decomposed into $\mu(X = \kappa \cup \psi)$, we also have $\llbracket \sigma_{\mathfrak{f}}(\mu(X = \kappa \cup \psi)) \rrbracket_V = \llbracket \mu(X = \sigma_{\mathfrak{f}}(\kappa) \cup \psi) \rrbracket_V$.*

Proof. Clearly, all lineages of $\llbracket \mu(X = \sigma_{\mathfrak{f}}(\varphi)) \rrbracket_V$ are lineages of $\llbracket \mu(X = \varphi) \rrbracket_V$ and all $w \in \llbracket \mu(X = \sigma_{\mathfrak{f}}(\varphi)) \rrbracket_V$ pass the filter $\mathfrak{f}$.

Let $w \in \llbracket \sigma_{\mathfrak{f}}(\mu(X = \varphi)) \rrbracket_V$. Let $w_0, \ldots, w_n$ be a lineage of w: w passes the filter and by Lemma 3, w has the same values as all w_i on $FC(f)$; therefore $w_0, \ldots, w_n$ is also a lineage of $\llbracket \mu(X = \sigma_{\mathfrak{f}}(\varphi)) \rrbracket_V$. □

Theorem (2 Pushing anti-joins). *Let $\mu(X = \varphi)$ be a fixpoint term, V an environment and ψ a term of type $t \subseteq stab(\varphi, X)$ (we suppose that X is not a free variable of ψ). Then we have $\llbracket \mu(X = \varphi) \triangleright \psi \rrbracket_V = \llbracket \mu(X = \varphi \triangleright \psi) \rrbracket_V$, and if $\mu(X = \varphi)$ can be decomposed into $\mu(X = \kappa \cup \xi)$, we also have $\llbracket \mu(X = \kappa \cup \xi) \triangleright \psi \rrbracket_V = \llbracket \mu(X = \kappa \triangleright \psi \cup \xi) \rrbracket_V$.*

Proof. Clearly, all lineages of $\llbracket \mu(X = \varphi \triangleright \psi) \rrbracket_V$ are lineages of $\llbracket \mu(X = \varphi) \rrbracket_V$ and all $w \in \llbracket \mu(X = \varphi \triangleright) \rrbracket_V$ are not compatible with any mapping of $\llbracket \psi \rrbracket_V$.

Let $w \in \llbracket \mu(X = \varphi) \triangleright \psi \rrbracket_V$. Let $w_0, \ldots, w_n$ be a lineage of w: w is not compatible with any mapping $w' \in \llbracket \psi \rrbracket_V$ and by Lemma 3, w has the same values as all w_i on t (the type of ψ); therefore $w_0, \ldots, w_n$ is also a lineage of $\llbracket \mu(X = \varphi \triangleright \psi) \rrbracket_V$. □

Lemma (4). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t, let $c \in (\mathfrak{C} \setminus t)$ that can be added to ψ, and w a mapping of type t. We note $w(v) = w \cup \{c \to v\}$.*

If $\forall R \in \mathcal{R}, c \notin \Gamma(R)$, then we have:

- $c \in stab(\psi, X)$
- $\Gamma \cup \{X \to t \cup \{c\}\} \vdash \psi : t \cup \{c\}$
- $\llbracket \psi \bowtie |c \to v| \rrbracket_{V[X/\{w\}]} = \llbracket \psi \rrbracket_{V[X/\{w(v)\}]}$

Proof. We will prove the result $\llbracket \psi \rrbracket_{w(v)} = \llbracket \psi \bowtie |c \to v| \rrbracket_w$ inductively on the size of ψ a term recursive in X.

Note that when a subformula ξ of ψ is constant in X we have that $\llbracket \xi \rrbracket_{V[X/\{w\}]} = \llbracket \xi \rrbracket_{V[X/\{w(v)\}]}$ by lemma 1 and we also have that c is not in the type of this ξ (since $\forall R, c \notin \Gamma R$ and the definition of *add* forbids to rename a column into c). Note also that subformula that are fixpoints or constants are necessarily constant in X.

Let us explore the various cases. For the simplicity of proofs, we use $\bowtie$ and $\triangleright$ directly with sets of mappings (e.g. $A \bowtie B = \{m_A + m_B \mid m_A \in A\ m_B \in B \land m_A \sim m_B\}$).

- For the formula X, the result is trivial and it is the only base case (constants and other variables cannot be recursive in X).
- For $\varphi_1 \bowtie \varphi_2$, one of φ_1, φ_2 has to be constant, the other recursive. By symmetry, we suppose that φ_1 is recursive and φ_2 constant. We have $\llbracket \varphi_1 \bowtie \varphi_2 \rrbracket_{V[X/\{w(v)\}]} = \llbracket \varphi_1 \rrbracket_{V[X/\{w(v)\}]} \bowtie \llbracket \varphi_2 \rrbracket_{V[X/\{w(v)\}]} = \llbracket \varphi_1 \bowtie |c \to v| \rrbracket_{V[X/\{w\}]} \bowtie \llbracket \varphi_2 \rrbracket_{V[X/\{w\}]} = \llbracket (\varphi_1 \bowtie |c \to v|) \bowtie \varphi_2 \rrbracket_{V[X/\{w\}]} = \llbracket (\varphi_1 \bowtie \varphi_2) \bowtie |c \to v| \rrbracket_{V[X/\{w\}]}$
- For $\varphi_1 \triangleright \varphi_2$ we similarly have $\llbracket \varphi_1 \triangleright \varphi_2 \rrbracket_{V[X/\{w(v)\}]} = \llbracket \varphi_1 \rrbracket_{V[X/\{w(v)\}]} \triangleright \llbracket \varphi_2 \rrbracket_{V[X/\{w(v)\}]} = \llbracket \varphi_1 \bowtie |c \to v| \rrbracket_{V[X/\{w\}]} \triangleright \llbracket \varphi_2 \rrbracket_{V[X/\{w\}]} = \llbracket (\varphi_1 \bowtie |c \to v|) \triangleright \varphi_2 \rrbracket_{V[X/\{w\}]} = \llbracket (\varphi_1 \triangleright \varphi_2) \bowtie |c \to v| \rrbracket_{V[X/\{w\}]}$ and the last line that uses the commutativity of $\triangleright$ over $\bowtie$ is only true because c cannot be in the type of φ_2.
- For $\sigma_{\mathfrak{f}}(\varphi')$, $\widetilde{\pi}_a(\varphi')$ and $\rho_a^b(\varphi)$ the result comes easily as $c \notin \{a, b\} \cup FC(f)$.

□

Theorem (3 Pushing joins). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ and $\varphi \in \mathcal{F}[\Gamma]$ (with $X \notin free(\varphi)$) a term of type t_φ such that:*

(1) $t_\varphi \subseteq stab(\psi, X)$
(2) $\forall c \in t_\varphi \setminus t_\kappa\ add(\psi, X, c)$

Then we have $\Gamma \vdash \mu(X = \kappa \bowtie \varphi \cup \psi) : t_\varphi \cup t_\kappa$ with for all V compatible with Γ:

$$\llbracket \varphi \bowtie \mu(X = \kappa \cup \psi) \rrbracket_V = \llbracket \mu(X = \kappa \bowtie \varphi \cup \psi) \rrbracket_V$$

Proof. Lemma 4 ensures us that $\Gamma \cup \{X \to t_\varphi \cup t_\kappa\} \vdash \psi : t_\varphi \cup t_\kappa$, and thus $\Gamma \vdash \mu(X = \varphi \bowtie \kappa \cup \psi) : t_\varphi \cup t_\kappa$.

Then if we take a lineage $w_0 \ldots w_n$ of $\llbracket \mu(X = \kappa \cup \psi) \rrbracket_V$ and there exists $u \in \llbracket \varphi \rrbracket_V$ compatible with w_n then $t_\varphi \subseteq stab(\psi, X)$ ensures us that u is compatible with all w_i.

Then by iterating Lemma 4, for each i and for each $c \in t_\varphi \setminus t_\kappa$, we have that $w_0(u), \ldots, w_n(u)$ is a valid lineage of $\llbracket \mu(X = \kappa \bowtie \varphi \cup \psi) \rrbracket_V$ and reciprocally. □

Theorem (4 Merging fixpoints). *Given two decomposed fixpoints $\mu(X = \kappa_1 \cup \psi_1)$ and $\mu(X = \kappa_2 \cup \psi_2)$ of types t_1 and t_2 such that:*

(1) $t_1 \cap t_2 \subseteq stab(\psi_2, X, C_2) \cap stab(\psi_1, X, C_1)$
(2) $\forall c \in t_1 \setminus t_2\ add(\psi_2, X, c)$
(3) $\forall c \in t_2 \setminus t_1\ add(\psi_1, X, c)$

then we have: $[\![\mu(X = \kappa_1 \cup \psi_1) \bowtie \mu(X = \kappa_2 \cup \psi_2)]\!]_V = [\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$.

Proof. For $i \in \{1, 2\}$, let $w_0, \ldots, w_{n_i}$ be a lineage of $[\![\mu\big(X = \kappa^i \cup \psi^i\big)]\!]_V$ with w_{n_1} compatible with w_{n_2}; we can easily construct a lineage of size $n_1 + n_2$ of the form $(w_0^1 + w_0^2) \ldots (w_{n_1}^1 + w_0^2) \ldots (w_{n_1}^1 + w_{n_2}^2)$ and for the same reason as the last theorem, it holds.

Now let us take a lineage $w_0, \ldots, w_n$ of $[\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$. We decompose w_i into $w_i^1 + w_i^2$ where w_i^j is the restriction of w_i to the type of κ_j. Those w_i^j are not necessarily forming a lineage but we consider the subsequence containing w_0^i and for each $i > 0$ w_j^i when $w_i^j \in [\![\psi_i]\!]_{V[X/\{w_{i-1}^j\}]}$. Then by the theorem condition when $w_i^j \in [\![\psi_i]\!]_{V[X/\{w_{i-1}^j\}]}$ we have $w_i^{\bar{j}} = w_{i-1}^{\bar{j}}$. The two resulting sequences are thus lineages and we have the expected theorem.

□

Theorem (5 Pushing antiprojections). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ. Let $b \in \mathfrak{C}$ be such that $add(\psi, X, b)$. Then:*

$$[\![\widetilde{\pi}_b\ (\mu(X = \kappa \cup \psi))]\!]_V = [\![\mu(X = \widetilde{\pi}_b\ (\kappa) \cup \psi)]\!]_V$$

Proof. This is a conclusion of lemma 4. Let $w_0, \ldots, w_n$ be a lineage of $[\![\mu(X = \widetilde{\pi}_c\ (\kappa) \cup \psi)]\!]_V$ there exists v such that $w_0(v) \in [\![\kappa]\!]_{V[X/\emptyset]}$ and if we have $w_i(v)$ we can find $w_{i+1}(v) \in [\![\psi]\!]_{V[X/\{w_i(v)\}]}$ by lemma 4. In the end we have a lineage for $w(v) \in [\![\mu(X = \kappa \cup \psi)]\!]_V$ and which means $w \in \widetilde{\pi}_c\ (\mu(X = \kappa \cup \psi))$.

Notice that lemma 4 gives an equality therefore this is a bijection between lineage and also proves the converse way.

□

REFERENCES FOR THE APPENDIX

[2] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.
[4] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43(1):62–124, August 1991.