On the Optimization of Recursive Relational Queries: Application to Graph Queries
Louis Jachiet, Pierre Genevès, Nils Gesbert, Nabil Layaida

To cite this version:

HAL Id: hal-01673025
https://inria.hal.science/hal-01673025v4
Submitted on 28 Nov 2019 (v4), last revised 19 Jun 2021 (v6)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
On the Optimization of Recursive Relational Queries: Application to Graph Queries

Louis Jachiet
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Pierre Genevès
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Nils Gesbert
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

Nabil Layaida
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

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 forms of recursion which reveal 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 rely on techniques of relational 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 $\mu$-RA, a variation of the Relational Algebra equipped with a fixpoint operator for expressing recursive relational queries. $\mu$-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 make it possible to generate new query execution plans, that cannot be obtained with previous approaches. We present the syntax and semantics of $\mu$-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 [29]. 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) [27], and various extensions such as Conjunctions of them (CRPQs) and Union of CRPQs (UCRPQs) [19, 20, 26, 50]. For instance, the query language SPARQL 1.1 [43] introduced Property Paths, and language proposals such as OpenCypher [34, 55] 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 [54, 71]. In practice, even with datasets of modest sizes, the benchmarking work found in [15] 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:

$$\exists x \text{ isLocatedIn}+ \exists y \quad \text{Emmy_Noether worksAt} \ ?x$$

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 $Qex$. For instance:

- A first QEP, named $P_1$, 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 $P_2$, would rather start by computing the set of solutions for Emmy_Noether worksAt $?x$. Then, these results are joined with the set of pairs...
We will use as examples: 

The regular path expression is recursive because it includes the following query (taken from [5]), still intended to be executed against the YAGO dataset:

\(?x\) hasChild/livesIn/isLocatedIn/dealsWith Japan \(Q_2\)

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 \(R=\text{isLocatedIn+}/\text{dealsWith+}\) 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: \(P_3\), \(P_4\), \(P_5\) that we will use as examples:

- Plan \(P_3\) consists in first computing the two transitive closures \(\text{isLocatedIn+}\) and \(\text{dealsWith+}\) of the relations \(\text{isLocatedIn}\) and \(\text{dealsWith}\), respectively, and then joining the results. One pitfall of this execution 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 “\(\text{isLocatedIn+}\)”), 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 \(P_3\) can be optimized using this technique: the constant “Japan” can be “pushed inside the computation of \(\text{dealsWith+}\). This corresponds to plan \(P_4\) where the evaluation starts from \(?t\) \(\text{dealsWith}\) Japan, then iteratively adds \(\text{dealsWith}\) steps on the left. Once this computation is finished, the results are joined with \(\text{isLocatedIn}\) and then, once again, iterative \(\text{isLocatedIn}\) steps are performed on the left. One advantage of \(P_4\) compared to \(P_3\) is that \(P_4\) processes each node at most twice while the whole transitive closures of \(\text{dealsWith}\) and \(\text{isLocatedIn}\) can be considerably larger (quadratic in the worst case).

- Another completely different way to evaluate \(R\), noted plan \(P_5\), consists in not computing any transitive closure but, instead, computing first the composed relation “\(\text{isLocatedIn}/\text{dealsWith}\)” and then recursively navigating in the graph either hopping on the left with \(\text{isLocatedIn}\) or hopping on the right by \(\text{dealsWith}\) to retrieve nodes. Notice that this slightly more general form of recursion is not a transitive closure of any relation.

The plan \(P_3\) will generally be slower than \(P_4\) or \(P_5\), even when using smart algorithms for computing transitive closures [9, 21, 61]. On some instances \(P_3\) will be much more efficient (if, for instance, \(?x\) \(\text{dealsWith+ Japan}\) has few solutions) and on some other instances \(P_5\) will be faster (if, for instance, there are few solutions of \(\text{dealsWith/\text{isLocatedIn}}\)). This is the case when evaluating \(Q_2\) over the YAGO graph (that has more than 62 million of edges): \(P_5\) is faster by a factor of more than 200x compared to other plans.

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

Contribution. We introduce a theory, which is an extension of Codd’s classical relational algebra, for the purpose of automatically obtaining efficient QEPs for recursive queries. Specifically, we introduce a fixpoint operator “\(\mu\)” 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 five 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 \(\mu\)-RA: a variation of the relational algebra (RA) equipped with a fixpoint operator. § 4 describes properties of \(\mu\)-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
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 [25] 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 interests 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 [51], instead they rely on a relational query engine to find an efficient way to process queries. Most relational query engines thus rely on an optimization process where the query to be processed is translated into an algebraic term. Then, this term is rewritten into a set of semantically equivalent terms (by e.g. pushing selections, or projections as close as possible to the sources). These different terms can be used to generate different QEPs. For a given query, the set of QEPs is called a Plan Space. The plan space depends on the set of equivalent terms that we can generate from the initial query.

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, 18] for complete surveys).

The \( a \)-extended RA [7] extends the RA with a recursive facility noted \( a \). The operator \( a \) is a transitive closure operator; given an \( a \)-extended RA term \( R \) defining a binary relation \( R \), \( a(R) \) represents its transitive closure, \( R^* \). In terms of expressive power over graphs, this corresponds to UCRAQs. It is for instance not capable of querying the path expression \( a^n / b^m \) which queries paths whose labels are first \( a \) 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 \( \text{PTIME} \).

Because LFP-RA terms can be computationally hard to evaluate, several syntactic fragments of LFP-RA have been studied. 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 \( a \)-extended RA and the expressive power of Datalog. For instance, all UCRPQs as well as the non-regular path expression \( a^n / b^m \) 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 \( \mu \)-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 \( \text{PTIME}=\text{PSPACE} \)).

2.3 Optimization of recursive RA

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 rule RW1). One drawback of their method is that it has more restrictions on fixpoints than \( \mu \)-RA. Being too restricted makes [8] unable to express some of our rewritten terms, e.g. merged fixpoints (see § 4.1.3). Furthermore, even if the authors note that it is sometimes possible to push projections (similar to our rule RW5) they do not provide an effective criterion for this.

Using the (unrestricted) LFP-RA, other authors [46] provided in 1990 a general framework for optimization. Their method works on so-called system graphs 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. In addition, they do not deal with conjunctions (our rules RW3 and RW4) and thus cannot find plan \( P_2 \) for \( Q_{ex} \) nor find plan \( P_5 \) for \( Q_{ex} \), for instance. Finally, compared to unrestricted LFP-RA, \( \mu \)-RA comes with restrictions on fixpoints that are needed for the validity of the rewrite rules we propose in § 4.1: the optimizations we present would not be correct on the unrestricted LFP-RA (as explained in § 3.4).
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. Many Datalog-based query evaluators exist [13, 28, 49, 57, 68] including recent fully-fledged commercial systems [13].

2.4.1 Magic Sets. A well-known Datalog optimization technique is the “Magic Sets” algorithm (see [17, 58]). 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 [53], 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 [53] 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 [65] over the Magic Sets. The idea is similar to the Magic Set: it pushes filters to avoid computing some “useless” facts. It has been proved 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 isLocatedIn+ / 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 found by the Datalog engine, even after Demand Transformation and program reversals.

2.4.4 Overall Comparison. Datalog engines do not explore Plan Spaces but use heuristics to find a good plan to evaluate queries. Depending on which combination of Magic Sets, Demand Transformation and Reversals it uses, a Datalog engine might be able to find a plan similar to $P_2$ for $Q_{ex}$, and to plan $P_1$ for $Q_2$. Since none of these optimizations can “merge” recursive terms, they do not allow to find a plan similar to $P_3$ for $Q_2$, which is —by an order of magnitude—the fastest plan for $Q_2$ on YAGO.

In a Datalog program corresponding to the optimized translation of A+/B+ at least one of the two transitive closures will be fully materialized (even if there are 0 solutions to A+/B+).

2.5 SQL

Since its 1999 version, the SQL standard supports recursive queries. SQL is more expressive than our $\mu$-RA as it allows, e.g., arithmetic and aggregation, but if we restrict SQL to its core our proposed $\mu$-RA is not very different from SQL with recursion. However, a restriction in SQL (the recursion variable cannot appear more than once in the recursive part of the query) forbids what would be the literal translation of some $\mu$-RA terms; in particular, we cannot translate the merged fixpoint of rule RW4 into standard SQL. Furthermore, recursive queries in SQL are not supported by all vendors and those who do support them tend to consider them 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 UCRPQs. 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 starts by computing the solution to the path $a$, then to the path $a/b$, then $a/b/c$ and recursively restarts (i.e. in a computational form we have $R_{i+1} = ((R_i/a)/(b/c))$. This method forces the associativity of the computation [70] and does 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
On the Optimization of Recursive Relational Queries: Application to Graph Queries

much more efficient, when e.g. there are only a handful of solutions to the path b/c.

2.6.2 \(\alpha\)-extended RA. The translation of the RPQ \((a/b/c)^+\) in the \(\alpha\)-extended RA includes a term \(\alpha(t)\) where \(t\) contains \((a/b/c)\). Therefore the relation \((a/b/c)\) will first be fully computed, and then its transitive closure. This is not always optimal, as it suffers from the same drawbacks as the previous approach.

2.6.3 Waveguide. The Waveguide paper [70] introduced a new technique to evaluate RPQs that mixes ideas from automata and from the \(\alpha\)-extended RA. Their idea is that the “interesting” plans to evaluate one RPQ can either start on Waveguide for individual RPQs. Tasweet improves on Waveguide, a tool focusing on conjunctions of RPQs. Wireframe, which is our variation of the domain-independent relational rewrite rules in the next section.

strictions on fixpoints will allow us to introduce powerful
and the fixpoint operator. Anti-projection replaces the more
included for self-completeness) except for the anti-projection

closures

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:

- \(\mathbb{B}\) an infinite set of values;
- \(\mathbb{C}\) an infinite set of column names;
- \(\mathbb{R}\) an infinite set of relation names.

\textbf{Definition 1.} A mapping or tuple is a partial function \(m: \mathbb{C} \rightarrow \mathbb{B}\) whose domain is finite. If \(dom(m) = \{c_1, \ldots, c_n\}\), \(m\) can also be seen as the set \(\{c_1 \rightarrow m(c_1), \ldots, c_n \rightarrow m(c_n)\}\).

\textbf{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 \(m_1 + m_2 : dom(m_1) \cup dom(m_2) \rightarrow \mathbb{B}\) 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.

\textbf{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 \(\mathbb{B}\)): for simplicity, a type is just a set of column names. 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: \(\mathcal{R} \subset \mathbb{R}\) is the set of relation names; \(\Gamma\), 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 \(\mu\)-RA terms

Our algebra \(\mu\)-RA is mainly a variation of the relational algebra, with the addition of a fixpoint operator \(\mu\); 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 \(\sigma(f)\), which operates on a relation by keeping only a subset of its mappings, depends on a filter \(f\) 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

\[
\text{syntax for filters, but we assume that for any filter } f \text{ we can compute a set } FC(f) \text{ of column names such that the result of } f(m) \text{ depends only on } \{c \rightarrow m(c) | c \in FC(f)\}.
\]
We add ensure that this fixpoint exists and can be computed with $\phi$ (representing a single mapping with a singleton domain). All other occurrences of $X$ which appear in a subterm of the form $\mu$ are free. The result of evaluating $\mu(X = \psi)$ is $[\mu(X = \psi)]_V$.

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 $\mu$-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 $\mu$; but in a subterm, we do not need to distinguish the two. In all cases, the semantics of a term $\phi$ depends on an environment $V$ which maps all free variables of $\phi$ to relations.

The semantics is defined in Fig. 1, where $[\phi]_V$ designates the result of evaluating $\phi$ 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 $\mu(X = \phi)$, 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 Restrictions on fixpoints

Our syntax for the $\mu$-RA is very general and comprises some counter-intuitive fixpoints 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 $\phi$ that contains a subterm $\psi$, if $\psi$ abides the restrictions then we can apply our rewrite rules on $\psi$.

#### 3.4.1 Properties of fixpoints

**Definition 4.** In a term $\phi$, 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.

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

**Definition 6.** A fixpoint term $\mu(X = \phi)$ is said:
• positive when for all subterms \( \varphi_1 \supset \varphi_2 \) of \( \varphi, \varphi_2 \) is constant in \( \mathbf{X} \);
• linear when for all subterms \( \varphi \) of the form \( \varphi_1 \supset \varphi_2 \) or \( \varphi_1 \supset \varphi_2 \), either \( \varphi_1 \) or \( \varphi_2 \) is constant in \( \mathbf{X} \);
• mutually recursive when there exists a subterm \( \mu(\mathbf{X} = \psi) \) of \( \varphi \) with \( \mathbf{X} \) free in \( \psi \).

**Proposition 1.** If \( \mu(\mathbf{X} = \varphi) \) is linear, positive and non mutually recursive then the function \( f(\mathbf{S}) = \new{\varphi}_{V[X/S]} \) is such that:

\[
f(\mathbf{S}) = f(\emptyset) \cup \bigcup_{x \in \mathbf{S}} f(\{x\})
\]

and thus \( f \) has a fixpoint with \( \new{\mu(\mathbf{X} = \varphi)}_V = f^\infty(\emptyset) \).

**Proof Sketch.** We prove by induction on the subterms \( \xi \) of \( \varphi \) where \( \mathbf{X} \) is free in \( \xi \) that \( \new{\xi}_{V[X/S]} = \new{\xi}_{V[X/\emptyset]} \cup \bigcup_{x \in \mathbf{S}} \new{\xi}_{V[X/\{x\}]} \). By linearity, such a \( \xi \) can only be combined with a constant term and by positivity, it cannot be negated.

3.4.2 Expressivity of restricted fixpoints. These restrictions do have an effect on the expressivity of our language. We can show that \( \mu \text{-RA} \) has, at least, the expressive power of inflationary-Datalog\(^-\) (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\(^-\). Finally with all our restrictions, our language has exactly the expressive power of linear datalog (see § 2.2). This fragment is expressive enough to capture a lot of interesting queries. For instance, the next section presents how to translate UCRPQs into \( \mu \text{-RA} \) (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-\( \mu \)-RA of \( \mu \)-RA containing only terms where all the fixpoints are linear, positive and non mutually recursive.

3.5 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 \( \varphi \) linear and positive in \( \mathbf{X} \), we say that \( \varphi \) is recursive in \( \mathbf{X} \) when \( \text{rec}(\varphi, \mathbf{X}) = \top \) with \( \text{rec} \) defined as:

\[
\begin{align*}
\text{rec}(\varphi_1 \cup \varphi_2, \mathbf{X}) &= \text{rec}(\varphi_1, \mathbf{X}) \land \text{rec}(\varphi_2, \mathbf{X}) \\
\text{rec}(\varphi_1 \Rightarrow \varphi_2, \mathbf{X}) &= \text{rec}(\varphi_1, \mathbf{X}) \lor \text{rec}(\varphi_2, \mathbf{X}) \\
\text{rec}(\varphi_1 \triangleright \varphi_2, \mathbf{X}) &= \text{rec}(\varphi_1, \mathbf{X}) \\
\text{rec}(\varphi_1, \mathbf{X}) &= \text{rec}(\varphi_1, \mathbf{X}) \\
\text{rec}(\sigma(\varphi), \mathbf{X}) &= \text{rec}(\varphi, \mathbf{X}) \\
\text{rec}(\pi_{\text{src}}(\varphi), \mathbf{X}) &= \text{rec}(\varphi, \mathbf{X}) \\
\text{rec}(\pi_{\text{trg}}^m(\varphi), \mathbf{X}) &= \text{rec}(\varphi, \mathbf{X}) \\
\text{rec}(\mu(\mathbf{X} = \psi), \mathbf{X}) &= \bot \\
\text{rec}(\mathbf{X}, \mathbf{X}) &= \mathbf{X} = \mathbf{X} \\
\text{rec}(\mathbf{c} \rightarrow v, \mathbf{X}) &= \bot
\end{align*}
\]

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 \( \varphi \) be a term.

• If \( \varphi \) is recursive in \( \mathbf{X} \) then for all \( \mathbf{V} \), \( \new{\varphi}_{V[X/\emptyset]} = \emptyset \).

• If \( \varphi \) is constant in \( \mathbf{X} \), then \( \varphi \) does not depend on \( \mathbf{X} \), i.e. for all \( \mathbf{S} \) and \( \mathbf{V} \), \( \new{\varphi}_{V[X/S]} = \new{\varphi}_{V[X/\emptyset]} \).

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

**Example 1.** Let us suppose that we want to compute the transitive closure of a binary relation \( \mathbf{R} \). We assume \( \mathbf{R} \) is represented as a two-column table of type \( \{\text{src, trg}\} \). The closure is captured by the term \( \mu(\mathbf{X} = R \cup \pi_{\text{src}}(\rho_{\text{trg}}^m(R) \Rightarrow \rho_{\text{src}}^m(X))) \).

This term is a decomposed fixpoint: \( \mathbf{R} \) is constant and \( \pi_{\text{src}}(\rho_{\text{trg}}^m(R) \Rightarrow \rho_{\text{src}}^m(X)) \) is recursive in \( \mathbf{X} \).

**Proposition 2.** A fixpoint term \( \mu(\mathbf{X} = \varphi) \), linear, positive and non mutually recursive can be rewritten to either: an empty term, a term \( \varphi \) with one less fixpoint or a decomposed fixpoint.

In the following, we assume that fixpoints are always decomposed.

3.6 Type System

Given a schema \( \Gamma \) for a set of relation variables \( \mathbf{R} \), we can infer types for terms whose free variables are in \( \mathbf{R} \). The typing judgement \( \Gamma \vdash \varphi : t \) means that when evaluated in an environment conforming to the schema \( \Gamma \), \( \varphi \) will yield a relation of type \( t \). Our type system is defined by the rules on Fig. 3; it is quite straightforward. The rule for typing fixpoints uses the fact that fixpoints are decomposed to infer first the type of the constant part and then use that information to typecheck the recursive part. Given a database schema \( \Gamma \), we write \( \forall \Gamma \) for the set of well-typed terms in \( \Gamma \) (i.e. the terms \( \varphi \) such that \( \Gamma \vdash \varphi : t \) holds for some \( t \).

**Proposition 3.** Given a database \( (\mathbf{R}, \Gamma, D) \) and \( \varphi \in \forall \Gamma \), if \( \Gamma \vdash \varphi : t \) then the relation \( \new{\varphi}_D \) has type \( t \).

**Example 2.** Consider again the term of Example 1: \( \mu(\mathbf{X} = R \cup \pi_{\text{src}}(\rho_{\text{trg}}^m(R) \Rightarrow \rho_{\text{src}}^m(X))) \). Assuming \( \mathbf{R} \) is of type \( \{\text{src, trg}\} \), we can conclude the whole term is also of type \( \{\text{src, trg}\} \).
Indeed, the fixpoint should have the type of its constant part, which is $R$. Then we can check the type of the recursive part: $ho_{\text{trg}}^m(R)$ has type $\{\text{src, m}\}$ and is joined with $\rho_{\text{src}}^m(X)$ of type $\{\text{trg, m}\}$. The result has type $\{\text{src, trg, m}\}$ but the $m$ column is discarded by the $\pi_m(\ldots)$; thus the whole term is well typed.

4 GENERATING NEW QUERY PLANS

The traditional RA has rewrite rules and the optimization of relational queries is usually done by rewriting to a (estimated) more efficient term using rewrite rules. For instance the rule:

$$\sigma_i(\varphi \Rightarrow \psi) \Rightarrow \sigma_i(\varphi) \Rightarrow \psi$$

describes that a filter can be pushed inside a join. This rule applies whenever $FC(t) \subseteq t$ where $t$ is the type of $\varphi$.

In this section, we discuss properties of rest-$\mu$-RA which allow us to introduce new rewrite rules for recursive terms. These rules extend the set of RA classical rewrite rules, which remain valid on rest-$\mu$-RA. We first describe the new rules informally and briefly explain how they can lead to terms that are more efficiently evaluated. Then in § 4.2 we introduce the “set of derivations” for a term which allows us to describe when the two first rules are correct. Finally, in § 4.3 we introduce “addable columns” which allow us to introduce the conditions for three other rewrite rules to apply.

4.1 The new rewrite rules

4.1.1 Pushing filters and anti-join into fixpoints:

$$\sigma_i(\mu(X = \varphi)) \Rightarrow \mu(X = \sigma_i(\varphi)) \tag{RW1}$$

$$\mu(X = \varphi) \Rightarrow \psi \Rightarrow \mu(\varphi \Rightarrow \psi) \tag{RW2}$$

This always reduces the amount of mappings manipulated by the fixpoint. For example, on the RPQ $\exists x \ isLocatedIn+ \ Japan$, the first rule permits to only compute the $?x$ where $?x \ isLocatedIn+ \ Japan$, instead of computing the whole transitive closure $isLocatedIn+$ and then filtering out the pairs not in Japan.

4.1.2 Pushing joins into fixpoints:

$$\mu(X = \varphi) \Rightarrow \psi \Rightarrow \mu(\varphi \Rightarrow \psi) \tag{RW3}$$

This can also decrease the number of mappings solutions of the fixpoint. This rewrite rule can typically be used on RPQs such as $Q_ex$. In this query, there is no need to compute the whole set of pairs (?x,?y) such that ?x isLocatedIn+ ?y, as only the ?x such that Emmy_Noether worksAt ?x are of interest to us. Rule RW3 can be used here to push Emmy_Noether worksAt ?x into the fixpoint of ?x isLocatedIn+ ?y.

4.1.3 Merging fixpoints:

$$\mu(X = \kappa \cup \psi) \Rightarrow \mu(X = \kappa \Rightarrow \psi) \tag{RW4}$$

This reduces the number of fixpoints and the number of mappings. For instance, on $Q_2$: instead of computing $isLocatedIn+$ and $dealsWith+$ separately and then joining them, rule RW4 permits to first compute $isLocatedIn/dealsWith$ and then to add steps $isLocatedIn$ on the left or steps $dealsWith$ on the right.

4.1.4 Pushing antiprojections into fixpoints:

$$\overline{\pi}_p(\mu(X = \varphi)) \Rightarrow \mu(X = \overline{\pi}_p(\varphi)) \tag{RW5}$$

This can also reduce the number of mappings: the value of the removed column being ignored, several mappings might get merged (due to set semantics). This is typically useful for RPQs in which a fixpoint is joined or filtered and the value of the join or filter variable is discarded.

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

4.2 Set of derivations of a term

In general, we do not have $\sigma_i(\mu(X = \varphi)) \equiv \mu(X = \sigma_i(\varphi))$. Indeed, it is possible that some mappings solution of $\mu(X = \varphi)$ do not pass the filter but are still useful to create mappings (with the fixpoint iteration) which do pass the filter.

To rule out this possibility, we can use the following criterion: if the filter depends only on columns which are untouched by the fixpoint iteration, then applying the filter before or after the iteration is equivalent. To check this criterion syntactically, we introduce the notion of derivation, which describes which columns in the result tuples depend on which columns in the initial ones. This then allows us to define the stabilizer of a fixpoint, the set of columns which are untouched during the iteration.

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

$$d(\varphi_1 \cup \varphi_2, X) = d(\varphi_1, X) \cup d(\varphi_2, X)$$

$$d(\varphi_1 \triangleright \varphi_2, X) = d(\varphi_1, X)$$

$$d(\varphi_1 \triangleright \varphi_2, X) = d(\varphi_1, X) \cup d(\varphi_2, X)$$

$$d(\rho^m_\varphi(R), X) = \{ p \circ (b \mapsto a, a \mapsto \bot) \mid p \in d(\varphi, X) \}$$

$$d(\Pi_{\varphi}(X), X) = \{ p \circ (a \mapsto \bot) \mid p \in d(\varphi, X) \}$$

$$d(\sigma(\varphi), X) = d(\varphi, X)$$

$$d(\mu(Y = \varphi), X) = \emptyset$$

$$d(X, X) = \{ X \}$$

$$d(c \rightarrow v, X) = \emptyset$$

where $\circ$ represents the composition and $(a_1 \rightarrow b_1, \ldots, a_n \rightarrow b_n)$ represents the function that maps each $a_i$ to its $b_i$ and every other column 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 $src \rightarrow \bot, m \rightarrow \bot$ and everything else to itself. In particular $trg$ is mapped to itself.

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

$$\left( p(c) \neq \bot \lor \left( p(c) \notin dom(w) \right) \lor \left( m(c) = w(p(c)) \right) \right)$$

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

Lemma 3. Given a fixpoint term $\mu(X = \varphi) \in \mathcal{F} [\Gamma]$ of type $t$ and a mapping $w$ of type $t$, $w \in [\mu(X = \varphi)]_V$ if and only if there exists a lineage for $w$, that is, a finite sequence $w_0, w_1, \ldots, w_n$ such that: $w_0 \in [\varphi]^V[X/\emptyset], w_{i+1} \in [\varphi]^V[X/\{w_i\}]$ and $w_n = w$.

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

4.2.2 Application to rewrite rules $RW1$ and $RW2$.

Theorem 1 (Pushing filters). Let $\mu(X = \kappa \cup \psi)$ be a decomposed fixpoint term, $V$ an environment and $t$ a filter condition with $FC(t)$ and by definition of $FC$, $eval(t, w_0) = eval(t, w)$.

Example 1 followup. In our previous example, $trg$ was in the stabilizer and $src$ was not. Indeed the term $\mu(X = R \cup \Pi_m(\rho^m trg(R) \triangleright \rho^m src(X)))$ computes the pairs $(src, trg)$ in the transitive closure of $R$ by taking a pair $(src, trg)$ already in the closure then finding a link $(src', src)$ in $R$ and finally adding $(src', trg)$ to the set of solutions.

On this term, pushing a filter on $trg$ is possible but not on $src$: a solution $(src, trg)$ might not pass the filter but be useful to discover a solution $(src', trg)$ passing the filter.

4.3 Addable columns to a fixpoint

Let us assume a fixpoint $\varphi = \mu(X = \kappa \cup \psi)$ such that $\Gamma \vdash \varphi : t$. $t$ is thus the type of $\kappa$. Suppose we change the constant part $k$ in such a way that it has no 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 $\psi$. For this, we introduce the predicate $add(\psi, X, c)$ which checks syntactically that $\psi$ does not depend on $X$ having a column named $c$.

4.3.1 Logical framework.

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

$$add(\varphi_1 \cup \varphi_2, X, c) = add(\varphi_1, X, c) \land add(\varphi_2, X, c)$$

$$add(\varphi_1 \triangleright \varphi_2, X, c) = add(\varphi_1, X, c) \land add(\varphi_2, X, c)$$

$$add(\rho^m_\varphi(R), X, c) = add(\varphi, X, c) \land c \notin \{a, b\}$$

$$add(\Pi_{\varphi}(X), X, c) = add(\varphi, X, c) \land c \notin FC(t)$$

$$add(\mu(\psi), X, c) = add(\mu, X, c)$$

$$add(R, X, c) = c \notin \Gamma(R) \text{ when } X \neq R$$

$$add(X, X, c) = \top$$

$$add(c \rightarrow v, X, c) = c \neq c'$$

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

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

- $c \in stab(\psi, X)$
- $\Gamma \cup \{X \rightarrow t \cup \{c\}\} \vdash t \cup \{c\}$
- $[\psi \triangleright t \rightarrow v]_V[X/\{w\}] = [\psi]_V[X/\{w(v)\}]$
Proof sketch. The first point can be proved inductively by definition of \( \text{stab} \) and \( \text{add} \). The second point is a consequence of the first. The third point can be proved by induction on the size of \( \psi \).

4.3.2 Application to rewrite rules RW3, RW4 and RW5.

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

1. \( t_\varphi \subseteq \text{stab}(\psi, X) \)
2. \( \forall c \in t_\varphi \setminus t_\kappa \text{ add}(\psi, X, c) \)

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

\[
\left[ \varphi \triangleright \mu(X = \kappa \cup \psi) \right]_V = \left[ \mu(X = \kappa \cup \varphi \cup \psi) \right]_V
\]

Proof sketch. First we prove that \( \psi \in \mathcal{F}[\Gamma \cup \{ X \rightarrow t_\kappa \sim t_\varphi \}] \) by iterating Lemma 4.1. Then for each lineage \( w_0, \ldots, w_n \in \left[ \mu(X = \kappa \cup \psi) \right]_V \) 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 \( w + v \in \left[ \mu(X = \varphi \triangleright \kappa \cup \psi) \right]_V \).

The reverse direction is proved similarly. □

**Theorem 4 (Merging Fixpoints).** Given two decomposed fixpoints \( \mu(X = k_1 \cup \psi_1) \) and \( \mu(X = k_2 \cup \psi_2) \) of types \( t_1 \) and \( t_2 \) such that:

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

then we have:

\[
\left[ \mu(X = k_1 \cup \psi_1) \triangleright \mu(X = k_2 \cup \psi_2) \right]_V = \left[ \mu(X = k_1 \triangleright k_2 \cup \psi_1 \cup \psi_2) \right]_V
\]

Proof sketch. The forward direction is easy: given two lineages \( w_0, \ldots, w_n \) and \( w'_0, \ldots, w'_m \) (for both \( [\mu(X = k_1 \cup \psi_1)]_V \)) we can build a lineage \( (w_0 + w'_0), \ldots, (w_n + w'_m) \).

The converse direction is more difficult but we can deinterlace the lineages and create two lineages, one for each \( [\mu(X = k_1 \cup \psi_1)]_V \)

□

Example 3. If we want to compute \( R^+_1(x, y) \land R^+_2(y, z) \), 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 \( R_1(y, z) \land R_1(x, y) \) and then will discover new \( x \) or new \( z \) by a fixpoint: \( \mu(X = R_1 \triangleright R_2 \cup \psi) \) with \( \psi = \pi_{ie}(\rho_{\kappa_1}^e(X) \triangleright \rho_{\kappa_2}^e(R_1)) \).

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

\[
\left[ \pi_b(\mu(X = \kappa \cup \psi)) \right]_V = \left[ \mu(X = \pi_b(\kappa) \cup \psi) \right]_V
\]

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

**Example 1 Followup.** In our running example, \( \text{trg} \) is an addeable column. This means that, if we are interested in \( \pi_{\text{trg}}(\mu(X = R \cup \pi_m(R) \triangleright \rho_{\text{src}}^m(X))) \) we can push the \( \pi_{\text{trg}}(\cdot) \) and obtain \( \mu(X = \pi_{\text{trg}}(R \cup \pi_m(R) \triangleright \rho_{\text{src}}^m(X))) \). Pushing \( \pi_{\text{trg}}(\cdot) \) will reduce drastically the number of solutions. We cannot push \( \pi_{\text{src}}(\cdot) \) as the column \( \text{src} \) is used.

5 GRAPH QUERY TRANSLATIONS

\( \mu \)-RA can be used to model queries over directed graphs with labeled edges. We assume that the set of values \( \mathcal{B} \) gathers both vertices and edge labels. The graph can then be represented as a pair \( (V, E) \) with \( V \subset \mathcal{B} \) denoting the set of vertices and \( E \subset \mathcal{B} \times \mathcal{B} \times \mathcal{V} \) 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 \( \Gamma = \{ V \rightarrow \{ \text{src}, E \rightarrow \{ \text{src}, l, \text{trg} \} \} \) where \( \text{src}, l, \text{trg} \) stand respectively for source, label, target.

Regular path queries (RPQs) \( [3, 23, 27, 32, 52] \) 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 \( K \subseteq \mathcal{B} \) 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:

\[
\begin{align*}
\text{r} &::= \text{L} & \text{a single edge label} \\
| r_1 \cup r_2 & \text{ concatenation} \\
| r_1 \triangleright r_2 & \text{ alternative} \\
| r^{-1} & \text{ reverse} \\
| r^+ & \text{ transitive closure}
\end{align*}
\]

For example, the sample query \( Q_2 \) given in the introduction is an RPQ. The basic component for translating graph queries into \( \mu \)-RA is the translation of the regular path expressions \( r \). We translate any \( r \) into a set of \( \mu \)-RA terms \( \phi \), representing one or more alternative translations, in such a way that the result of evaluating any of those \( \phi \) on the graph database is the set of all matchings \( \{ \text{src} \rightarrow v_1, \text{trg} \rightarrow v_2 \} \) 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 \( \langle \cdot \rangle \) that compiles any \( r \) into \( \mu \)-RA, as follows:

\[
\begin{align*}
\langle \text{L} \rangle & = \{ \pi_{\text{src}}(\kappa_1(\psi)) \} \\
\langle r_1 \cup r_2 \rangle & = \{ \pi_m(\rho_{\text{src}}^m(\phi_1) \triangleright \rho_{\text{src}}^m(\phi_2)) | \phi_1 \in \langle r_1 \rangle \land \phi_2 \in \langle r_2 \rangle \} \\
\langle r_1 \triangleright r_2 \rangle & = \{ \phi_1 \land \phi_2 | \phi_1 \in \langle r_1 \rangle \land \phi_2 \in \langle r_2 \rangle \} \\
\langle r^{-1} \rangle & = \{ \pi_{\text{src}}(\rho_{\text{trg}}^m(\phi), \kappa_1) \} \\
\langle r^+ \rangle & = \{ \mu(X = \phi \cup \pi_m(\rho_{\text{trg}}(\phi) \triangleright \rho_{\text{src}}^m(X))) | \phi \in \langle r \rangle \} \cup \{ \mu(X = \phi \cup \pi_m(\rho_{\text{trg}}^m(\phi) \triangleright \rho_{\text{src}}^m(X))) | \phi \in \langle r \rangle \}
\end{align*}
\]
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\textsuperscript{1}. This is the reason why $\langle r \rangle$ 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.

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 $\Pi H \leftarrow C_1 \cup \ldots \cup C_n$ in which each disjunct $C_i$ is a conjunction as defined previously. We translate a UCRPQ into $\mu$-RA as follows:

$$\langle H \leftarrow C_1 \cup \ldots \cup C_n \rangle = \langle C_1 \rangle_H \cup \langle C_2 \rangle_H \ldots \cup \langle C_n \rangle_H$$

with:

$$\langle C \rangle_H = \left\{ \phi_1 \Rightarrow \phi_2 \Rightarrow \ldots \Rightarrow \phi_n \mid (\phi_1, \phi_2, \ldots, \phi_n) \in \text{combine}(C)_H \right\}$$

The key point is the set of fixpoint subterms in the query. Each of those will always be computed in full into a temporary table by the engine, since it has no internal rules to move other operations in or out of the recursion block. Based on this observation, we estimate a cost for each $\mu$-RA term and pick one of the terms with the minimal estimated cost (see below).

We have two possible translations for fixpoint terms. When a fixpoint $\mu(X = \kappa \cup \psi)$ is such that $X$ appears exactly once in $\psi$, which is always the case for terms generated by translating a UCRPQ, this fixpoint can be translated into a CREATE RECURSIVE VIEW statement, which is purely declarative so that the engine can freely decide how to compute it. This allows us to test PostgreSQL on the initial translated query without restricting its plan space (this test is what we call ‘System P’ below). However, the rewriting rule for merging fixpoints generates terms where $X$ appears twice; these terms cannot be translated into pure SQL. In that case, we resort to procedural language (PL/pgSQL) and a WHILE loop which computes a temporary table containing the result of the fixpoint term.

Cost estimation. We use a simple cost estimation based on\textsuperscript{2} [60], extended to deal with fixpoint terms. It uses a few parameters: number of tuples in each relation in the database, number of distinct values for each column, and histogram values for selectivity\textsuperscript{3} [42]. These parameters are available from the PostgreSQL server where the data is stored. For a fixpoint term $\mu(X = \kappa \cup \psi)$, we assume that the number of new tuples generated is reduced, at each iteration, by a factor $s$ which we compute from selectivity values and the operations in $\psi$. As the iteration stops when no new tuples are generated, we estimate the number of steps as $\log_s(K)$ where $K$ is the estimated number of tuples in $\kappa$. This estimated number of steps allows us to estimate, in turn, the number of tuples in $\mu(X = \kappa \cup \psi)$ and the CPU time required.

6 EXPERIMENTAL WORK

6.1 Prototype

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 novel rewriting rules. To test these rules, we implemented a prototype executor for UCRPQs as follows. First, queries are translated into $\mu$-RA terms. Then, we enumerate equivalent terms using a Volcano-style strategy\textsuperscript{4}. We then pick one of the equivalent terms, translate it into SQL, and send it to PostgreSQL. Physical plan selection and evaluation are left to the PostgreSQL engine; however, depending on the $\mu$-RA term we pick, this engine will not be able to access the same plan space.

6.2 Methodology

6.2.1 Queries. 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\textsuperscript{5} [15] to randomly generate UCRPQs with corresponding synthetic datasets.

Queries over the yago dataset. We use a cleaned version of the real world dataset yago2s\textsuperscript{6}, that we have preprocessed

\textsuperscript{1}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.

\textsuperscript{2}Estimated number of distinct values for each column.

\textsuperscript{3}Selectivity is approximated from histogram values.

\textsuperscript{4}Volcano is a well-known algorithm for cost-based query optimization.

\textsuperscript{5}GMark is a benchmark suite for evaluating graph query engines.

\textsuperscript{6}Yago2s is a variant of the Yago knowledge base.
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: we collected 7 RPQs from [5], 2 from [70] and 2 from [41]. Since all of them are RPQs, we supplemented them with additional CRPQs to illustrate more complex forms of recursion. Fig. 4 presents the 20 queries tested with the yago dataset.

\[
\begin{align*}
?x & \leftarrow ?x \text{isMarriedTo}/\text{livesIn}/\text{isl}+/\text{dw}+ \text{Argentina} & Q_1 \\
?x & \leftarrow ?x \text{hasChild}/\text{livesIn}/\text{isl}/+\text{dw} \text{Japan} & Q_2 \\
?x & \leftarrow ?x \text{influences}/\text{livesIn}/\text{isl}/+\text{dw} \text{Sweden} & Q_3 \\
?x & \leftarrow ?x \text{livesIn}/\text{isl}/+\text{dw} + \text{United States} & Q_4 \\
?x & \leftarrow ?x \text{hasSuccessor}/\text{livesIn}/\text{isl}/+\text{dw} \text{India} & Q_5 \\
?x & \leftarrow ?x \text{hasPredecessor}/\text{livesIn}/\text{isl}/+\text{dw} \text{Germany} & Q_6 \\
?x & \leftarrow ?x \text{has}/\text{livesIn}/\text{isl}/+\text{dw} \text{Netherlands} & Q_7 \\
?x & \leftarrow ?x \text{isl}/+\text{dw} \text{United States} & Q_8 \\
?x & \leftarrow ?x \text{(actedIn}/-\text{actedIn})+ \text{Kevin_Bacon} & Q_9 \\
\text{?area} & \leftarrow \text{wce -type}/\text{isl}/+\text{dw} \text{Japan} & Q_{10} \\
\text{?person} & \leftarrow \text{?person}\text{isMarriedTo}/+\text{owns}/\text{isl}+\text{owns}/\text{isl}+ \text{USA} & Q_{11} \\
\text{?a, ?b} & \leftarrow \text{?a}\text{isl}/+\text{dw} \text{7b} & Q_{12} \\
\text{?a, ?b} & \leftarrow \text{?a}\text{isl}/+\text{dw} \text{7b} & Q_{13} \\
\text{?a, ?b, ?c} & \leftarrow \text{?a}\text{wasBornIn}/\text{isl}+ \text{7b}, \text{?b}\text{isConnectedTo}+ \text{7c} & Q_{14} \\
\text{?a, ?b, ?c} & \leftarrow \text{?a}\text{(isLocatedIn}+ \text{7b}) \text{?b}\text{wasBornIn} \text{7c} & Q_{15} \\
\text{?a, ?b, ?c} & \leftarrow \text{?a}\text{isLocatedIn}+ \text{7b}, \text{?b}\text{isConnectedTo}+ \text{7c} & Q_{16} \\
\text{?a} & \leftarrow \text{?a}\text{isl}+/\text{isConnectedTo}+\text{dw} \text{Japan} & Q_{17} \\
\text{?a, ?c} & \leftarrow \text{?a}\text{isl}+/\text{isConnectedTo}+\text{dw} \text{Japan} & Q_{18} \\
\text{?a} & \leftarrow \text{?a}\text{isl}+/\text{isConnectedTo}+\text{dw} \text{Japan} & Q_{19} \\
\text{?a} & \leftarrow \text{?a}\text{isl}+/\text{isConnectedTo}+\text{dw} \text{Japan} & Q_{20}
\end{align*}
\]

Figure 4: Queries for the yago dataset.

"Q1…Q20 are taken from [5], Q6 from [70], and Q10, Q11 from [41]. "isl" stands for "isLocatedIn," "dw" for "dealsWith," "haa" for "hasAcademicAdvisor," and "wce" for "wikicat_Capitals_in_Europe."

Generated queries over synthetic datasets, using gMark. We gathered queries generated by gMark [15], which are available on the gMark open source repository [16]. 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 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.

6.2.2 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 [56, 63] implementing SQL with recursive views.
- system P2: our prototype extending the PostgreSQL system with our optimizations.
- system V: the Virtuoso graph column store [30] which is backed by a relational database, and that implements the SPARQL 1.1 language [43] (with property paths).
- system L: a modern engine implementing Datalog [66].
- system N: a native graph database implementing the openCypher graph query language [40, 55].

### Table 1: Dataset Statistics.

<table>
<thead>
<tr>
<th>Dataset Origin</th>
<th>Predicates</th>
<th>Edges</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>yago 2.5 (cleaned)</td>
<td>83</td>
<td>62,643,951</td>
<td>42,832,856</td>
</tr>
<tr>
<td>gMark-Shop [15]</td>
<td>81</td>
<td>209,789</td>
<td>135,737</td>
</tr>
<tr>
<td>gMark-Uniprot [15]</td>
<td>7</td>
<td>76,707</td>
<td>21,130</td>
</tr>
</tbody>
</table>

6.3 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).²

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 theoreti- cally 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. Thus, for a given query, the time reported

²Experiments 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 the latest public Docker images for each tested system.

³Specifically, 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.
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 [38, 39, 47, 62], 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.

6.4 Results

6.4.1 Queries over the YAGO dataset. Fig. 5 shows the time spent with each system for evaluating each query of Fig. 4. The time scale is logarithmic. Whenever no time is reported (e.g. for generating the plan space – a task which can benefit from extensive research and various techniques found in the literature [38, 39, 47, 62], 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.

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. 6 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).

<table>
<thead>
<tr>
<th>Feasible queries</th>
<th>$N$</th>
<th>$V$</th>
<th>$L$</th>
<th>$P$</th>
<th>$P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2$</td>
<td>$10$</td>
<td>$18$</td>
<td>$19$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>$P$</th>
<th>$N$</th>
<th>$V$</th>
<th>$L$</th>
<th>$P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$</td>
<td>$184.3$</td>
<td>$99.8$</td>
<td>$127.0$</td>
<td></td>
</tr>
<tr>
<td>$Q_2$</td>
<td>$227.8$</td>
<td>$∞$</td>
<td>$151.4$</td>
<td></td>
</tr>
<tr>
<td>$Q_3$</td>
<td>$212.2$</td>
<td>$61.5$</td>
<td>$145.7$</td>
<td></td>
</tr>
<tr>
<td>$Q_4$</td>
<td>$2.1$</td>
<td>$∞$</td>
<td>$∞$</td>
<td></td>
</tr>
<tr>
<td>$Q_5$</td>
<td>$194.4$</td>
<td>$∞$</td>
<td>$134.8$</td>
<td></td>
</tr>
<tr>
<td>$Q_6$</td>
<td>$182.6$</td>
<td>$∞$</td>
<td>$125.1$</td>
<td></td>
</tr>
<tr>
<td>$Q_7$</td>
<td>$424.3$</td>
<td>$∞$</td>
<td>$446.7$</td>
<td></td>
</tr>
<tr>
<td>$Q_8$</td>
<td>$1.0$</td>
<td>$∞$</td>
<td>$∞$</td>
<td></td>
</tr>
<tr>
<td>$Q_9$</td>
<td>$∞$</td>
<td>$∞$</td>
<td>$∞$</td>
<td></td>
</tr>
<tr>
<td>$Q_{10}$</td>
<td>$2.2$</td>
<td>$∞$</td>
<td>$0.01$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Speedup with $P'$ for YAGO queries.

A speedup $x > 1$ means query evaluation is $x$ times faster, $x = 1$ means no speedup, $∞$ means $P'$ is always better than $P$, and $⊥$ means query evaluation is $x$ times slower, $x < 1$ means no speedup. $⊥$ means the query remains unfeasible, and $∞$ denotes cases where a formerly unfeasible query becomes feasible.

6.4.2 Generated queries over synthetic datasets. Fig. 7 and 8 show the times spent in evaluating recursive queries of the “UniProt” and the “Shop” gMark scenarios [15], 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 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.
Figure 5: Elapsed time for evaluating queries of Fig. 4 over the yago real-world dataset.

Figure 6: Number of results retrieved (with set semantics) by each query of Fig. 4 on the yago dataset.

Figure 7: Time spent for evaluating recursive queries of the synthetic “UniProt” gMark scenario.

Figure 8: Time spent for evaluating recursive queries of the synthetic “Shop” gMark scenario.
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 execution 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

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


A PROOFS FOR SECTION 3 (THE $\mu$-EXTENDED REL. ALGEBRA)

Proposition (1). If $\mu(X = \phi)$ is linear, positive and non mutually recursive then the function $f(S) = \llbracket \phi \rrbracket_{V[X/S]}$ is such that:

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

and thus $f$ has a fixpoint with $\llbracket \mu(X = \phi) \rrbracket_V = f^\omega(\emptyset)$.

Proof. We will first prove by induction on the size of terms the following property: given a valid term $\phi$, for all $S \not= \emptyset$ we have $\forall m \in \llbracket \phi \rrbracket_{V[X/S]} \exists w_m \in S \ m \in \llbracket \phi \rrbracket_{V[X/w_m]}$.

- Using lemma 1 the property is clearly true for terms $\phi$ such that $X$ is not free in $\phi$. And the only relation variable where $X$ appears free is $X$. For $X$ the property trivially holds (with $w_m = m$).
- For unary operators $\phi \in \{\rho^x_\phi(p_1), \bar{\tau}_a(\phi)\}$ we have $m \in \llbracket \phi \rrbracket_{V[X/S]}$ implies the existence of $m' \in \llbracket \phi \rrbracket_{V[X/S]}$ such that $m$ is the image of $m'$ through this operator. By the induction hypothesis, for $m'$ there is $w$ such that $m' \in \llbracket \phi \rrbracket_{V[X/w_m]}$ and thus $m \in \llbracket \phi \rrbracket_{V[X/w_m]}$.
- For a join operator $\phi = \phi_1 \bowtie \phi_2$ we have by linearity that $\phi$ is constant in $X$ for some $i$ (let us note $i = 3 - i$). If $\phi_i$ is also constant in $X$ then $\phi$ is constant in $X$ so we can refer to the first item. Otherwise, $m \in \llbracket \phi \rrbracket_{V[X/S]}$ implies the existence of $m_1 \in \llbracket \phi_1 \rrbracket_{V[X/S]}$ and $m_2 \in \llbracket \phi_2 \rrbracket_{V[X/S]}$ such that $m = m_1 + m_2$. By induction, there exists $w_{m_1}$ such that $m_1 \in \llbracket \phi_1 \rrbracket_{V[X/w_{m_1}]}$. For $i$ we have $\llbracket \phi_i \rrbracket_{V[X/S]} = \llbracket \phi_i \rrbracket_{V[X/w]} = \llbracket \phi_i \rrbracket_{V[X/w_m]}$ which means that in any case $m_1 \in \llbracket \phi_1 \rrbracket_{V[X/w_m]}$, $m_2 \in \llbracket \phi_2 \rrbracket_{V[X/w_m]}$ and thus $m \in \llbracket \phi \rrbracket_{V[X/w_m]}$.
- For the term $\phi = \phi_1 \bowtie \phi_2$ with any mapping $m \in \llbracket \phi \rrbracket_{V[X/S]}$ is built using at least one mapping $m_1$ from $\llbracket \phi_1 \rrbracket_{V[X/S]}$: By induction, we have $w$ such that $m_1 \in \llbracket \phi_1 \rrbracket_{V[X/w]}$. But $X$ does not appear free in $\phi_2$, thus $\llbracket \phi_2 \rrbracket_{V[X/S]} = \llbracket \phi_2 \rrbracket_{V[X/w]}$ and thus $\llbracket \phi \rrbracket_{V[X/S]} = \llbracket \phi \rrbracket_{V[X/w]}$.
- For the term $\phi = \phi_1 \cup \phi_2$, $m \in \llbracket \phi \rrbracket_{V[X/S]}$ implies $m \in \llbracket \phi_1 \rrbracket_{V[X/S]}$ or $m \in \llbracket \phi_2 \rrbracket_{V[X/S]}$. By induction we have $w$ such that $m \in \llbracket \phi_1 \rrbracket_{V[X/w]}$ or $m \in \llbracket \phi_2 \rrbracket_{V[X/w]}$ and thus $m \in \llbracket \phi \rrbracket_{V[X/w]}$.

- Given the term $\mu(Y = \phi)$ we have $\mu(Y = \phi)$ constant in $X$ and thus the result by lemma 1.

Lemma (1). Let $\phi$ be a term.

- If $\phi$ is recursive in $X$ then for all $V$, $\llbracket \phi \rrbracket_{V[X/\emptyset]} = \emptyset$.
- If $\phi$ is constant in $X$, then $\phi$ does not depend on $X$, i.e. for all $S$ and $V$, $\llbracket \phi \rrbracket_{V[X/S]} = \llbracket \phi \rrbracket_{V[X]}$.

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 $\text{rec}$ this can only happen for $\phi_1 \bowtie \phi_2$, but one of the $\phi_i$ has to be recursive and since the join with an empty set leads to an empty set the result holds by induction.

Proposition (2). A fixpoint term $\mu(X = \phi)$, linear, positive and non mutually recursive can be rewritten to either: an empty term, a term $\phi$ with one less fixpoint or a decomposed fixpoint.

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

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

- For a term $\phi$ constant in $X$ the result is clear ($R = \emptyset, C = \phi$).
- For $X, R = X, C = \emptyset$.
- For a unary operator $f(\phi) \in \{\rho^x_\phi(p), \bar{\tau}_a(\phi), \sigma_t(\phi)\}$, we have $\phi$ that can be decomposed into $R, C, \phi$ the solution is $f(R, \phi), f(C, \phi)$ (where $f(s)$ represents $f(s)$ if $s$ is a term and $\emptyset$ otherwise).
- For a join $\phi_1 \bowtie \phi_2$, let us suppose by symmetry that $\phi_2$ is constant in $X$; then $\phi_1$ can be decomposed into $R, C$ and the result is $R \bowtie \phi, C \bowtie \phi$.
- 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.
We recall in this section that datalog programs can always be transformed to programs that have only one recursive rule and work and most of them are already known in the literature (with very similar statements and with similar proofs, see e.g. [4] for references).

To do that we simply take

\[ \text{Access}(X,c,c) \leftarrow \text{Access}(Y,c,c), \text{Path}(X,Y,c) \]
\[ \text{Access}(1,c,c). \]

\[ \text{Access}(X) \leftarrow \text{Access}(Y), \text{Path}(X,Y) \]
\[ \text{Path}(1,2,c). \]

\[ \text{Rec(access,X,c,c)} \leftarrow \text{Rec(access,Y,c,c)}, \text{Rec(path,X,Y,c)}. \]

Finally we use joins with constants for the constants of the head and renames for the variables.

Given a production

\[ \text{Rec(access,X,c,c)} \leftarrow \text{Rec(access,Y,c,c)}, \text{Rec(path,X,Y,c)}. \]

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]).

B.2 From a derivation rule to \( \mu \)-RA

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 Datalog with only one IBD

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:
\[ \text{Path}(1,2). \]
\[ \text{Access}(1). \]
\[ \text{Access}(X) \leftarrow \text{Access}(Y), \text{Path}(X,Y) \]
\[ \text{can be made 3-ary in the following way:} \]
\[ \text{Path}(1,2,c). \]
\[ \text{Access}(1,c,c). \]
\[ \text{Access}(X,c,c) \leftarrow \text{Access}(Y,c,c), \text{Path}(X,Y,c) \]

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:
\[ \text{Rec(path,1,2,c)}. \]
\[ \text{Rec(access,1,c,c)}. \]
\[ \text{Rec(access,X,c,c), Rec(path,X,Y,c).} \]
\[ \text{Output}(X) \leftarrow \text{Rec(access,X,c,c)}. \]

B.2 From a derivation rule to \( \mu \)-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 \( \text{head}(Y) : \neg \text{body}_1(X_1), \ldots, \text{body}_k(X_k) \) we can translate \( \text{body}_1(X_1), \ldots, \text{body}_k(X_k) \) using \( k - 1 \) joins between each \( \text{body}_i \), Renames to rename arguments of \( \text{body}_i \), 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 \( \text{Rec} \) into a term \( \text{Rec} \) that has 4 columns (\( a_1, a_2, a_3 \) and \( a_4 \)) the translation of the body
\[ \text{Rec(access,X,c,c), Rec(path,X,Y,c).} \]

is
\[ \rho_a^y(\bar{\pi}_a(\bar{\pi}_a(\bar{\pi}_a(\bar{\pi}_{a_1}(\text{access} \land a_3 = c \land a_4 = c (\text{Rec}) ))))) \]
\[ \Rightarrow \bar{\pi}_a(\bar{\pi}_a(\rho_{a_2}(\bar{\pi}_{a_1}(\text{access} \land a_3 = c (\text{Rec}) ))))) \]

The whole translation is (using \( \text{body} \) to denote the above term):
\[ \rho_a^y(\text{body}) \Rightarrow \rho_{a_2}(a_3 \rightarrow c) \Rightarrow a_4 \rightarrow c \Rightarrow a_1 \rightarrow \text{access} \]
B.3 From inflationary Datalog\(^{-}\) to \(\mu\)-RA

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

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

\[
\begin{align*}
B_1 &= |a_1 \rightarrow \text{path}| \Rightarrow |a_2 \rightarrow 1| \Rightarrow |a_3 \rightarrow 2| \Rightarrow |a_4 \rightarrow c| \\
B_2 &= |a_1 \rightarrow \text{access}| \Rightarrow |a_2 \rightarrow 1| \Rightarrow |a_3 \rightarrow c| \Rightarrow |a_4 \rightarrow c| \\
B_3 &= \rho^a_X(\pi_Y(\rho^Y_{a_2}(\pi_{a_1}(\pi_{a_1}(\sigma_{a_1=\text{access} \land a_3=\text{c} \land a_4=c}(\text{Rec})))))) \Rightarrow \rho^Y_{a_2}(\pi_{a_1}(\pi_{a_1}(\sigma_{a_1=\text{path} \land a_4=c}(\text{Rec})))) \\
B_4 &= \mu(\text{Rec} = B_1 \cup B_2 \cup B_3) \\
O &= \pi_{a_1}(\pi_{a_1}(\sigma_{a_1=\text{access}(B)_4}))
\end{align*}
\]

The semantics does coincide with inflationary-Datalog\(^{-}\) 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 \(\mu\)-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 \(\text{Rec}_i\). 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 \(\text{Rec}_j\) where \(j \leq i\). We translate \(\text{Rec}_j\) into \(X_i\) and the \(\text{Rec}_i\) into \(\mu(X_i = \varphi_i)\). Note that each \(\varphi_i\) can contain several occurrences of \(\text{Rec}_j\) 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):

\[
\begin{align*}
\text{Path}(\ldots) &\text{ an EDB} \\
\text{Access}_1(y) &\text{.} \\
\text{Access}_1(X) &\Rightarrow \text{Access}_1(Y), \text{Path}(Y,X) \\
\text{Access}_2(1) &\text{.} \\
\text{Access}_2(X) &\Rightarrow \text{Access}_2(Y), \text{Path}(Y,X), \text{ not } \text{Access}_1(Y) \\
\text{We translate } \text{Path} &\text{ into a term } \mu(X_0 = \varphi_0) \text{ (despite the fixpoint } \varphi_0 \text{ is actually not recursive as } \text{Path} \text{ is an EDB). Then we translate } \text{Access}_1:\n\mu(X_1 = |a_1 \rightarrow 0| \cup \rho^a_{a_1}(\pi_{a_1}(X_1 \Rightarrow \text{Path}))) \\
\text{Then we translate } \text{Access}_2 &\text{ (using } \text{Access}_1 \text{ to denote the term above):} \\
\mu(X_2 = |a_1 \rightarrow 1| \cup \rho^a_{a_1}(\pi_{a_1}(X_2 \Rightarrow \text{Path } \text{ Access}_1)))
\end{align*}
\]

B.5 From linear Datalog to rest-\(\mu\)-RA

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

B.6 From rest-\(\mu\)-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 \(\varphi\) to a Datalog rule. Datalog rules have columns that are indexed (there is a first column, a second, a third, etc.) while \(\mu\)-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 \( s_\varphi(T) \) corresponding the term \( \varphi \) \( s_\varphi(T) \) (where \( T \) is the ordered set of columns of the type of \( \varphi \)).

- For \( \varphi = \varphi_1 \bowtie \varphi_2 \) we create a rule for the join: \( s_\varphi(T) \leftarrow s_{\varphi_1}(T_1), s_{\varphi_2}(T_2) \).
- For \( \varphi = \varphi_1 \cup \varphi_2 \) we have two production rules, one for each \( \varphi_i \): \( s_\varphi(T) \leftarrow s_{\varphi_i}(T_i) \).
- For \( \varphi = \varphi_1 \rightarrow \varphi_2 \) we create the rule \( s_{\varphi_1}(T_1) \leftarrow s_{\varphi_2}(T_2) \).
- For \( \varphi = \sigma_{a=b}(\varphi') \) we create the rule \( s_\varphi(T_1, b, T_2) \leftarrow s_{\varphi'}(T_1, b, T_2) \) if we suppose that the ordered type of \( \varphi' \) is \( T_1, a, T_2 \).
- For \( \varphi = \pi_p(\varphi') \) we create the rule \( s_\varphi(T_p) \leftarrow s_{\varphi'}(T_p') \) where \( T' \) is \( T_p' \) where we inserted a \( a \) in the place of where \( b \) will be stored.
- For \( \varphi = \mu(X = \varphi') \) we create the rule \( s_X(T) \leftarrow s_{\varphi'}(T_{\varphi'}) \).
- For \( \varphi = X \) we create the rule \( s_X(T) \leftarrow s_{\varphi'}(T_{\varphi'}) \).

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 \( \varphi \) a term linear, positive and non mutually recursive in \( X \). For all \( m \in \| \varphi \|_{V[X/\varnothing]} \) either \( m \in \| \varphi \|_{V[X/w]} \) or there exists \( p \in d(\varphi, X) \) such that for all \( c \in \text{dom}(w) \):

\[
(p(c) = \bot) \lor \left( p(c) \notin \text{dom}(w) \right) \lor \left( m(c) = w(p(c)) \right)
\]

**Proof.** Let \( w \) and \( m \in \| \varphi \|_{V[X/\varnothing]} \setminus \| \varphi \|_{V[X/w]} \). By induction:

- Since \( m \) exists, \( X \) can only be free in \( \varphi \) (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 \( m \in \| \varphi_i \|_{V[X/\varnothing]} \) and thus the result.
- For a join \( \varphi_1 \bowtie \varphi_2 \) let us suppose by symmetry that \( \varphi_1 \) is not constant in \( X \) and that \( \varphi_2 \) is. We have \( m_1 \in \| \varphi_1 \|_{V[X/\varnothing]} \), \( d_1 \in d(\varphi_1, X) \subseteq d(\varphi_1 \bowtie \varphi_2, X) \) (with inductive hypothesis) and \( m_2 \in \| \varphi_2 \|_{V[X/\varnothing]} \).

For each \( c \in \text{dom}(w) \) we either have \( d_1(c) = \bot \) or \( d_1(c) \notin \text{dom}(w) \) 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 \( d_1(c) \neq \bot \land (p(c) \in \text{dom}(w)) \) 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 \( m \in \| \lambda(\varphi) \|_{V[X/\varnothing]} \) that implies \( m' \in \| \varphi \|_{V[X/\varnothing]} \) and \( d' \in d(\varphi, X) \) with the property. And \( d' \circ \lambda \) works.

\[ \Box \]

**Lemma (3).** Given a fixpoint term \( \mu(X = \varphi) \in \mathcal{F}[\Gamma] \) of type \( t \) and a mapping \( w \) of type \( t \), \( w \in \| \mu(X = \varphi) \|_V \) if and only if there exists a lineage for \( w \), that is, a finite sequence \( w_0, \ldots, w_n \) such that: \( w_0 \in \| \varphi \|_{V[X/\varnothing]} \), \( w_{i+1} \in \| \varphi \|_{V[X/w_i]} \), and \( w_n = w \).

Furthermore, for all lineages \( w_0, \ldots, w_n \) and all \( c \in t \cap \text{stab}(\varphi, X) \), we have for all \( i \), \( w_0(c) = w_i(c) \).

**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 1 we find \( w_0, \ldots, w_n = w \) 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 \( c \in \text{stab}(\varphi, X) \cap t \), \( w_{i+1}(p(c)) = w_i(p(c)) \). By iteration so does \( w_0 \) and \( w \).

\[ \Box \]

**Theorem (1 Pushing filters).** Let \( \mu(X = \kappa \cup \psi) \) be a decomposed fixpoint term, \( V \) an environment and \( \psi \) a filter condition with \( FC(\psi) \subseteq \text{stab}(\psi, X) \). Then we have \( \| \sigma_1(\mu(X = \kappa \cup \psi)) \|_V = \| \mu(X = \sigma_1(\kappa) \cup \psi) \|_V \).
Proof. Let \( w \in \llbracket \sigma(\mu(X = \kappa \cup \psi)) \rrbracket_V \). 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 the \( w_i \) on \( FC(f) \); therefore \( w_0 \) also passes the filter: we have \( \llbracket \sigma(\kappa) \rrbracket_V \). Thus \( w_0, \ldots, w_n \) is also a lineage of \( \llbracket \mu(X = \sigma(\kappa) \cup \psi) \rrbracket_V \).

Conversely, if \( w \in \llbracket \mu(X = \sigma(\kappa) \cup \psi) \rrbracket_V \), let \( w_0, \ldots, w_n \) be a lineage of \( w \); we have \( w_0 \in \llbracket \sigma(\kappa) \rrbracket_V \), thus \( w_0 \) passes the filter and by the same argument as above, so must \( w \).

\( \Box \)

**Theorem (2 Pushing anti-joins).** Let \( \mu(X = \kappa \cup \psi) \) be a decomposed fixpoint term, \( V \) an environment and \( \xi \) a term of type \( t \subseteq \text{stab}(\psi, X) \) (we suppose that \( X \) is not a free variable of \( \xi \)). Then we have \( \llbracket \mu(X = \kappa \cup \psi) \gg \xi \rrbracket_V = \llbracket \mu(X = (\kappa \gg \xi) \cup \psi) \rrbracket_V \).

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

If \( \forall R \in R, c \notin \Gamma(R) \), then we have:

- \( c \in \text{stab}(\psi, X) \)
- \( \Gamma \cup \{ X \to t \cup \{ c \} \} \vdash \psi : t \cup \{ c \} \)
- \( \llbracket \psi \gg \{ c \to v \} \rrbracket_V[X/w(v)] = \llbracket \psi \rrbracket_V[X/w] \) inductively on the size of \( \psi \) a term recursive in \( X \).

**Proof.** We will prove the result \( \llbracket \psi \rrbracket_V[X/w] = \llbracket \psi \gg \{ c \to v \} \rrbracket_V \) inductively on the size of \( \psi \) a term recursive in \( X \).

Note that when a subformula \( \xi \) of \( \psi \) is constant in \( X \) we have \( \llbracket \xi \rrbracket_V[X/w] = \llbracket \xi \rrbracket_V[X/w(v)] \) by lemma 1 and we also have that \( c \) is not in the type of this \( \xi \); since \( \forall R, c \notin \Gamma(R) \) and the definition of \( B \) assigns 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 \( \gg \) and \( \gg \) directly with sets of mappings (e.g. \( A \gg B = \{ m_A + m_B \mid m_A \in A \land m_B \in B \} \)).

- 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 \( \varphi_1 \gg \varphi_2 \), one of \( \varphi_1, \varphi_2 \) has to be constant, the other recursive. By symmetry, we suppose that \( \varphi_1 \) is recursive and \( \varphi_2 \) constant. We have \( \llbracket \varphi_1 \gg \varphi_2 \rrbracket_V[X/w(v)] = \llbracket \varphi_1 \rrbracket_V[X/i/w(v)] \gg \llbracket \varphi_2 \rrbracket_V[X/i/w(v)] = \llbracket \varphi_1 \gg \varphi_2 \rrbracket_V[X/i/w(v)] = \llbracket \varphi_1 \rrbracket_V[X/i/w(v)] \gg \llbracket \varphi_2 \rrbracket_V[X/i/w(v)] \).
- For \( \varphi_1 \gg \varphi_2 \) we similarly have \( \llbracket \varphi_1 \gg \varphi_2 \rrbracket_V[X/i/w(v)] = \llbracket \varphi_1 \rrbracket_V[X/i/w(v)] \gg \llbracket \varphi_2 \rrbracket_V[X/i/w(v)] \).
- For \( \varphi \gg \psi \) the result comes easily as \( c \notin \{ a, b \} \cup FC(f) \).

\( \Box \)

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

1. \( t_\varphi \subseteq \text{stab}(\psi, X) \)
2. \( \forall c \in t_\varphi \setminus t_k \), add(\( \psi, X, c \))

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

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

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

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

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

\( \Box \)

**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. \( t_1 \cap t_2 \subseteq \text{stab}(\psi_2, \psi_3, C_2) \cap \text{stab}(\psi_1, \psi_2, C_1) \)
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:

\( \llbracket \mu(X = \kappa_1 \cup \kappa_2 \cup \psi_1 \cup \psi_2) \rrbracket_V = \llbracket \mu(X = \kappa_1 \cup \psi_1) \gg \mu(X = \kappa_2 \cup \psi_2) \rrbracket_V \).

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

Now let us take a lineage \( w_0, \ldots, w_n \) of \([\mu(X = \kappa \cup \psi)]_V\). 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 \( \kappa_j \). Those \( w_i^j \) are not necessarily forming a lineage but we consider the subsequence containing \( w_i^0 \) and for each \( i > 0 \) \( w_i^j \) when \( w_i^j \in [\psi_i]_V[X/(w_{i-1}^j)] \) we have \( w_i^j = w_{i-1}^j \). The two resulting sequences are thus lineages and we have the expected theorem.

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

\[
[\text{rev}_b(\mu(X = \kappa \cup \psi))]_V = [\mu(X = \text{rev}_b(\kappa) \cup \psi)]_V
\]

Proof. This is a conclusion of lemma 4. Let \( w_0, \ldots, w_n \) be a lineage of \([\mu(X = \text{rev}(\kappa) \cup \psi)]_V\) 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 \( w \in \text{rev}_c(\mu(X = \kappa \cup \psi)) \).

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

REFERENCES FOR THE APPENDIX
