On the Optimization of Recursive Relational Queries

Louis Jachiet, Nils Gesbert, Pierre Genevès, Nabil Layaïda

On the Optimization of Recursive Relational Queries

Louis Jachiet
DI ENS, ENS, CNRS, PSL Research University &
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP*, LIG
Paris, France
*Institute of Engineering Univ. Grenoble Alpes
louis.jachiet@ens.fr

Nils Gesbert
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP*, LIG
38000 Grenoble, France
*Institute of Engineering Univ. Grenoble Alpes
nils.gesbert@grenoble-inp.fr

Pierre Genevès
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP*, LIG
38000 Grenoble, France
*Institute of Engineering Univ. Grenoble Alpes
pierre.geneves@cnrs.fr

Nabil Layaïda
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP*, LIG
38000 Grenoble, France
*Institute of Engineering Univ. Grenoble Alpes
nabil.layaida@inria.fr

ABSTRACT

Graph databases have received a lot of attention recently as they are particularly useful in many applications such as social networks or for the semantic web. Various languages have emerged to query such graph databases. At the heart of many of those query languages, there is a construction to navigate through the graph which allows some form of recursion.

The relational model has benefited from a huge body of research in the last half century and that is why many graph databases either rely on (or have adopted the techniques of) relational based query engines. Since its introduction, the relational model has seen various attempts to extend it with recursion and it is now possible to use recursion in several SQL or Datalog based database systems. The optimization of recursive queries remains, however, a challenge.

In this paper, we introduce μ-RA, a variation of the Relational Algebra that allows for the expression of relational queries with recursion. μ-RA can express unions of conjunctive regular path queries as well as certain non-regular properties. We present its syntax, semantics and the rewriting rules we specifically devised to tackle the optimization of recursive queries. A prototype evaluator implementing these rewriting rules is shown to be more efficient than previous approaches.

ACM Reference Format:
Louis Jachiet, Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2018. On the Optimization of Recursive Relational Queries. In *Proceedings of Bases de données avancées (BDA'18).* ACM, New York, NY, USA, 15 pages.

1 INTRODUCTION

The querying capability of languages to extract information can be greatly improved with the introduction of some form of recursion. In graph databases, such features are particularly useful, which is why graph languages often include constructions such as Regular Path Queries (RPQ) [12], Conjunctive RPQ and various further extensions of them such as Union of CRPQ (UCRPQ) [8, 9, 11, 22]. For instance, SPARQL recently [18] introduced Property Paths, PGQL has RPQs, and proposals for OpenCypher [15] include Path Patterns; all these constructions contain a recursive capability.

BDA'18, Octobre 2018, Bucarest
2018.

In SPARQL, for instance, Property Paths express non-trivial but crucial relations in RDF data. They correspond to recursive regular expressions particularly useful to navigate in deeply linked graph structures such as those found in social networks, life sciences and transportation networks. For example, the following SPARQL query retrieves the species of birds that can be found somewhere in Romania:

```
SELECT * {
  ?species :subTaxon+  :Bird .
  ?species :livesIn/:locatedIn+ :Romania . }
```

using property path expressions that describe possible relations between species, Bird and Romania.

Finally, recursive queries are also particularly interesting in ontology-based data integration where they can be used to query knowledge implicitly encoded with transitive relations in ontologies [5].

The efficient evaluation of queries with property paths is a key issue with the increasing amounts of RDF data available [1, 19]. However, queries with property paths are notoriously known to be much harder to evaluate than non-recursive ones [23, 27]. Even with modest amounts of data, the recent benchmarking work found in [7] 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 main difficulty of property path evaluation comes from the fact that it is not straightforward to find an appropriate plan for efficient evaluation.

Contribution. We introduce an extended relational model with a fixpoint operator that captures UCRPQ. We introduce rewrite rules for terms with fixpoint which allow generating new execution plans. We show empirically that a prototype implementation generating these new plans performs better than previous approaches.

Outline of the paper. The paper is decomposed as follows: Section 2 presents μ-RA: our variation of the relational algebra (RA) to equip it with a fixpoint construction inspired by the μ-calculus [21]. In Section 3, we show how recursive queries over graphs can be translated into μ-RA. Section 4 describes properties of μ-RA and demonstrates that our fixpoint can be rewritten (to push filters, projections and joins inside of the fixpoints) and that two (or more) fixpoints can sometimes be merged into a unique fixpoint. Section 5

explains why our approach is able to generate efficient plans that were beyond reach.

For reading purposes, we present only proof sketches of our main theorems; the full proofs are all available in the appendix available at https://louis.jachiet.com/tmp/mura.pdf

2 THE μ-EXTENDED RELATIONAL ALGEBRA

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

2.1 Data model

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

Formally, we assume the following constants:

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

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

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

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

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

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

The empty relation is considered compatible with all types.

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

2.2 Syntax of μ-RA terms

Our algebra μ-RA is mainly a variation of the relational algebra, with the addition of a fixpoint operator μ; 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 Section 2.3.

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

φ	::=		term
		X	relation variable
	$\mid$	$\lvert c \to v \rvert$	constant
	$\mid$	$\varphi_1 \cup \varphi_2$	union
	$\mid$	$\varphi_1 \bowtie \varphi_2$	join
	$\mid$	$\varphi_1 \triangleright \varphi_2$	antijoin
	$\mid$	$\sigma_{\mathfrak{f}}(\varphi)$	filtering
	$\mid$	$\beta_a^b(\varphi)$	duplication
	$\mid$	$\widetilde{\pi}_a(\varphi)$	anti-projection
	$\mid$	$\mu(X = \varphi)$	fixpoint

Figure 1: Grammar of μ-RA

Filters. The standard selection operation $\sigma_{\mathfrak{f}}$, which operates on a relation by keeping only a subset of its mappings, depends on a *filter* $\mathfrak{f}$ indicating which mappings are to be kept. This filter can be seen as a function from mappings to booleans.

To keep things focused, we do not detail here a syntax for filters, but we assume that for any filter $\mathfrak{f}$ we can compute a set $FC(\mathfrak{f})$ of column names such that the result of $\mathfrak{f}(m)$ depends only on $\{m(c) \mid c \in FC(\mathfrak{f})\}$.

Terms. The core syntax of terms is defined in Figure 1. The base terms are relation variables X and constants $\lvert c \to v \rvert$ (representing a single mapping with a singleton domain). Two relations can be combined with the classical relational operators $\cup$, $\bowtie$ and $\triangleright$. One relation can be filtered using the classical selection operation $\sigma_{\mathfrak{f}}$ where $\mathfrak{f}$ is a filter.

Less classically, the unary operation $\beta_a^b(\cdot)$ (column duplication) copies the values of column a onto column b, and the anti-projection $\widetilde{\pi}_a(\cdot)$ (or column dropping) removes column a. These operations can express the classical operations projection and renaming:

- The rename operator $\rho_a^b(\varphi)$ (renaming the column a into b) is defined as syntactic sugar for $\widetilde{\pi}_a\left(\beta_a^b(\varphi)\right)$;
- The projection operator $\pi_{p_1,\ldots,p_n}(\varphi)$ can be expressed in terms of $\widetilde{\pi}(\cdot)$ provided we know the type of φ: if φ has type $t = \{p_1, \ldots, p_n, a_1, \ldots, a_k\}$ we have $\pi_{p_1,\ldots,p_n}(\varphi)$ equivalent to $\widetilde{\pi}_{a_1}\left(\ldots \widetilde{\pi}_{a_k}(\varphi)\right)$. Our choice of anti-projection will allow us to extend the domains of subterms without changing the projections, as in $\widetilde{\pi}_a(\varphi) \bowtie \psi \to \widetilde{\pi}_a(\varphi \bowtie \psi)$ when a is not in the type of ψ.

Finally, we introduce the fixpoint term $\mu(X = \varphi)$ representing a recursive query. In this term, there are some additional restrictions on φ, which will be detailed in Section 2.5. The result R of this operation is a fixpoint in the sense that evaluating φ with X bound to R must yield R again. The restrictions we add ensure that this fixpoint exists and can be computed iteratively.

We consider μ as a variable binder, yielding the standard notions of *free* and *bound* variable occurrences:

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

As will be clear from the semantics, bound variables can be renamed, as usual, without changing the meanings of the terms.

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

Figure 2: Semantics of μ-RA

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

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

Figure 3: Typing rules for μ-RA

We can thus assume for simplicity that all bound variables are different from each other and from free variables.

2.3 Semantics

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

The semantics is defined in Figure 2, where $[\![\varphi]\!]_V$ designates the result of evaluating φ in the environment V. This result is defined recursively from the results of evaluating the subterms. The initial environment for evaluating the whole term is a database body D, but in evaluating $\mu(X = \varphi)$, the recursive calls use different environments where the recursion variable X is given a value: the notation $V[X/S]$ represents the environment V altered by mapping X to S.

2.4 Type System

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

Our type system is defined by the rules on Figure 3; it is quite straightforward. The only difficulty is to infer a type for fixpoints (last rule), since the rule does not give an explicit way to guess the value of t. However, this rule is still operational. Indeed, we can start typing φ with an unknown type for X (a type variable). During the typing, the value of this type variable can get constrained (typically, if $\cup$ is used it forces the two types to be equal). Then when we finish there are three possibilities: either t does not exist (the constraints are incompatible with each other), meaning the whole term is not typable; or t is entirely determined and we computed it; or the constraints are not enough to determine t entirely, meaning that the term is actually typable with different values for t. In the last case, it means that the result of the fixpoint term is always empty. Indeed, the type of a relation is the common domain of all its mappings; it is unique unless the relation is empty.

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

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

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

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

2.5 Restrictions on fixpoints

We need some additional syntactic restrictions to ensure that fixpoint terms make sense. For instance, the syntax as defined by our grammar allows $\mu(X = |c \to 0| \triangleright X)$. Although the semantics of Fig. 2 is always defined in the mathematical sense (here the semantics of this term would be $\{\{c \to 0\}\}$ because we do the union of all U_i), it is not in this case a fixpoint semantics: the U_i alternate between $\emptyset$ and $\{\{c \to 0\}\}$. We consider such terms ill-formed, and introduce the following restrictions to avoid them:

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

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

- positive *when for all subterms $\varphi_1 \triangleright \varphi_2$ of φ, φ_2 is constant in X;*

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

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

$$f(S) = \bigcup_{x \in S} f(\{x\}) \text{ for } S \neq \emptyset$$

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

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

From now on, we will only consider terms where all the fixpoints are **linear, positive and non mutually recursive.**

Example 1. *Let us suppose that we want to compute the transitive closure of R that maps a and b. Then we can do that with the term $\mu\left(X = R \cup \widetilde{\pi}_m\left(\rho_b^m(R) \bowtie \rho_a^m(X)\right)\right)$ of type $\{a, b\}$.*

3 EXPRESSING QUERIES OVER GRAPHS

An important use case for recursive queries is graph databases. Although our algebra is based on the relational model, it is able to model queries over graphs. Indeed, consider a directed graph where edges are labelled. We can assume that both vertices and edge labels are elements of our set of values $\mathfrak{V}$.

The graph can then be represented as a pair $(\mathcal{V}, \mathcal{E})$ with $\mathcal{V} \subset \mathfrak{V}$ and $\mathcal{E} \subset \mathcal{V} \times \mathfrak{V} \times \mathcal{V}$. This can be modelled as a relational database with two relations V and E representing these two sets, with the schema $\Gamma = \{V \to \{f\}, E \to \{f, l, t\}\}$ where f, l, t stand respectively for from, label, to (the column name f for V is arbitrary but will avoid some renamings).

As we discussed in the introduction, there are many formalisms for expressing recursive queries on graphs, most of which include a language of regular expressions to describe a set of paths (regular path queries or RPQ). As a simple example, take the following syntax for regular path expressions r:

r	::=	
	v	a single label
|	r_1/r_2	concatenation
|	$r_1\|r_2$	Alternative
|	r^{-1}	Reverse
|	$r?$	Zero or One
|	r^*	Kleene star
|	r^+	Transitive closure

These regular expressions denote sequences (words) of labels in the standard way. We want to translate them to queries such that the result of evaluating the query $(\!|r|\!)$ on our graph database is the set of all mappings $\{f \to v_1, t \to v_2\}$ such that there is a path from v_1 to v_2 in the graph whose sequence of labels matches r.

One possible translation is the following:

$$\begin{aligned}
(\!|v|\!) &= \widetilde{\pi}_l(\sigma_{l=v}(E)) \\
(\!|r_1/r_2|\!) &= \widetilde{\pi}_m\left(\rho_t^m((\!|r_1|\!)) \bowtie \rho_f^m((\!|r_2|\!))\right) \\
(\!|r_1|r_2|\!) &= (\!|r_1|\!) \cup (\!|r_2|\!) \\
(\!|r^{-1}|\!) &= \rho_m^f\left(\rho_f^t\left(\rho_t^m((\!|r|\!))\right)\right) \\
(\!|r?|\!) &= \beta_f^t(V) \cup (\!|r|\!) \\
(\!|r^+|\!) &= \mu\Big(X = (\!|r|\!) \cup \widetilde{\pi}_m\left(\rho_t^m((\!|r|\!)) \bowtie \rho_f^m(X)\right)\Big) \\
(\!|r^*|\!) &= \mu\Big(X = \beta_f^t(V) \cup \widetilde{\pi}_m\left(\rho_t^m((\!|r|\!)) \bowtie \rho_f^m(X)\right)\Big)
\end{aligned}$$

Note that such a translation produces some V symbols that might have been eliminated in another translation. For instance on $:A/:B?$ where $:A$ and $:B$ are labels, our translation will produce a V to handle the case of paths where $:B$ is not used. Another translation could decide to treat this special pattern as the equivalent pattern $:A|:A/:B$ that does not require to get the whole set V of nodes. Note that this rewriting is not always optimal as on regular expressions such as $(:A/:B?)/:C$ it chooses between $(:A|:A/:B)/:C$ while $:A/(:C|:B/:C)$ might be more efficient to compute. In our translation, we translate $r?$ and $r*$ "naively" and thus we introduce V symbols that could have been eliminated. However, we can then introduce a rewrite rule $V \bowtie \varphi \to \varphi$ for all terms φ such that the type of φ contains $\{f\}$. This way, we leave the removal of V symbols to this optimizer (this is what we actually implemented in our prototype described in section 5).

This translation of regular path expressions gives us the main brick to translate many graph query languages. With the relational operators, it is easy to extend this to conjunctive regular path queries and union of conjunctive regular path queries (UCRPQ); we can also represent SPARQL basic graph patterns composed of property paths (when interpreted in set semantics).

Note that μ-RA can also represent some queries which are not regular. For instance, let R be a binary relation between a and b, the term

$$\mu\Big(X = \rho_b^p(R) \bowtie \rho_a^p(R) \cup \widetilde{\pi}_{a'}\left(\rho_a^{a'}\left(\rho_b^{b'}(X)\right) \bowtie \rho_a^{a'}(R) \bowtie \rho_b^{b'}(R)\right)\Big)$$

computes the triples a, p, b such that there are a path from a to p of size $k > 0$ and from p to b of the same size k, which is not a regular property. This cannot be expressed in just UCRPQ, although it could be expressed in ECRPQ (extended conjunctive regular path queries [9]).

4 PROPERTIES OF THE μ-RA

4.1 Motivation

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

We first describe our four new rewrite rules informally; then in the following subsections we discuss the conditions under which these rules are valid. In Section 5, we will then see that our μ-RA

has plans reachable using our rewrite rules that other methods do not have.

Pushing filters into fixpoints. $\sigma_{\mathfrak{f}}(\mu(X = \varphi)) \rightarrow \mu\big(X = \sigma_{\mathfrak{f}}(\varphi)\big)$ (see Theorem 1). This always reduces the amount of mappings manipulated by the fixpoint. This rule is typically useful on *e.g.* RPQs such as $R^+(a, b)$ where a is a constant to force the evaluation to only compute the (a, b) where b is reachable from this a and not compute the whole R^+ before filtering the pairs with the wrong a.

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

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

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

We now discuss formally the conditions which allow these rewritings. All the proofs are in Appendix B.

4.2 Decomposed fixpoints

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

$$\begin{aligned}
rec(\varphi_1 \cup \varphi_2, X) &= rec(\varphi_1, X) \wedge rec(\varphi_2, X)\\
rec(\varphi_1 \bowtie \varphi_2, X) &= rec(\varphi_1, X) \vee rec(\varphi_2, X)\\
rec(\varphi_1 \triangleright \varphi_2, X) &= rec(\varphi_1, X)\\
rec(\sigma_{\mathfrak{f}}(\varphi), X) &= rec(\varphi, X)\\
rec(\widetilde{\pi}_a(\varphi), X) &= rec(\varphi, X)\\
rec(\beta_a^b(\varphi), X) &= rec(\varphi, X)\\
rec(\mu(Y = \varphi), X) &= \bot\\
rec(X, Y) &= X = Y\\
rec(|c \rightarrow v|, X) &= \bot
\end{aligned}$$

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

Lemma 1. *Let φ be a term.*

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

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

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

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

4.3 Image of a variable through a term

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

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

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

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

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 name to itself. Note that this definition manipulates functions with an infinite domain but the domain where they do not coincide with the identity is finite.

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

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

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

Definition 10. *Given a term φ linear and positive in a variable X, we define the* stabilizer *of X in φ as the following set of column*

names:

$$stab(\varphi, X) = \{c \in \mathfrak{C} \mid \forall p \in d(\varphi, X)\ p(c) = c\}$$

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

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

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

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

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

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

4.4 Reversing fixpoints

Our fixpoints often have a "direction". For instance, in the translation of r^+ that we presented, the term produced computes at the first iteration the pairs (f, t) such that $f \xrightarrow{r} t$ is an edge of the graph then it iteratively replaces (f, t) with (f, t') where $t \xrightarrow{r} t'$ is an edge of the graph. We could have designed the fixpoint to go the other way and replace at each step (f, t) with (f', t) such that $f' \xrightarrow{r} f$ is an edge of the graph.

The following proposition shows how to reverse the direction of fixpoints that are similar to the one that our translation produces for regular expressions of the form r^+ or r^*. It is possible to reverse fixpoints that have a much more general form as the examples of appendix C show.

Proposition 4. *Given a decomposed fixpoint of type $\{a, b\}$ of the form $\mu\left(X = \kappa \cup \widetilde{\pi}_c\left(\kappa \bowtie \rho_a^c(X)\right)\right)$, then it is equivalent to $\mu\left(X = \kappa \cup \widetilde{\pi}_c\left(\rho_a^c\left(\rho_b^c(\kappa)\right) \bowtie \rho_b^c(X)\right)\right)$.*

Example 1 followup. Building on our example, if we wanted to filter the transitive closure of R to limit b we could. If we wanted to filter on a we would first have to reverse the fixpoint using the proposition above.

4.5 Adding columns to fixpoints

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

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

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

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

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

Proof sketch. Point 1 can be proved inductively by definition of $stab$ and add. The second is a consequence of the first and for the third point, the only part that is not a consequence of $c \in stab(\psi, X)$ is to $[\![\psi \bowtie |c \to v|]\!]_{V[X/\{w\}]} \subseteq [\![\psi]\!]_{V[X/\{w(v)\}]}$ as adding c to w could prevent to mappings to be included in the result. For instance $c \in stab(\sigma_{x=c'}(X), X)$ but we do not have $[\![\sigma_{x=c'}(X) \bowtie |c \to v|]\!]_{V[X/\{w\}]} = [\![\sigma_{x=c'}(X)]\!]_{V[X/\{w(v)\}]}$ □

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

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

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

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

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

The reverse direction is proven the same manner. □

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

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

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

(3) $\forall c \in t_2 \setminus t_1$ $add(\psi_1, X, c)$

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

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

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

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

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

$$\begin{aligned} [\![\widetilde{\pi}_b(\mu(X = \kappa \cup \psi))]\!]_V &= [\![\mu(X = \widetilde{\pi}_b(\kappa) \cup \psi)]\!]_V \\ [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V &= [\![\mu\left(X = \beta_a^b(\kappa) \cup \psi\right)]\!]_V \\ [\![\mu(X = \kappa \cup \psi) \triangleright \varphi]\!]_V &= [\![\mu(X = \kappa \triangleright \varphi \cup \psi)]\!]_V \end{aligned}$$

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

5 COMPARISON WITH OTHER APPROACHES

Recursive queries have been well studied using various types of formalisms. In this section, we present the main ways of evaluating such queries and why our approach can be more efficient for the specific kind of recursive queries expressed with UCRPQ. In this section, we compare first from a theoretical then an empirical point of view the several approaches.

A first line of works gathers the various methods to equip the RA with a recursive mechanism. In that regards, [4] is one of the oldest and closest to ours. They introduce a "Least Fixpoint" operator in the RA and show that selections can be pushed into these fixpoints (similar to our theorem 1) however their method produces more complex terms and handles only some type of fixpoints (for instance the recursive relation can only appear once). Authors conjecture that it is possible to push projections (similar to first item of theorem 5) but do not provide a criterion, and they do not deal with conjunction (our theorems 3 and 4).

The formalism of [20] seems even closer to ours as they use a μ (inspired by the μ-calculus). However, they mainly consider the rewriting of a n-ary "regular" fixpoints into queries with only transitive closures. In contrast, we use unary fixpoints with more general query forms.

5.1 Theoretical comparison

5.1.1 Automata & α-extended Relational Algebra. One way to evaluate UCRPQ is to translate individual RPQ to automata and then union-join the individual results. Another approach is to use a Relational Algebra equipped with a transitive closure operator (thus less general than our μ operator). The automata-based method is clearly not optimal as the various RPQ are considered individually and therefore the constraints on one RPQ cannot be used on others RQP.

In fact, both methods can be shown to be sub-optimal on a single RPQ. Indeed, let us suppose that we are considering the regular expression $(a/b/c)^+$, the automata approach will start by computing the solution to a, then add b, then add c and recursively restart (i.e. in a computational form we have $R_{i+1} = ((R_i/a)/b)/c)$. In contrast the α-extended RA will compute once and for all $(a/b/c)$ and then compute the transitive closure (i.e. $R_{i+1} = R_i/(a/b/c)$).

The paper introducing Waveguide [26] notes that both methods force the associativity of the computation and do not test some associations such as computing $R_{i+1} = (R_i/a)/(b/c)$ which, in some cases, might be much more efficient. Their method thus mixes both methods to test more diverse plans. However they still can only consider one RPQ at a time and then join them which is sometimes suboptimal.

For instance on a query `?a :knows+ ?b, ?b :firstname :Axel`, Waveguide will not take advantage of the constraint on $?b$ and will materialize the full relation `:knows+`. Moreover on a query `?a :knows+ ?b, ?b` $: sameAs^+$ $?c$ our approach will have a single fixpoint in which the number of mappings treated is exactly the number of solutions while their approach will join the full relation `:knows+` with `:sameAs+`.

5.1.2 SQL. Since its 1999 version, the SQL standard supports recursive queries via the Common Table Expressions (CTE). Note that our proposed μ-RA is not very different from SQL equipped with recursive CTE, and therefore the best plans offered by SQL 1999 are not very different from ours, however what is lacking is the possibility for SQL to rewrite such terms. Not all vendors support CTE or recursive CTE and those who do support CTE tend to consider CTE as optimization barriers. There are exceptions (such as DB2) but these vendors use a technique inspired by the Magic Set technique invented for Datalog, that we now review.

5.1.3 Datalog. Datalog is a query language supporting recursive. queries. Datalog can make use of the "Magic Sets" algorithm to rewrite queries and limit the size of results to sub queries by 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 `axel:me :knows+ ?a` then the magic set method detects that, on the recursive use of `:knows`, the left side is bounded and it will not compute the full `:knows+` relation. However on an example corresponding to the translation of, e.g., `?a :knows+ ?b, ?b :presidentOf ?c` the magic set technique cannot be applied as all terms are free (even if we know that they are very few presidents).

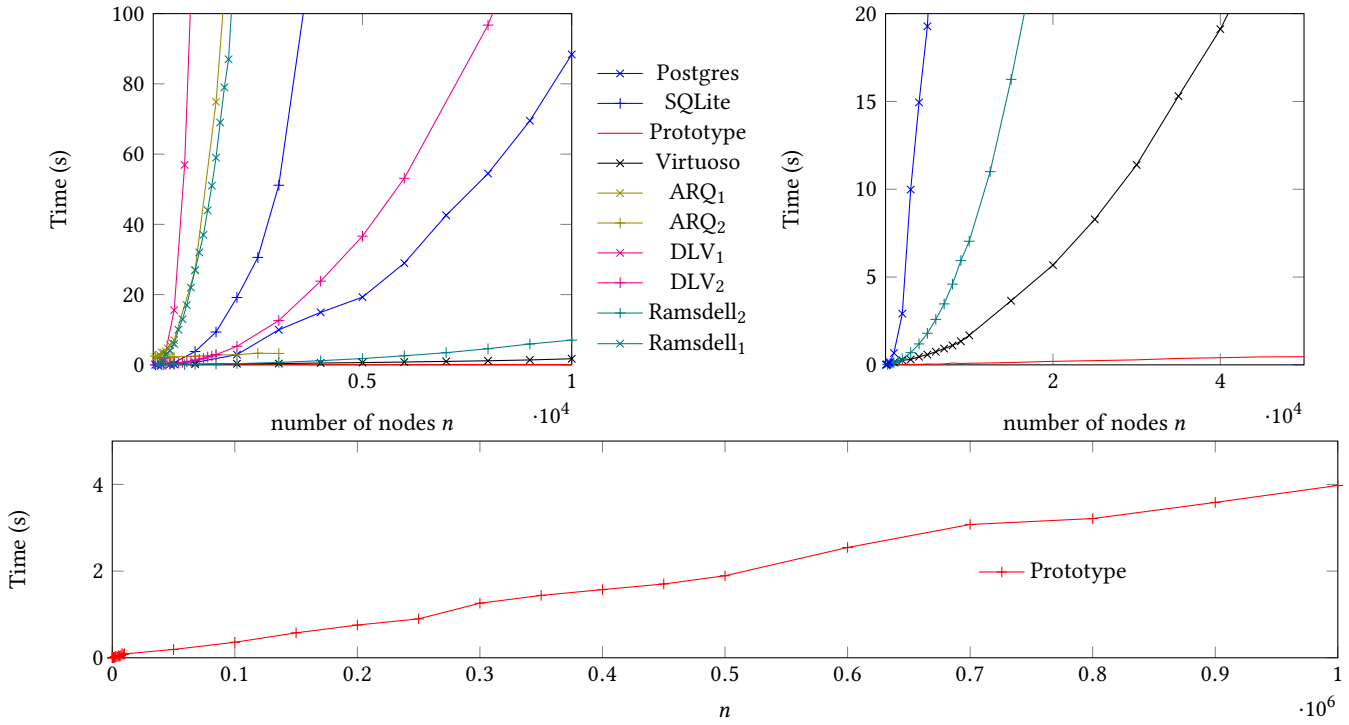

Figure 4: Evaluation time for the PP `axel:me (foaf:knows)*` ?x on a linear graph of size n.

5.2 A first benchmark

5.2.1 Competitors. We implemented a prototype based on the μ-RA and tested it against 6 other query engines:

Postgres. a popular open-source relational database. The data was encoded into one table for the relation `foaf:knows` with two indexes (on its left side and its right side).

SQLite. a popular "lightweight" open-source relational database. The same encoding as Postgres was used.

Openlink Virtuoso. an open-source Object-relational database management system capable of native SPARQL querying. We use the native SPARQL capabilities of virtuoso.

ARQ. the query engine of Apache Jena. We use its native SPARQL capabilities to encode the query.

DLV. a datalog engine[2]. The translation of the SPARQL query to datalog was done manually.

Ramsdell. a datalog engine[3]. The name of the tool is simply "datalog" but for the sake of clarity we will use the name of its author: Ramsdell.

Not included. We did not include *Neo4j* in our comparative test as its semantics is very different from the semantics of UCRPQ. We did not include Waveguide as it is not publicly available and the authors never responded.

[2]http://www.dlvsystem.com/dlv/
[3]http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html

5.2.2 Setup. For all query engines, we tried to manually reverse the direction of the fixpoint. For virtuoso, Postgres and SQLite, it did not seem to have an effect. For ARQ, DLV and Ramsdell the two directions had not the same behavior and we represent these difference by using two curves for each of these engines: we therefore have ARQ_1 and ARQ_2, $Ramsdell_1$ and $Ramsdell_2$, DLV_1 and DLV_2.

The graph used was a simple chain of n nodes where the transitive closure has $O(n^2)$ elements.

5.2.3 Results. Figure 4 shows the running times for the benchmark query FOAF-1 of [6] (it corresponds to our example 3).

One can experimentally observe that the other tested engines (even on the datalog solvers) seem to perform quadratically in practice. Possibly at the exception of ARQ_2 which fails at 3000 nodes. This result is quite surprising since the magic set technique is applied and one could expect datalog solver to be linear when the translation goes in the right way.

The more suitable plans naturally generated by our approach translate into practical performance gains: on this query, our graph takes more than a second only on graph with more than a million nodes.

5.3 Pushing the benchmark further

In order to get a more precise picture of how much our method improves on existing method we devised a benchmark to compare the tools performing the best in our first benchmark (Postgres, Ramsdell) with our prototype based on the μ-RA.

Name	Query			Number of solutions for n =						
				1000	2000	5000	10000	20000	40000	80000
Q1	$?a$	$(P1+)/P5$	$?b$	7968	16681	39611	89440	168129	381985	561526
Q2	$?a$	$(P1+)/(P5+)$	$?b$	7968	16681	39611	89440	168129	381985	561526
Q3	$?a$	$(P1+)/P2$	$?b$							
	$?b$	$P3+$	$?c$	1666431	9520309	?	?	?	?	?
Q4	$?a$	$(P4\|P5)+$	$?b$							
	$?b$	$P3+$	$?c$	2992	6280	10387	27376	52613	101599	203617
Q5	$?a$	$P2+$	$?b$							
	$?a$	$P4+$	$?c$							
	$?a$	$P5$	$N42$	0	0	0	0	0	0	0
Q6	$?a$	$P1+/P2$	$?b$							
	$N42$	$P3+$	$?b$	9959	0	3299	70269	0	1	0
Q7	$N42$	$P1/(P2+)$	$?a$	0	795	1461	0	6304	12870	3

Figure 5: Our seven queries: Pi are labels, Ni are nodes

Our prototype								
n	1000	2500	5000	10000	20000	40000	80000	100000
$Q1$	84	157	298	591	1404	3172	4995	8481
$Q2$	110	204	392	762	1711	4059	6426	10752
$Q3$	23106	139842	⊥	⊥	⊥	⊥	⊥	⊥
$Q4$	52	105	210	376	771	1544	3259	4517
$Q5$	54	53	60	71	105	174	352	506
$Q6$	2739	134	158	2651	322	614	1276	5433
$Q7$	16	28	24	37	121	265	429	588

Ramsdell datalog								
n	1000	2500	5000	10000	20000	40000	80000	100000
$Q1$	19437	138628	⊥	⊥	⊥	⊥	⊥	⊥
$Q2$	19901	135643	⊥	⊥	⊥	⊥	⊥	⊥
$Q3$	56579	⊥	⊥	⊥	⊥	⊥	⊥	⊥
$Q4$	4636	31639	148052	⊥	⊥	⊥	⊥	⊥
$Q5$	9813	82840	⊥	⊥	⊥	⊥	⊥	⊥
$Q6$	28017	⊥	⊥	⊥	⊥	⊥	⊥	⊥
$Q7$	66	131	228	453	1331	2397	6704	9382

Postgres								
n	1000	2500	5000	10000	20000	40000	80000	100000
$Q1$	386	2548	14256	60939	⊥	⊥	⊥	⊥
$Q2$	387	2486	15142	52866	⊥	⊥	⊥	⊥
$Q3$	3984	31430	100527	⊥	⊥	⊥	⊥	⊥
$Q4$	23	28	63	97	218	410	815	1805
$Q5$	15	13	16	17	17	15	22	26
$Q6$	18	22	13352	66128	68	⊥	242	⊥
$Q7$	51	607	1498	11514	52626	⊥	⊥	⊥

Figure 6: Time in milliseconds to evaluate queries in each query engine, ⊥ indicates a timeout (>150 000 ms = 2 min 30)

Virtuoso. While Virtuoso was the top performing in our first benchmark, we did not select it as it is limited to very specific types of recursive queries (in our benchmarks only 2 of the 7 queries were accepted by virtuoso.

Queries. We wrote 7 diverse queries containing one or two fixpoints. These queries are presented in figure 5.

Graphs. We also generated randomly 8 graphs with varying number of nodes $n \in \{1000, 2500, 5000, 10000, 20000, 40000, 80000, 100000\}$. In these graphs there were 5 predicates $P1$ to $P5$. The edges were generated randomly such that the label Pi corresponds to $2n(1 - i/5) + 20$ edges. This allows for $P1+$ to be very large (roughly n^2) while $P5$ is always very selective (with 20 edges).

Setup. For each of those 8 graphs and each of those 7 queries we measured the query time and thus does not include the time to load the data (which is only meaningful for Postgres). These times include the time to optimize the query but the optimization process seems to take less than a millisecond for all query evaluators. The results are shown in figure 5 (time are in milliseconds).

5.3.1 Results.

Q1 & Q2. As we can guess from the columns "number of solutions" of figure 5, Q1 and Q2 have the same solutions. This is not necessary (the queries are not equivalent) but it is due to the fact that $P1+$ captures almost all pairs of nodes while $P5/P5$ is very often empty ($P5$ corresponds to 20 triples). Postgres have the same behaviour on both queries: by looking at the plan the Postgres optimizer selected, we see that it computes $P1+$ and $P5+$ or $P5$ separately and then joins the results. It is thus not very surprising that both queries take roughly the same time since the computation of $P5+$ is fast.

On the opposite, our prototype is slower on $Q2$. The query plan for $Q2$ selected by our prototype is a merged fixpoint starting from $P1/P5$ and appending/preppending $P1$ and $P5$. For $Q1$, our prototype starts from $P1/P5$ and just tries to preppend $P1$ which takes less time, hence the time difference between $Q1$ and $Q2$. However, our prototype still is the *fatest* on both queries as it does not try to materialize the full $P1+$.

Q3 & Q4 & Q5. On these queries, our prototype starts from a set of valid triples $?a, ?b, ?c$ then discovers new solutions. These plans are optimal in the sense that all partial solutions are solutions of the query but our prototype is not the fastest.

Postgres is the *fastest* on these queries. Postgres evaluates those queries by computing all the transitive closures and then joining them. Postgres is very efficient to retrieve data and join huge quantities of partial results. It thus gains on our prototype as long as the individual transitive closures are not much larger than the final result (which is what happens on $Q1$ and $Q2$ with $P1+$).

Q6 & Q7. Both queries include a constant node ($N42$) from which our prototype will start building solutions. It is therefore not surprising that our prototype outperforms the two other systems. $Q7$ to the only query where the datalog engine outperformed Postgres.

Synthesis. In all queries, our system performs well, beating the datalog engine on all the graphs for all the queries. Furthermore, in the majority of the queries we considered, our system outperformed Postgres by several order of magnitude. For the remaining queries, where our system is outperformed by Postgres, the difference is less important and can be imputed (in a large part) to the sheer performance of the plan execution. Furthermore, in Postgres, we stored triples as integers with indexes on both sides of edges while our prototype stores triples as text in plain text files.

Sources to run the benchmark can be found on https://gitlab.inria.fr/tyrex-public/mu-RA.

6 CONCLUSION

In this paper, we considered a relational algebra μ-RA equipped with a fixpoint operator that allowed us to consider new query execution plans. These plans are obtained by an extended set of rewriting rules in μ-RA.

Our model captures plans that are more efficient than the plans previously considered by existing methods. This allows querying performance to be better or at least as fast as other systems, which can be deduced theoretically and is verified experimentally.

We showed empirically that querying performance can be significantly enhanced with a prototype implementation compared to some relational engines such as Postgres, SQLite, SPARQL evaluators such as Virtuoso or Datalog Engines such as DLV.

As a future work, we plan to extend our experiments to cover more variety in graph instances and graph query workloads. Another extension we consider is to make Postgres execute directly the query plan optimized by our prototype thereby benchmarking only the optimization of the query plan (and not its execution).

REFERENCES

[1] Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch, and Richard Cyganiak. Linking open data cloud diagram, 2017.
[2] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H. Chignell. TASWEET: Optimizing Disjunctive Path Queries in Graph Databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 470–473. OpenProceedings.org, 2017.
[3] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, July 1988.
[4] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM.
[5] Faisal Alkhateeb and Jérôme Euzenat. Constrained regular expressions for answering rdf-path queries modulo RDFS. *IJWIS*, 10(1):24–50, 2014.
[6] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 629–638, New York, NY, USA, 2012. ACM.
[7] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869, 2017.
[8] Pablo Barcelo, Diego Figueira, and Leonid Libkin. Graph logics with rational relations and the generalized intersection problem. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 115–124, Washington, DC, USA, 2012. IEEE Computer Society.
[9] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31:1–31:46, December 2012.
[10] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
[11] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.
[12] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. *SIGMOD Rec.*, 16(3):323–330, December 1987.

[13] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 323–330, New York, NY, USA, 1987. ACM.

[14] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings 14th International Conference on Data Engineering*, pages 14–23, February 1998.

[15] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[16] Georges Gardarin and Christophe de Maindreville. Evaluation of Database Recursive Logic Programs As Recurrent Function Series. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 177–186, New York, NY, USA, 1986. ACM.

[17] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Sparqling Kleene: Fast Property Paths in RDF-3x. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 14:1–14:7, New York, NY, USA, 2013. ACM.

[18] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language, W3C recommendation, March 2013.

[19] Olaf Hartig and Giuseppe Pirrò. SPARQL with property paths on the web. *Semantic Web*, 8(6):773–795, 2017.

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

[21] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[22] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, March 2016.

[23] Van-Quyet Nguyen and Kyungbaek Kim. Estimating the evaluation cost of regular path queries on large graphs. In *Proceedings of the Eighth International Symposium on Information and Communication Technology*, SoICT 2017, pages 92–99, New York, NY, USA, 2017. ACM.

[24] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[25] Nikolay Yakovets, P Godfrey, and J Gryz. Evaluation of sparql property paths via recursive sql. 1087, 01 2013.

[26] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Waveguide: Evaluating sparql property path queries. In *EDBT*, pages 525–g528, 2015.

[27] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Query planning for evaluating sparql property paths. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1875–1889, New York, NY, USA, 2016. ACM.

A PROOFS FOR SECTION 2 (THE μ-EXTENDED RELATIONAL ALGEBRA)

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

$$f(S) = \bigcup_{x \in S} f(\{x\}) \text{ for } S \neq \emptyset$$

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

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

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

□

B PROOFS FOR SECTION 4 (PROPERTIES OF THE μ-RA)

Lemma (1). *Let φ be a term.*

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

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

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

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

Proof. Given a fixpoint $\mu(X = \varphi)$ we can always decompose φ into a Constant part C and a Recursive part R (possibly empty).
The idea is to prove by induction on φ that it is true for φ where X is linear, positive and non-mutually recursive:

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

□

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

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

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

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

□

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

Proof. Let $w \in [\![\mu(X = \varphi)]\!]_V$ and let n minimal such that $w \in U_n$ (as defined by the semantic). By iterating proposition 2 we $w_0, \ldots, w_n = w$ as expected.
Conversely if we have such $w_0, \ldots, w_n = w$ then clearly $w \in [\![\mu(X = \varphi)]\!]_V$.
Now, by Lemma 2, for each $0 \leq i \leq n-1$, the mappings w_i and w_{i+1} there is $p \in d(\varphi, X)$ such that for all $c \in stab(\varphi, X) \cap t$, $w_{i+1} = w_i(p(c) = w_i(c)$. By iteration so does w_0 and w. □

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

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

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

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

Proposition (4). *Given a decomposed fixpoint of type $\{a, b\}$ of the form $\mu\big(X = \kappa \cup \widetilde{\pi}_c\left(\kappa \bowtie \rho_a^c(X)\right)\big)$, then it is equivalent to $\mu\Big(X = \kappa \cup \widetilde{\pi}_c\left(\rho_a^c\left(\rho_b^c(\kappa)\right) \bowtie \rho_b^c(X)\right)\Big)$.*

Proof. See Appendix C. □

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

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

Proof. We will prove the result $[\![\psi]\!]_{w(v)} = [\![\psi \bowtie |c \to v|]\!]_w$ inductively on the size of ψ a term recursive in X.
Note that when a subformula ξ of ψ is constant in X we have that $[\![\xi]\!]_{V[X/\{w\}]} = [\![\xi]\!]_{V[X/\{w(v)\}]}$ by lemma 1 and we also have that c is not in the type of this ξ (since $\forall R, c \notin \Gamma R$ and the definition of *add* forbids to duplicate a column onto c). Note also that subformula that are fixpoints or constants are necessarily constant in X.
Let us explore the various cases. For the simplicity of proofs, we use $\bowtie$ and $\triangleright$ directly with sets of mappings (e.g. $A \bowtie B = \{m_A + m_B \mid m_A \in A\, m_B \in B \wedge m_A \sim m_B\}$).

- For the formula X, the result is trivial and it is the only base case (constants and other variables cannot be recursive in X).

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

□

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

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

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

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

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

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

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

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

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

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

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

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

□

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

$$\begin{aligned} [\![\widetilde{\pi}_b(\mu(X = \kappa \cup \psi))]\!]_V &= [\![\mu(X = \widetilde{\pi}_b(\kappa) \cup \psi)]\!]_V \\ [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V &= [\![\mu\left(X = \beta_a^b(\kappa) \cup \psi\right)]\!]_V \\ [\![\mu(X = \kappa \cup \psi) \triangleright \varphi]\!]_V &= [\![\mu(X = \kappa \triangleright \varphi \cup \psi)]\!]_V \end{aligned}$$

Proof.
- This is a conclusion of lemma 4. Let $w_0, \dots, w_n$ be a lineage of $[\![\mu(X = \widetilde{\pi}_c(\kappa) \cup \psi)]\!]_V$ there exists v such that $w_0(v) \in [\![\kappa]\!]_{V[X/\emptyset]}$ and if we have $w_i(v)$ we can find $w_{i+1}(v) \in [\![\psi]\!]_{V[X/\{w_i(v)\}]}$ by lemma 4. In the end we have a lineage for $w(v) \in [\![\mu(X = \kappa \cup \psi)]\!]_V$ and which means $w \in \widetilde{\pi}_c(\mu(X = \kappa \cup \psi))$.
  Notice that lemma 4 gives an equality therefore this is a bijection between lineage and also proves the converse way.
- Let $w \in [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V$ we have $w_0, \dots, w_n$ a lineage of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ such that $w = w_n$ except $w(b) = w(a)$. Let $w_i' = w_i + \{b \to w_0(a)\}$ we have that $w_0', \dots, w_n'$ is a lineage of $[\![\mu\left(X = \beta_a^b(\kappa) \cup \psi\right)]\!]_V$ with lemma 4 but $w_i'(a) = w_0'(a) = w_n'(a) = w_n'(b)$ therefore $w = w_n' \in [\![\mu\left(X = \beta_a^b(\kappa) \cup \psi\right)]\!]_V$.
  Conversely, let $w \in [\![\mu\left(X = \beta_a^b(\kappa) \cup \psi\right)]\!]_V$ we have $w_0, \dots, w_n$ a lineage of $[\![\mu\left(X = \beta_a^b(\kappa) \cup \psi\right)]\!]_V$ such that $w = w_n$. Let $w_i' = w_i$ but where we removed the column b we have that $w_0', \dots, w_n'$ is a lineage of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ with lemma 4 and that $w_{i+1}'(b) = w_i'(b) = w_0'(b) = w_n'(b)$ therefore $w \in [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V$.

- The argument is the same as the one of theorem 3: let $w_0, \ldots, w_n$ be a lineage of $[\![\mu(X = \kappa \cup \psi)]\!]_V$, then $u \in [\![\varphi]\!]_V$ is compatible with w_n if and only if w_0 is which $w_n \in [\![\mu(X = \kappa \cup \psi) \triangleright \varphi]\!]_V$ if and only if $w_0 \in [\![\kappa \triangleright \varphi]\!]_V$ and thus if and only if $w_n \in [\![\mu(X = \kappa \triangleright \varphi \cup \psi)]\!]_V$. □

C REVERSING FIXPOINTS

C.1 Motivation

As we have seen, pushing filters, selections and joins into a fixpoint $\mu(X = \varphi)$ depends on φ and the derivations of φ.

C.2 Simple case

Let us consider a fixpoint of the form $\mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_a^c(\varphi) \bowtie \rho_b^c(X)\right)\right)$ of type $\{a, b\}$ and that a, b and c are all distinct. φ represents a binary relation from a to b and the fixpoint computes its transitive closure. Therefore the role of a and b are symmetrical and this fixpoint computes the same relation as $\mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_b^c(\varphi) \bowtie \rho_a^c(X)\right)\right)$.

C.3 Simple case extended to n columns

Let us now consider a fixpoint of the form $\mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\ldots \widetilde{\pi}_{c_n}\left(\rho_{a_1}^{c_1}(\ldots \rho_{a_n}^{c_n}(\varphi)) \bowtie \rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(X)\right)\right)\right)\right)$ and let us suppose that the type is $\{a_1, \ldots, a_n, b_1, \ldots, b_n\}$ with the (a_i), (b_i) and (c_1) all distinct. φ represents a binary relation from the n-uplet $a_1, \ldots, a_n$ to the n-uplet $b_1, \ldots, b_n$ and the fixpoint computes its transitive closure. Similarly as in the simple case the role of (a_i) and (b_i) are symmetrical and this fixpoint computes the same set as
$\mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\ldots \widetilde{\pi}_{c_n}\left(\rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(\varphi)\right) \bowtie \rho_{a_1}^{c_1}(\ldots \rho_{a_n}^{c_n}(X))\right)\right)\right)$.

C.4 Via an unfolding of a fixpoint over one column

Given a fixpoint of the form $\mu\left(X = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(X) \bowtie \varphi\right)\right)$ with type $\{a, b\}$, and type of Γ is $\{a, d\}$ and a, b, d all different.

This fixpoint actually computes the reflexive transitive closure of the relation between a and b induced by φ and then joins the relation between a and b induced by ψ which gives an overall relation between a and d.

From a relation point of view, we can replace the reflexive transitive closure with the union of the transitive closure plus the identity relation. With a fresh c (i.e. $c \notin \{a, b, d\}$) we have:
$\mu\left(X = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(X) \bowtie \varphi\right)\right) = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(\psi) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_a^c(X) \bowtie \rho_b^c(\varphi)\right)\right)\right)$
or using the reversion presented earlier:
$\mu\left(X = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(X) \bowtie \varphi\right)\right) = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(\psi) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_b^c(X) \bowtie \rho_a^c(\varphi)\right)\right)\right)$

C.5 Via the unfolding of a fixpoint over several columns

Just as before but a is $a_1, \ldots, a_n$, b is $b_1, \ldots, b_n$ and d is $d_1, \ldots, d_k$ (note that we do not need to have $k = n$). Given a fixpoint of the form $\mu\left(X = \psi \cup \widetilde{\pi}_{b_1}\left(\ldots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\right)$ with $P(\varphi, \Gamma) = C(\varphi, \Gamma) = \{a_1, \ldots, c_n, b_1, \ldots, b_n\}$, $P(\psi, \Gamma) = C(\psi, \Gamma) = \{a_1, \ldots, a_n, d_1, \ldots, d_k\}$ and a_i, b_j, d_l all different.

We have $\mu\left(X = \psi \cup \widetilde{\pi}_{b_1}\left(\ldots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\right) =$

$$\psi \cup \widetilde{\pi}_{b_1}\left(\ldots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(\psi)\right) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\ldots \widetilde{\pi}_{c_n}\left(\rho_{a_1}^{c_1}(\ldots \rho_{a_n}^{c_n}(X)) \bowtie \rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(\varphi)\right)\right)\right)\right)\right)\right)$$

or using the reversion presented earlier:
$\mu\left(X = \psi \cup \widetilde{\pi}_{b_1}\left(\ldots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\right) =$

$$\psi \cup \widetilde{\pi}_{b_1}\left(\ldots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\ldots \rho_{a_n}^{b_n}(\psi)\right) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\ldots \widetilde{\pi}_{c_n}\left(\rho_{b_1}^{c_1}\left(\ldots \rho_{b_n}^{c_n}(X)\right) \bowtie \rho_{a_1}^{c_1}(\ldots \rho_{a_n}^{c_n}(\varphi))\right)\right)\right)\right)\right)$$