Online Appendix to Generating Plans from Proofs

Michael Benedikt, University of Oxford
and Balder ten Cate, LogicBlox and UC-Santa Cruz
and Efthymia Tsamoura, University of Oxford

Categories and Subject Descriptors: H.2.3 [**Information Systems**]: Database Management—*Languages*

General Terms: Theory; Languages

Additional Key Words and Phrases: access methods, optimization, hidden web

Relationship of first-order languages for querying with access methods

We prove results concerning the equivalence of several languages for describing queries that are implementable using access methods.

Recall that a conjunctive query Q with atoms $A_1 \dots A_n$ is *executable* relative to a schema with access patterns if there is an annotation of each atom $A_i = R_i(\vec{x}_i)$ with an access method mt_i on R such that for each variable x of Q, for the first A_i containing x, x occurs only in an output position of mt_i. A UCQ $\bigvee_i Q_i$ where Q_i is a CQ is said to be executable if each disjunct is executable.

Recall that in the body we proved that:

Every executable CQ can be converted to an SPJ-plan, where the number of access commands of the plan is equal to the number of atoms in the query. Similarly every executable UCQ can be converted to a $USPJ$-plan.

In the body we also defined a formalism for performing queries with negation and universal quantification using a set of access methods.

An FO formula is *executable for membership checks* (relative to an access schema Sch) if it is built up from equalities and the formula True using arbitrary boolean operations and the relativized-quantifier quantifications:

$$\forall \vec{y}\ [R(\vec{x}, \vec{y}) \rightarrow \varphi(\vec{x}, \vec{y}, \vec{z})]$$

$$\exists \vec{y}\ R(\vec{x}, \vec{y}) \wedge \varphi(\vec{x}, \vec{y}, \vec{z})$$

and for all quantifications above, if R is a Sch relation, then there is an access method mt such that, in $R(\vec{x}, \vec{y})$ above, all of the input positions of mt are occupied by by a free variable or constant.

An *executable FO query* consists of:

— a set $x_1 \dots x_k$ of variables
— a first order formula $\tau(x_1 \dots x_l)$ which has free variables containing $x_1 \dots x_k$ uses a distinguished relation $T_{\vec{x}}$, and whose arity matches the number of free variables in τ, with τ executable for membership checks
— an executable UCQ $\epsilon(x_1 \dots x_l)$

We refer to $x_1 \dots x_k$ as the *return variables*, ϵ as the *output envelope* and τ as the *filter formula*.

Example 0.1. Consider an access schema with relations $R_1(x, y)$ and $R_2(x, y)$ having input-free access methods mt_1 and mt_2 respectively. Consider the plan PL that first accesses R_1 via mt_1, putting the output in temporary table T_1, then accesses R_2 via mt_2

placing the output in T_2, and finally returns tuples x lying in the first position of T_1 such that $\forall y\ T_1(x,y) \to T_2(x,y)$. This can be expressed as an RA-plan:

$$\begin{aligned} T_1 &\Leftarrow \mathsf{mt}_1 \Leftarrow \emptyset \\ T_2 &\Leftarrow \mathsf{mt}_2 \Leftarrow \emptyset \\ \mathsf{Return}\ &\pi_{\#1}(T_1) - \pi_{\#1}(T_1 - T_2) \end{aligned}$$

PL can also be expressed by the executable FO query with:

— return variable x
— output envelope $R_1(x,y)$
— filter formula $T_{x,y}(x,y) \wedge \forall y\ (R_1(x,y) \to R_2(x,y))$

That is, we first get all tuples $(x,y) \in R_1$, then filter down to those for which $\forall y\ R_1(x,y) \to R_2(x,y)$ holds, and for all such (x,y) return x. Since we have input-free access to R_1 and R_2, the output envelope is an executable CQ, and the filter formula is executable for membership checks, as required. □

In the body of the paper we proved one result about the relationship of executable FO queries and RA-plans:

Every executable FO query can be converted into an RA-plan.

We will now show that, conversely, nested RA-plans can be translated into executable FO queries, and thus the same is true for RA-plans. This will imply that RA-plans, nested RA-plans, and executable FO queries have the same expressiveness.

For any nested plan PL and assigned table T in PL, we let PL_T be the plan that is identical to PL except the unique return command is Return T.

THEOREM 0.2. *For any nested plan* PL *with free temporary tables $T_1 \dots T_k$, and any assigned temporary table T of* PL *there is an executable FO query $\varphi(\vec{x})$ over* $\mathsf{Sch} \cup \{T_1 \dots T_k\}$ *with free variables x_a for each output attribute a of* PL *such that φ is equivalent to* PL_T. *That is, for all instances I for* Sch *and the free temporary tables of* PL*,*

$$I^*, \vec{o}^* \models \varphi \textit{ iff } \vec{o} \in \mathsf{PL}_T(I)$$

Similarly for PL *with a top-level* Return*, there is an executable φ that is equivalent to* PL*.*

In the statement above, I^* is the same as I "up to the distinction between positional and named notation". That is they differ only in that for each free temporary table T_i, $I^*(T_i)$ uses positional tuples (required for a first-order formula φ) while $I(T_i)$ uses named attributes (required by PL, which accesses the free tables T_i via relational algebra expressions). We will assume that the free temporary tables T_i with arity n have attributes $\#1 \dots \#n$, and that a tuple $I^*_{T_k}(v_1 \dots v_n)$ holds iff the tuple $\vec{t}$ with attribute $\#i = v_i$ is in I_{T_i}. Likewise $\vec{o}^*$ is the variable binding with variables $x_{a_1} \dots x_{a_r}$ that corresponds to the tuple $\vec{o}$ with attributes $a_1 \dots a_r$.

Below we will sometimes drop the distinction between tuples and bindings for brevity.

PROOF. In this proof we will make use of a few closure properties of executable UCQs and executable FO queries. Executable UCQs are closed under projection (adding existential quantifiers), since the newly-quantified variables must have been free, and thus would have appeared for the first time in output positions.

The formulas that are executable for membership checks are closed under boolean combinations. They are also closed under replacement of relational atoms $R(\vec{w})$ with conjunctions $\bigwedge w_i = y_i$ with y_i variables that are either free or bound by other quantifications. Furthermore, for any executable UCQ $\epsilon(\vec{x})$ and formula φ executable for membership checks, the formula $\exists x_1 \dots x_j\ \epsilon \wedge \varphi$ is equivalent to a formula executable for membership checks, by converting $\exists x_1 \dots x_j\ \epsilon$ to a sequence of relativized-quantifier quantifications. For example, if $\epsilon(x_1, x_2) = A(x_1) \wedge B(x_1, x_2)$, with A having an input-free access and B having access on the first position. Then $\exists x_1 x_2\ \epsilon(x_1, x_2, x_2) \wedge \varphi$ can be converted to

$$\exists x_1\ A(x_1) \wedge \exists x_2\ B(x_1, x_2) \wedge \varphi$$

Universal quantifications $\forall x_1 \dots x_j\ \epsilon \rightarrow \varphi$ can be transformed similarly.

Translation. We now provide an inductive translation for both statements in the theorem.

We begin with the case for access commands. We can assume that the command is of the form $T \Leftarrow_{\mathsf{OutMap}} \mathsf{mt}_{\mathsf{InMap}} \Leftarrow T'$, since an RA expression on the right can be pushed into a middleware query command. We can also assume that the attributes of T are $\#1 \dots \#n$ and OutMap maps position i of R to attribute $\#i$ of T; an additional query middleware command doing a renaming can be applied to model a non-trivial OutMap. Likewise we can assume that the attributes of T' are $\#j_1 \dots \#j_m$, where $j_1 \dots j_m$ are the input positions of mt within relation R. We produce $x_1 \dots x_n$ as the return variables, $\epsilon = T'(\vec{y}) \wedge R(\vec{y}, \vec{x})$ as the output envelope, and $\tau = T_{\vec{x} \cup \vec{y}}(\vec{x}, \vec{y})$ as the filter formula.

Consider $\mathsf{PL}^2 \cdot \mathsf{PL}^1$, PL^2 followed by PL^1. By induction we have an executable FO query specified by $x^1_1 \dots x^1_k$, $\epsilon^1(x^1_1 \dots x^1_l)$, $\tau^1(x^1_1 \dots x^1_l)$ for PL^1. For each assigned table T^2_i of PL^2 having k_i attributes, we have an executable FO query for $\mathsf{PL}^2_{T^2_i}$ specified by $x^2_1 \dots x^2_{k_i}$, $\epsilon^2_i(x^2_1 \dots x^2_{l_i})$, and $\tau^2_i(x^2_1 \dots x_{l_i})$, where variable x^2_m corresponds to the m^{th} attribute of T^2_i. τ^2_i will be filtering some table $T_{x^2_1 \dots x^2_{l_i}}$ We create an executable FO query with

— return variables $x^1_1 \dots x^1_k$;
— output envelope ϵ obtained from ϵ^1 by replacing each atom $T^2_i(w_1 \dots w_{k_i})$ with $\exists w_{k_i+1} \dots w_{l_i}\ \epsilon^2_i(\vec{w})$. Since executable UCQs are closed under projection, this can be converted to an executable UCQ.
— filter formula τ obtained from τ^1 by replacing each relativized-quantifier existential quantification

$$\exists y_{d+1} \dots y_{k_i}\ T^2_i(y_1, \dots y_d, y_{d+1} \dots y_{k_i}) \wedge \gamma$$

by

$$\exists y_{d+1} \dots y_{k_i} \dots y_{k_i+1} \dots y_{l_i}\ \epsilon^2_i(\vec{y}) \wedge \tau^2_i(\vec{y}, T_{x^2_1 \dots x^2_{l_i}} := \vec{y}) \wedge \gamma$$

where $\tau^2_i(\vec{y}, T_{x^2_1 \dots x^2_{l_i}} := \vec{y})$ is obtained from τ^2 by replacing the free variables with $\vec{y}$ and replacing every formula $T_{x^2_1 \dots x^2_{l_i}}(\vec{w})$ with $\bigwedge_i w_i = y_i$. Relativized universal quantifiers involving T^2_i are transformed similarly.

The case of a middleware query command $T := E$, is handled via the standard translation of relational algebra to relational calculus. Since the tables in E are free, and inductively we treat free tables as having input-free access, the filter formula can be an arbitrary relativized-quantifier formula, with no further restrictions on the form of quantification. Similarly, the output envelope can be an arbitrary UCQ: for example, one that returns all tuples over the active domain.

Finally we consider a subplan call $T_1 \Leftarrow \mathsf{PL}^2[T^2_{\vec{x}^0}] \Leftarrow T_0$, where PL^2 has a distinguished free table $T^2_{\vec{x}^0}$ with attributes matching those of T_0. We assume without loss of generality that those attributes are $\#1 \dots \#n$, and in positional notation we will assume that they correspond to positions 1 to n. Again by induction we have an executable FO query for PL^2, given by return variables $x_1^2 \dots x_k^2$, output envelope $\epsilon^2(x_1^2 \dots x_l^2)$, and filter formula $\tau^2(x_1^2 \dots x_l^2)$. We form an executable FO query as follows

— the return variables are $x_1^2 \dots x_k^2$,
— the output envelope $\epsilon(\vec{x}, \vec{y}) = T_0(\vec{y}) \wedge \epsilon^{2+}(\vec{x}^2, \vec{y})$, where ϵ^{2+} is constructed from ϵ^2 as follows: for each atom of the form $T^2_{\vec{x}}(\vec{w})$, remove the atom and in the remaining collection of atoms substitute each $w_{\#j}$ occurring in the atom by $y_{\#j}$. Note that this substitution is semantically equivalent to the replacement of atoms $T^2_{\vec{x}^0}(\vec{w})$ by $\bigwedge_i w_{\#i} = y_{\#i}$. Since $y_{\#i}$ is a free variable that appears first in the relation T_0, which has input-free access, this substitution does not introduce free variables in dangerous positions, and thus does not take us out of executable UCQs.
— the filter formula $\tau(\vec{x}^2, \vec{y})$ is constructed from τ^2 by replacing atoms $T^2_{\vec{x}^0}(\vec{w})$ by $\bigwedge_i w_{\#i} = y_{\#i}$. Since this does not introduce quantification, it does not break executability for membership checks.

Above we have handled the inductive cases for the first part of Theorem 0.2. We now give the inductive step for the second statement. The executable FO query for a plan PL having a top-level Return statement $\mathsf{Return}\ E$ is the same as the executable FO query corresponding to T_{fin} in the plan formed from PL by replacing the Return statement with a final query middleware command $T_{\mathit{fin}} := E$.

Correctness. We sketch the proof that this construction is correct. We show that for any instance I interpreting the Sch relations and the free temporary tables of PL, for each assigned temporary table T of PL, if $\{x_1 \dots x_k\}, \epsilon, \tau$ are the executable FO query produced for PL_T by the algorithm above, then:

— $\{x_1 \dots x_k \mid \exists x_{k+1} \dots x_l\ \epsilon\}$ contains all tuples in $\mathsf{PL}_T(I)$
— if $I_{\vec{x}}$ consists of the tuples for attributes $\vec{x}$ satisfying ϵ in I, then for any tuple $t_1 \dots t_l$ such that $I, I_{\vec{x}}, \vec{t} \models \tau$, we have $t_1 \dots t_k \in \mathsf{PL}_T(I)$.

And similarly for the executable FO query corresponding to the final output of a plan PL having a Return statement.

Consider the translation for an access command. The first item above requires us to show that $\exists \vec{y}\, T'(\vec{y}) \wedge R(\vec{y}, \vec{x})$ returns a superset of the tuples in $\mathsf{PL}_T(I)$. But this formula returns exactly the tuples in $\mathsf{PL}_T(I)$. For the second item, plugging in the definition of the filter formula, we must show that $t_1 \dots t_l \in I_{\vec{x},\vec{y}}$ implies $t_1 \dots t_k \in \mathsf{PL}_T(I)$. But this follows from the first item.

We provide the induction step for the case of concatenation. Fix an instance I. For each assigned temporary table T_i^2 in PL^2, by induction $\exists w_{k_i+1} \dots w_{l_i}\ \epsilon_i^2(\vec{w})$ returns a superset of the tuples in $\mathsf{PL}^2_{T_i^2}$ on I. Thus by induction and monotonicity of ϵ^1, the projection of the output envelope formed for the concatenation returns a superset of the tuples in $\mathsf{PL}^1_T(I)$ evaluated with T_i^2 assigned to the output of $\mathsf{PL}^2_{T_i}$.

By induction, for each T_i^2, $\tau_i^2(\vec{y}, T_{x_1^2 \dots x_{l_i}^2} := \vec{y})$ holding of $I, \vec{t}$ implies that $\vec{t} \in \mathsf{PL}^2_{T_i^2}(I)$. Using this, the correctness of τ follows from that of τ_1.

Consider the induction step for subplan calls, $T_1 \Leftarrow \mathsf{PL}^2[T^2_{\vec{x}^0}] \Leftarrow T_0$. We start with the invariant for ϵ. By induction, we know that for any $J^2_{\vec{x}^0}$ interpreting $T^2_{\vec{x}^0}$, the tuples for which $I, J^2_{\vec{x}^0}$ satisfies $\exists x_{k+1}^2 \dots x_l^2\ \epsilon^2$ are a superset of the output of PL^2 run on $I, J^2_{\vec{x}^0}$. Using this we see that $\exists \vec{y}\, T_0(\vec{y}) \wedge \exists x_{k+1}^2 \dots x_l^2\ \epsilon^{2^+}$ returns a superset of the output of the subplan call, as required.

By induction we know that for any $J^2_{\vec{x}^0}$ as above, for $I^2_{\vec{x}^2,y_1\dots y_n}$ consisting of the tuples satisfying ϵ^2 in I, and for any tuple $t^1_1 \dots t^2_l$ matching the free variables of ϵ^2. $I, J^2_{\vec{x}^0}, I^2_{\vec{x}^2}, \vec{t}^2 \models \tau^2$ implies $t^2_1 \dots t^2_k \in \mathsf{PL}^2(I, J^2_{\vec{x}^0})$. In particular, for any tuple $\vec{y}^0 = y^0_1 \dots y^0_d$ matching the arity of T_0 we know that $I, \{\vec{y}^0\}, I^2_{\vec{x}^2}, \vec{t}^2 \models \tau^2$ implies that $t^2_1 \dots t^2_k \in \mathsf{PL}^2(I, \{\vec{y}^0\})$. Thus the output of the subplan call is contained in

$$\{t^2_1 \dots t^2_k \mid \exists\, \vec{y}^0 \in I(T_0)\ I, \{\vec{y}^0\}, I^2_{\vec{x}^2}, \vec{t}^2 \models \tau^2\}$$

which is the same as

$$\{t^2_1 \dots t^2_k \mid I, I^2_{\vec{x}^2}, \vec{t}^2_1 \dots t^2_k \models \exists \vec{y}\ \exists x_{k+1} \dots x_l\ T_0(\vec{y}) \wedge \tau\}$$

where τ is the filter formula constructed for subplan calls. Thus the inductive invariant is proven.

This completes the proof of Theorem 0.2. $\square$

Putting together the conversion of executable FO queries to RA-plans and the mapping from RA-plans to executable FO queries, we obtain:

COROLLARY 0.3. *Nested RA plans, RA plans, and executable FO queries have the same expressiveness, and there are computable transformations going from each formalism to an equivalent query in the other.*

Relationship of existential languages for querying with access methods

In the body we defined the notion of $USPJ^{\neg}$-plan, in which relational algebra's difference operator could only be used in non-membership checks, a definition that is reviewed below.

We claimed that:

PROPOSITION 0.4. *Every $USPJ^{\neg}$-plan* PL *can be translated to a $USPJ^{\neg}$ query.*

We now prove this proposition, after which we will compare $USPJ^{\neg}$-plans with other ways of capturing "executable" $USPJ^{\neg}$-queries.

Some notation. We assume that table names are only set once within a plan. Given plan PL, temporary table T that occurs in PL, and an instance I for the schema Sch, we let $\mathsf{PL}(T, I)$ be the output of T when PL is run on I. Similarly, given a relational algebra expression E over temporary tables $T_1 \dots T_n$ of PL and instance I, $\mathsf{PL}(E, I)$ represents the result of E when run on $\mathsf{PL}(T_1, I) \dots \mathsf{PL}(T_n, I)$.

Recall that for an instance I and relation R in the schema of I, $I(R)$ denotes the interpretation of R in I.

Translation. Our translation is via induction on the number of commands in the plan PL. It takes as input a $USPJ^{\neg}$-plan PL, as well as a temporary table T used in PL and it produces a $USPJ^{\neg}$ query $\mathsf{ToQuery}(T, \mathsf{PL})$ over Sch. Since $\mathsf{ToQuery}(T)$ is in relational algebra, it will work over a schema with a "named version" of each relation R with arity n, which has attributes $\#1 \dots \#n$.

We aim to maintain the invariant that $\mathsf{ToQuery}$ is equivalent to PL:

On any instance I, $\mathsf{ToQuery}(T, \mathsf{PL})(I) = \mathsf{PL}(T, I)$

Applied to the final table of PL, this invariant implies the correctness of the translation.

The base case of the induction is a single access command, which we translate to an SPJ query in the obvious way. We consider the induction step for a plan PL_i whose last command is a middleware query command $T_i := E_i(T^{i-1}_1 \dots T^{i-1}_k)$ where E_i does not include the difference operator and $T_1^{i-1} \dots T_k^{i-1}$ are temporary tables occurring in the prior command PL_{i-1}. This is straightforward: we first substitute, for each $1 \leq j \leq k$, the inductively-defined $USPJ^{\neg}$ query $\mathsf{ToQuery}(T_j^{i-1}, \mathsf{PL}_I)$ for occurrences of T_j^{i-1} within E_i. We then use the fact that $USPJ^{\neg}$ queries are closed under each of the $USPJ$ operations to obtain $\mathsf{ToQuery}(T_i, \mathsf{PL}_i)$.

We turn to the induction step for a plan PL_i that consists of a plan PL_{i-1} followed by a non-membership check. Recall that a non-membership check is of the form:

$$T'_i \Leftarrow_{\mathsf{OutMap}} \mathsf{mt} \Leftarrow_{\mathsf{InMap}} \pi_{a_{j_1} \dots a_{j_m}}(T_{i-1})$$
$$T_i := T_{i-1} - (T_{i-1} \bowtie T'_i)$$

where in the first command (1) T_{i-1} is a temporary table produced by PL_{i-1} (2) mt is an access method on relation R of arity n with input positions $j_1 \dots j_m$, (3) the input mapping InMap maps attribute a_{j_i} to position j_i, (4) the attributes of the output table T'_i are a subset of the attributes of T_{i-1} (5) the output mapping OutMap maps position j_i back to a_{j_i}. Let $\mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1})$ be the query formed inductively for PL_{i-1} with temporary table T_{i-1}. We explain the construction of the query $\mathsf{ToQuery}(T_i, \mathsf{PL}_i)$ with the construction for the remaining tables being routine. We set:

$$\mathsf{ToQuery}(T_i, \mathsf{PL}_i) = \mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1}) - (\mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1}) \bowtie_\sigma R)$$

where the join condition σ identifies $\#i \in R$ with attribute $\mathsf{OutMap}(i)$ in $\mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1})$. Note that this is indeed a $USPJ^{\neg}$ query.

Correctness. We first show the following alternative characterization of the intermediate expression $T_{i-1} \bowtie T'_i$ that is being removed from T_{i-1} to get T_i in PL^i:

CLAIM 1. *On any instance I*

$$\mathsf{PL}^i(T_{i-1} \bowtie T'_i, I) = \{\vec{t} \in \mathsf{PL}^{i-1}(T_{i-1}, I) | \mathsf{OutMap}^{-1}(\pi_{atts(T'_i)}\vec{t}) \in I(R)\}$$

where OutMap^{-1} *is the inverse of* OutMap *and* $atts(T'_i)$ *are the attributes of* T'_i.

PROOF. It is clear from the definition of T'_i that if $\vec{t} \in \mathsf{PL}^i(T_{i-1} \bowtie T'_i, I)$ then it is in the expression on the right above.

On the other hand if $\vec{t}$ is in the expression on the right above, set $\vec{t'} = \pi_{atts(T'_i)}(\vec{t})$. So $\vec{t} = \vec{t} \bowtie \vec{t'}$, and since $\vec{t} \in \mathsf{PL}^{i-1}(T_{i-1}, I)$, it suffices to show $\vec{t'} \in \mathsf{PL}^i(T'_i, I)$. We have $\mathsf{OutMap}^{-1}(t') \in I(R)$ by assumption that $\vec{t}$ is in the expression on the right above. In addition, $\vec{t}$ witnesses that $\pi_{a_{j_1} \dots a_{j_m}} \vec{t'}$ is in $\pi_{a_{j_1} \dots a_{j_m}}(\mathsf{PL}^{i-1}(T_{i-1}, I))$, and thus the fact that OutMap is the inverse of InMap guarantees that $\vec{t'} \in \mathsf{PL}^i(T'_i, I)$. This completes the proof of the claim. □

We now turn to the proof of the invariant. Suppose tuple $\vec{t}$ is in $\mathsf{ToQuery}(T_i)(I)$. Then $\vec{t} \in \mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1})(I))$, by definition of T_i, and thus by induction $\vec{t} \in \mathsf{PL}^{i-1}(T_{i-1}, I)$. Suppose $\vec{t}$ were in $\mathsf{PL}^i(T_{i-1} \bowtie T'_i, I)$. Let $\vec{t'}$ be the projection of $\vec{t}$ on the attributes of T'_i. Then $\mathsf{OutMap}^{-1}(\vec{t'})$ is in $I(R)$. We conclude that $\vec{t}$ is in $\mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1})(I) \bowtie_\sigma I(R)$, a contradiction.

In the other direction, suppose that $\vec{t}$ is in $\mathsf{PL}^i(T_i, I)$. Then $\vec{t} \in \mathsf{PL}^{i-1}(T_{i-1}, I)$, and hence by induction $\vec{t} \in \mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1})(I)$. Suppose by way of contradiction that $\vec{t}$ is in $(\mathsf{ToQuery}(T_{i-1}, \mathsf{PL}_{i-1}) \bowtie_\sigma R)(I)$. Then $\mathsf{OutMap}^{-1}(t')$ defined as above is in $I(R)$. Thus by Claim 1 $\vec{t} \in \mathsf{PL}^i(T_{i-1} \bowtie T'_i, I)$ a contradiction.

This completes the proof that the translation is correct, completing the proof of Proposition 0.4.

Relationship of $USPJ^{\neg}$-plans to other formalisms. Proposition 0.4 shows that $USPJ^{\neg}$-plans can do no more than $USPJ^{\neg}$ queries. An obvious questions is whether there is a converse: can $USPJ^{\neg}$-plans capture all $USPJ^{\neg}$ queries that have plans?

We argue that $USPJ^{\neg}$-plans express all $USPJ^{\neg}$ queries that have an RA-plan. Deutsch, Ludäscher, and Nash [Deutsch et al. 2007] define an *executable union of conjunctive queries with atomic negation*, extending the notion of executable for CQs. This is a union of an existential quantification of a sequence of literals A_i of form either $R_i(\vec{x}, \vec{c})$ or $\neg R_i(\vec{x}, \vec{c})$ such that there is an annotation of each literal S_i with an access method mt_i on R_i such that for each variable x of Q, the first A_i containing x is not negated and x occurs only in an output position of mt_i. This can be seen to be equivalent to an executable FO query (as defined in this paper) in which the filter formula is existential.

Recall that in the body of the paper we presented a translation of executable UCQs into SPJ-plans. We can extend this translation to a mapping taking an executable union of conjunctive queries with negation to a $USPJ^{\neg}$-plan, by adding non-membership checks for the negated atoms. The proof of Theorem 22 in [Deutsch et al. 2007] shows that every access-determined $USPJ^{\neg}$ query can be converted to an executable union of conjunctive queries with negation. Thus in particular any $USPJ^{\neg}$ query that has an RA-plan has a $USPJ^{\neg}$-plan. Since a $USPJ^{\neg}$-plan is both access-determined (since all RA-plans are access-determined) and equivalent to a $USPJ^{\neg}$ query (by Proposition 0.4), we can conclude that every $USPJ^{\neg}$-plan is equivalent to an executable union of conjunctive queries with negation.

We summarize the results in the following theorem:

THEOREM 0.5. *Every $USPJ^{\neg}$-plan is equivalent to an executable union of conjunctive queries with negation, and vice versa. Every access-determined query that is equivalent to a $USPJ^{\neg}$ query is equivalent to a $USPJ^{\neg}$-plan.*

Correctness of the RA-plan algorithm

Recall the algorithm for producing nested RA plans from proofs. If the proof is trivial (only one configuration), the algorithm returns the identity function: $\mathsf{Return}T_{\vec{x}^j}$. The inductive cases are presented in Figure 1.

We will prove:

THEOREM 0.6. *For any conjunctive query Q that is access-determined with respect to schema* Sch*, if* PL_1 *is the result of the RA plan-generation algorithm, then* $\pi_{\mathsf{Free}Q}(\mathsf{PL}_1)$ *answers Q.*

In the correctness proofs below, a Sch *fact* is a fact where the relation is in schema Sch. An InfAcc *fact* is a fact over a relation of the form $\mathsf{InfAcc}R$.

Inductive correctness argument. The proof of correctness will consist of separate soundness and completeness claims, each of which is proven via induction on steps in the proof, as with the SPJ-plan algorithm. We will require some definitions in order to give the inductive invariant on intermediate plans produced by the algorithm. The definitions assume a chase configuration config_i over the schema $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ and an instance I for the relations in the original schema and the relations of the form $\mathsf{InfAcc}R$.

— A $\mathsf{Sch}(\mathsf{config}_i)$-tuple of I is a tuple $\vec{t}$ with attributes for each chase constant occurring in a Sch fact of config_i taking values in the domain of I, such that if a Sch fact $F(c_1 \ldots c_n)$ holds in config_i then $F(\vec{t}.c_1 \ldots \vec{t}.c_n)$ holds in I.
— An $\mathsf{InfAcc}(\mathsf{config}_i)$-tuple, of I is as above, but replacing "Sch fact" with "InfAcc fact".

— If the transition from config_i to config_{i+1} consists of firing a Sch or an InfAcc constraint, the plan $\mathsf{PL}_i(\vec{x}^i)$ is defined to be $\mathsf{PL}_{i+1}(\vec{x}^i)$. Note that $\vec{x}^{i+1} = \vec{x}^i$ in this case.

— We consider a suffix $\mathsf{config}_i \ldots \mathsf{config}_j$ where the transition from config_i to config_{i+1} is formed via a forward accessibility axiom firing via access mt exposing fact $R(\vec{c})$. We will generate the nested plan:

$$\begin{aligned} T_0^{i+1} &\Leftarrow \mathsf{mt} \Leftarrow \pi_{c_{j_1} \ldots c_{j_m}} T_{\vec{x}^i} \\ T_1^{i+1} &:= T_0^{i+1} \bowtie T_{\vec{x}^i} \\ T_{i+1} &\Leftarrow \mathsf{PL}_{i+1}[T_{\vec{x}^{i+1}}] \Leftarrow T_1^{i+1} \\ \text{Return } & \pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)} T_{i+1} \end{aligned}$$

— We now consider a suffix $\mathsf{config}_i \ldots \mathsf{config}_j$ where the transition from config_i to config_{i+1} is formed via a backward accessibility axiom firing exposing fact $\mathsf{InfAcc}R(\vec{c})$. We will generate a plan that differs from the plan in the forward case by replacing the last line by commands returning empty if T_0^{i+1} is empty, and otherwise returning:

$$\{\vec{u} \in \pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)}(T_{i+1}) \mid \exists \vec{t} \in T_{\vec{x}^i}$$
$$\vec{u} \in \bigcap_{\vec{w} \in T_1^{i+1} \, \pi_{\vec{x}^i} \vec{w} = \vec{t}} \pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)}(\{\vec{z} \in T_{i+1} \mid \pi_{\mathsf{accessible}(\mathsf{config}_{i+1})} \vec{z} = \vec{w}\})\}$$

Fig. 1. Pseudo-code for generating RA-plans for schemas with TGDs

When I is clear from context, we will omit it from the notation. We will also need the following observation:

CLAIM 2. *At any configuration of a chase sequence starting from the canonical database of Q, and applying the rules of* $\mathsf{AcSch}^{\leftrightarrow}$*, the accessible constants are exactly the constants that occur in both a* Sch *fact and an* InfAcc *fact within a configuration.*

This follows easily from the proof that entailment using $\mathsf{AcSch}^{\leftrightarrow}$ is equivalent to entailment using $\mathsf{AltAccSch}^{\leftrightarrow}$ (Proposition 4.8 in the body), which shows the $\mathsf{accessible}$ relations can be seen as "macros" capturing all values that are in both a Sch fact and an InfAcc fact.

In our correctness argument below, we will make use of the equivalence of these two interpretations of the accessible constants.

Recall that generated plans PL_i have a free temporary table $T_{\vec{x}^i}$. Given an instance I for the schema Sch and a single tuple $\vec{t}^i$ for $T_{\vec{x}^i}$, we let $\mathsf{PL}(I, \vec{t}^i)$ be the result of PL when evaluated using the Sch relations in I and $\{\vec{t}^i\}$ for $T_{\vec{x}^i}$.

We will begin our analysis with the following lemma:

LEMMA 0.7. *Let* $\mathsf{config}_1 \ldots \mathsf{config}_j$ *be a full proof witnessing that Q entails* $\mathsf{InfAcc}Q$ *w.r.t.* $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$*, and let* $\mathsf{config}_i \ldots \mathsf{config}_j$ *be a proof suffix. Let I be an instance for* $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$*, and* PL_i *be the plan generated from this suffix using the algorithm above. Let $\vec{s}_i$ be a* $\mathsf{Sch}(\mathsf{config}_i)$*-tuple of I. Then* $\mathsf{PL}_i(I, \pi_{\mathsf{accessible}(\mathsf{config}_i)} \vec{s}_i)$ *contains the tuple* $\pi_{\mathsf{accessible}(\mathsf{config}_i) \cup \mathsf{Free}(Q)}(\vec{s}_i)$.

Above $\mathsf{PL}_i(I, \vec{t})$ represents the output of PL_i on instance I for Sch, where the unique free table of PL_i, $T_{\vec{x}_i}$, is interpreted by the single tuple $\vec{t}$.

Before proving the lemma, let us show that it easily implies that for a "full proof" (recall that this is a chase sequence witnessing $Q \models \mathsf{InfAcc}Q$ *w.r.t.* $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$) the plan produced by the algorithm returns a superset of the output of Q.

Assume we have a full proof that Q entails $\mathsf{InfAcc}Q$ with configurations $\mathsf{config}_1 \ldots \mathsf{config}_j$, and let PL_1 be the result of the algorithm applied to this proof, considered as a trivial suffix of itself. Consider an output tuple $\vec{o}$ for Q on a Sch-instance I_0. We can extend I_0 to an instance I for $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ by setting the interpretation of each $\mathsf{InfAcc}R$ in I to be identical to R in I_0. Since $\vec{o}$ is in the output of Q, it is a $\mathsf{Sch}(\mathsf{config}_1)$-tuple, whose projection on $\mathsf{accessible}(\mathsf{config}_1)$ is the empty tuple $\emptyset$. Applying Lemma 0.7, we see that $\mathsf{PL}_1(I, \emptyset)$ will return $\pi_{\mathsf{Free}(Q)}(\vec{o}) = \vec{o}$.

PROOF OF LEMMA 0.7. Fix a full proof given by chase sequence $\mathsf{config}_1 \ldots \mathsf{config}_j$. We prove the statement by downward induction on i (that is, with base case $i = j$).

The base case is clear, since PL_j just returns $T_{\vec{x}^j}$.

We first consider the inductive case where config_{i+1} is formed from config_i by firing a Sch constraint $\mathsf{PL}_i(I, \vec{s}_i)$ is just $\mathsf{PL}_{i+1}(I, \vec{s}_i)$ in this case. Since I satisfies the constraints of $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$, there is an extension of $\vec{s}_i$ to a $\mathsf{Sch}(\mathsf{config}_{i+1})$-tuple $\vec{s}_{i+1}$. By induction, $\mathsf{PL}_{i+1}(I, \vec{s}_i)$ contains $\pi_{\mathsf{accessible}(\mathsf{config}_{i+1}) \cup \mathsf{Free}(Q)} \vec{s}_{i+1}$. But $\mathsf{accessible}(\mathsf{config}_i) = \mathsf{accessible}(\mathsf{config}_{i+1})$ in this case, and hence we are done.

In the case where config_{i+1} is formed by firing an $\mathsf{InfAccCopy}$ constraint, we note that $\vec{s}_i$ is also a $\mathsf{Sch}(\mathsf{config}_{i+1})$-tuple, and thus the result follows again by induction.

We next consider the case where the transition from config_i to config_{i+1} involves the firing of a backward accessibility axiom exposing fact $\mathsf{InfAcc}R(c_1 \ldots c_n)$ via an access method on relation R with input positions $j_1 \ldots j_m$. Let $\vec{a}_i = \pi_{\mathsf{accessible}(\mathsf{config}_i)} \vec{s}_i$.

We will show that for every extension $\vec{a}_{i+1}$ of $\vec{a}_i$ on the accessible constants of config_{i+1} whose projection onto $c_1 \ldots c_n$ gives a tuple $\vec{r}$ in $I(R)$ agreeing with the values of $\vec{a}_i$ on the common attributes, $\pi_{\mathsf{Free}(Q) \cup \mathsf{accessible}(\mathsf{config}_i)} \mathsf{PL}_{i+1}(I, \vec{a}_{i+1})$ returns $\pi_{\mathsf{accessible}(\mathsf{config}_i) \cup \mathsf{Free}(Q)} \vec{s}_i$. Recalling that the definition of PL_{i+1} for a backward accessibility axiom involves intersecting over all such extensions (whenever at least one such extension exists, which we will show further below) we see that this will imply the inductive step for such axioms.

It is enough to show that there is some $\mathsf{Sch}(\mathsf{config}_{i+1})$-tuple $\vec{s}_{i+1}$ such that (i) $\vec{a}_{i+1}$ is the projection of $\vec{s}_{i+1}$ to the accessible constants of config_{i+1}, (ii) $\vec{s}_{i+1}$ projects on to $\vec{s}_i$. Assuming this, we get the conclusion we want by induction.

We claim that the constants in $\mathsf{Sch}(\mathsf{config}_i) - \mathsf{accessible}(\mathsf{config}_i)$ are disjoint from the constants in $\mathsf{domain}(\vec{r}) - \mathsf{accessible}(\mathsf{config}_i)$. This follows because, by Claim 2 mentioned above, the accessible constants are exactly those that occur in both Sch facts and InfAcc facts, and hence the constants in $\mathsf{Sch}(\mathsf{config}_i) - \mathsf{accessible}(\mathsf{config}_i)$ only appear in Sch facts of config_i, while those in $\mathsf{domain}(\vec{r}) - \mathsf{accessible}(\mathsf{config}_i)$ only appear in InfAcc facts of config_i. From the above we see that the domains of $\vec{s}_i$ and $\vec{r}$ overlap only in constants that are in $\mathsf{accessible}(\mathsf{config}_i)$, on which they are (by definition) compatible. Hence $\vec{s}_i$ and $\vec{r}$ have a join, which we denote as $\vec{s}_{i+1}$. We claim that $\vec{s}_{i+1}$ is the required tuple.

We first note that $\vec{s}_{i+1}$ is a tuple that projects on to $\vec{a}_{i+1}$. Each of the attributes of $\vec{s}_{i+1}$ is either an attribute of $\vec{r}$ or is an attribute of $\vec{s}_i$. $\vec{r}$ is a restriction of $\vec{a}_{i+1}$ by definition. For attributes in the domain of $\vec{s}_i$, if they are in the domain of $\vec{a}_{i+1}$ they must be in $\mathsf{accessible}(\mathsf{config}_i)$. For such attributes $\vec{a}_{i+1}$ is equal to $\vec{a}_i$, which is a restriction of $\vec{s}_i$, hence they must be compatible with $\vec{s}_{i+1}$. In addition, we see by definition that $\vec{s}_{i+1}$ projects on to $\vec{s}_i$. It remains to show that $\vec{s}_{i+1}$ is a $\mathsf{Sch}(\mathsf{config}_{i+1})$-tuple. Recall that $\vec{s}_{i+1}$ is a mapping from attributes corresponding to constants in $\mathsf{Sch}(\mathsf{config}_{i+1})$ to I. We need to verify that $\vec{s}_{i+1}$ preserves every Sch fact F of config_{i+1}, not just those in config_i. If F is the newly-added fact $R(c_1 \ldots c_n)$, then F will be preserved, since $\vec{s}_{i+1}$ restricted to these attributes is the same as $\vec{r}$, and by assumption $\vec{r}$ is returned by the access to

R. Otherwise we can assume F was present in the prior configuration config_i. Since F is over the schema Sch, every constant c_i mentioned in it must be in the domain of $\vec{s}_i$. Then we are done because $\vec{s}_i$ is a $\mathsf{Sch}(\mathsf{config}_i)$-tuple.

We have shown that the algorithm's behavior for this inductive case is correct, assuming there is *some* extension $\vec{a}_{i+1}$ of $\vec{a}_i$ on the accessible constants of config_{i+1} that is "consistent with the access" — that is, which has an R fact whose values match the values of $\vec{a}_i$ on the common attributes. If there is no such extension, an access on the corresponding values of $\vec{a}_i$ will return empty, and recall that the algorithm gives empty output in this case. However the tuple $\vec{s}_i$ is itself an extension consistent with $\vec{a}_i$. Hence this case cannot occur.

We now check the firing of a forward accessibility axiom exposing fact $R(c_1 \dots c_n)$ via an access method on R with input positions $j_1 \dots j_m$.

Letting $\vec{s}_i$ be as before, and $\vec{a}_i$ its projection to $\mathsf{accessible}(\mathsf{config}_i)$. We need to show that for some extension $\vec{a}_{i+1}$ of $\vec{a}_i$ that is consistent with the access to R on $c_{j_1} \dots c_{j_m}$, the nested plan PL_{i+1} returns a tuple that projects on to $\pi_{\mathsf{accessible}(\mathsf{config}_i) \cup \mathsf{Free}(Q)} \vec{s}_i$. Inductively, it suffices to show that $\vec{a}_i$ has an extension consistent with the access that is the projection of a Sch config_{i+1}-tuple to the accessible constants of config_{i+1}. Note that in this inductive case the Sch facts of config_{i+1} are the same as those of config_i. The restriction $\vec{a}_{i+1}$ of $\vec{s}_i$ to the accessible attributes of config_{i+1} can serve as the required extension. □

The corresponding "completeness" claim is as follows:

LEMMA 0.8. *Let $I, \mathsf{config}_i \dots \mathsf{config}_j, \mathsf{PL}_i$ be as in Lemma 0.7, and $\overrightarrow{\mathsf{inf}}_i$ be an $\mathsf{InfAcc}(\mathsf{config}_i)$-tuple of I. Let $\vec{a}_i$ be the projection of $\overrightarrow{\mathsf{inf}}_i$ on the accessible constants of config_i. Suppose $\mathsf{PL}_i(I, \vec{a}_i)$ returns the tuple $\vec{o}$. Then there is an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple $\overrightarrow{\mathsf{inf}}_j$ which projects onto $\vec{o}$.*

Recall that we use $\emptyset$ to denote the instance of a table with no attributes consisting only of the empty tuple. Reasoning as with Lemma 0.7, we will now derive from this claim that:

When PL_1 is the plan generated from a full proof that Q entails $\mathsf{InfAcc}Q$, then if PL_1 with parameter table set to $\emptyset$ returns a tuple $\vec{z}$ on an instance I_0 for the schema Sch, then Q evaluated on I_0 returns $\vec{z}$. That is, PL_1 returns a subset of Q.

To prove this, extend I_0 to an instance I for $\mathsf{AcSch}^{\leftrightarrow}(\mathsf{Sch})$ via copying each relation R to $\mathsf{InfAcc}R$. The initial configuration config_1 has no InfAcc facts, and thus the empty tuple is an $\mathsf{InfAcc}(\mathsf{config}_1)$-tuple of I. Applying Lemma 0.8 we conclude that for any output tuple $\vec{o}$ of PL_1 there is an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple that projects onto it. But for a full proof, the projection of an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple onto constants for the free variables of Q will be an output of the query $\mathsf{InfAcc}Q$. Since the instance I is formed by "cloning" the instance I_0, we can conclude that the output of Q contains $\vec{o}$.

PROOF OF LEMMA 0.8. We show this by downwards induction on i.

The base case is $i = j$. $\mathsf{PL}_j(I, \vec{a}_j)$ returns only $\vec{a}_j$, and $\vec{a}_j$ is the projection of an InfAcc config_j-tuple by assumption.

We consider the inductive case where we apply a Sch constraint. If $\mathsf{PL}_i(I, \vec{a}_i)$ returns $\vec{o}$, then $\mathsf{PL}_{i+1}(I, \vec{a}_i)$ returns $\vec{o}$ by definition. Then by induction, there is an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple $\overrightarrow{\mathsf{inf}}_j$ that projects onto $\vec{o}$, as required.

In the case where we apply an InfAcc constraint, we extend $\overrightarrow{\mathsf{inf}}_i$ to an InfAcc config_{i+1} tuple $\overrightarrow{\mathsf{inf}}_{i+1}$. By induction, there is an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple $\overrightarrow{\mathsf{inf}}_j$ that projects onto the output $\vec{o}$ of $\mathsf{PL}_{i+1}(I, \vec{a}_i)$ and since $\mathsf{PL}_i = \mathsf{PL}_{i+1}$ in this case, we are done.

Next, consider the case where the next step is the firing of a backward accessibility axiom exposing fact $\mathsf{InfAcc}R(c_1 \ldots c_n)$ via an access method mt on R with input positions $j_1 \ldots j_m$. We know $\mathsf{InfAcc}R(c_1 \ldots c_n)$ must hold in config_i. Let $\overrightarrow{\mathsf{inf}}_{\mathsf{Inp}}$ be the restriction of $\overrightarrow{\mathsf{inf}}_i$ to the constants appearing in $\mathsf{InfAcc}R(c_1 \ldots c_n)$ within input positions of mt, and let $\vec{a}_{i+1}$ be the restriction of $\overrightarrow{\mathsf{inf}}_i$ to the accessible constants of config_{i+1}. Note that $\vec{a}_{i+1}$ extends $\vec{a}_i$. Since $\mathsf{PL}_i(I, \vec{a}_i)$ returns $\vec{o}$, we know that for every tuple $\vec{u}$ in I that extends $\vec{a}_i$ by joining on a tuple of R that extends $\overrightarrow{\mathsf{inf}}_{\mathsf{Inp}}$, $\mathsf{PL}_{i+1}(I, \vec{u})$ returns a tuple that projects onto $\vec{o}$. We can see that $\vec{a}_{i+1}$ is such a tuple, and thus $\mathsf{PL}_{i+1}(I, \vec{a}_{i+1})$ returns a tuple $\vec{o}_{i+1}$ that projects onto $\vec{o}$. By induction, there is an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple $\overrightarrow{\mathsf{inf}}_j$ that projects onto $\vec{o}_{i+1}$. But then $\overrightarrow{\mathsf{inf}}_j$ projects onto $\vec{o}$.

Finally, consider the case where the next step is the firing of a forward accessibility axiom exposing fact $R(c_1 \ldots c_n)$ via an access method on R with input positions $j_1 \ldots j_m$. Then if $\mathsf{PL}_i(I, \vec{a}_i)$ returns $\vec{o}$ we know that for some extension $\vec{a}_{i+1}$ of $\vec{a}_i$ compatible with the access, $\mathsf{PL}_{i+1}(I, \vec{a}_{i+1})$ returns some tuple $\vec{o}_{i+1}$ that projects onto $\vec{o}$. We claim that $\vec{a}_{i+1}$ must be the projection of an $\mathsf{InfAcc}(\mathsf{config}_{i+1})$-tuple. Assuming this, we are done, since by induction there would be an $\mathsf{InfAcc}(\mathsf{config}_j)$-tuple $\overrightarrow{\mathsf{inf}}_j$ that projects onto $\vec{o}_{i+1}$, and since $\vec{o}_{i+1}$ projects onto $\vec{o}$, $\overrightarrow{\mathsf{inf}}_j$ would also project onto $\vec{o}$.

The $\mathsf{InfAcc}(\mathsf{config}_i)$-tuple $\overrightarrow{\mathsf{inf}}_i$ and $\vec{a}_{i+1}$ are consistent on their common domain, and hence have a join $\overrightarrow{\mathsf{inf}}_{i+1}$. We claim $\overrightarrow{\mathsf{inf}}_{i+1}$ is the required $\mathsf{InfAcc}(\mathsf{config}_{i+1})$-tuple. Clearly $\overrightarrow{\mathsf{inf}}_{i+1}$ projects onto $\vec{a}_{i+1}$, so we need only show that $\overrightarrow{\mathsf{inf}}_{i+1}$ is an $\mathsf{InfAcc}(\mathsf{config}_{i+1})$-tuple. The newly-added fact $\mathsf{InfAcc}R(c_1 \ldots c_n)$ is preserved since the access producing $\vec{a}_{i+1}$ is compatible, while the config_i facts are preserved via the assumption on $\overrightarrow{\mathsf{inf}}_i$. □

Applying the two lemmas to a full proof (as described below each lemma), we have shown that whenever there is a full proof using $\mathsf{AcSch}^{\leftrightarrow}$, there is a corresponding plan that returns exactly the same output as Q, completing the proof of Theorem 0.6.

Proof-based Plans Runtime Dominate Arbitrary Plans

Recall that an SPJ-plan PL *uses no more runtime accesses than* SPJ-plan PL', denoted $\mathsf{PL} \preceq_{\mathsf{RTA}} \mathsf{PL}'$, if for every pair consisting of a method mt and method input $\vec{t}$ that is executed in running PL on instance I of the schema, the same pair is also executed in running PL' on I.

We recall the theorem stated in the body of the paper:

For conjunctive query Q and access schema with TGD constraints Σ, for every SPJ-plan PL that answers Q, there is a chase sequence v proving $\mathsf{InfAcc}Q$, such that, letting PL^v be the SPJ-plan PL^v generated from v via the proof-to-plan algorithm, $\mathsf{PL}^v \preceq_{\mathsf{RTA}} \mathsf{PL}$.

We now give the proof. It follows along the lines of the corresponding argument for $\preceq_{\mathsf{Meth}}$, but since it deals with arbitrary plans, rather than "left-deep" plans, it requires a more involved analysis.

Normal form for plans. We start by showing that the shape of plans can be restricted. First, we can assume that a plan performs no middleware query commands: all of the other middleware query commands can be pushed into the expressions that are inputs to the access commands or the Return command. This "inlining" can blow up the size of the expressions, since the use of middleware queries gives the ability to iteratively create views, which may require exponential space to flatten back to a conjunctive query. But it will not impact the accesses performed, which is all that matters for this theorem. We can also assume that every temporary table is the output table of at most one access command. This can be achieved by renaming tables. We refer to these as *normalized plans* below.

Construction. Let $\mathsf{PL} = \mathsf{Command}_1 \ldots \mathsf{Command}_j$ be any normalized plan equivalent to conjunctive query Q. We consider running PL on the configuration config_∞ — any database (possibly infinite) generated from the canonical database of Q by repeatedly firing all Sch constraints. For $i \leq j$, we let $\mathsf{PL}_i = \mathsf{Command}_1 \ldots \mathsf{Command}_i$, the i^{th} prefix of PL. Q has a match on the elements in config_∞ corresponding to free variables of Q, and since PL is equivalent to Q on instances satisfying the constraints, PL must have such a match on config_∞ as well. There is thus a finite subinstance config_0 of config_∞ on which PL returns the elements corresponding to the free variables of Q.

We now construct a chase proof v which begins with config_0, proceeds by firing accessibility axioms only, and which includes distinguished intermediate configurations $\mathsf{config}_0 \ldots \mathsf{config}_j$. Further v will be *chase-faithful to* PL, meaning (informally) that the firings of accessibility axioms in v mimic the accesses made by PL_j on config_0. Formally, we will construct the proof so that if $\mathsf{Command}_i$ resulted in an access via method mt on config_0 that returned $R(\vec{c})$, then fact $\mathsf{InfAcc}R(\vec{c})$ will be generated by an accessibility axiom corresponding to method mt associated with relation R in one of the rule firings linking config_{i-1} to config_i.

We construct the proof, and thus the corresponding proof-based plan, by induction on i. Consider inductively a prefix of PL PL_i consisting of the concatenation $\mathsf{PL}_{i-1} \cdot \mathsf{Command}_i$, where $\mathsf{Command}_i$ is an access command $R \Leftarrow \mathsf{mt} \Leftarrow E$.

By the induction hypothesis, for every tuple $R(\vec{c})$ returned by an access of PL_{i-1} on config_0, $\mathsf{InfAcc}R(\vec{c})$ was generated by an accessibility axiom in the proof leading up to config_{i-1}. Since the temporary tables produced by PL_{i-1} all use values that come from these accesses, all values returned by E on config_0 must be members of the relation $\mathsf{accessible}$ in config_{i-1}. Hence for any fact F returned by the access command $R \Leftarrow \mathsf{mt} \Leftarrow E$, we can fire an accessibility axiom in config_{i-1} to expose F. Iterating this, we arrive at an extension of the proof that exposes every such fact. This completes the induction.

We now claim that we can complete the proof to get a match for $\mathsf{InfAcc}Q$. Consider the facts generated by PL's accesses, and let config'_∞ be the result of chasing this set of facts with the Sch constraints. It is clear that config'_∞ satisfies the constraints, and its accessible part contains the accessible part of config_0. Since Q has an SPJ-plan it is access-monotonically-determined, and hence config'_∞ has a match for Q. Thus renaming the facts to use InfAcc-relations and taking the $\mathsf{InfAccCopy}$ copy of the rules fired to get a match for Q, we get an extension of the chase sequence with a set of firings of $\mathsf{InfAccCopy}$ rules that gives a match for $\mathsf{InfAcc}Q$ on config'_∞. The extended chase sequence began with the finite instance config_0, rather than with the canonical database of Q, as required of a full proof that Q entails $\mathsf{InfAcc}Q$. But we can generate config_0 by a finite sequence of firings of the Sch constraints, which we can add as a prefix to get a full proof.

Correctness of construction. Above we have constructed a valid proof v of $\mathsf{InfAcc}Q$ from Q using AcSch, and hence the corresponding proof-based plan PL^v will answer Q. Further PL^v performs no more accesses than PL in when run on the database config_0. We know that PL^v will make the same accesses on config_∞ as it does on config_0, and PL will make at least as many accesses on config_∞ as on config_0. Hence PL^v performs no more accesses than PL on config_∞. We claim that on an arbitrary instance I satisfying the constraints of Sch, accesses made by PL^v must also be made by PL.

Consider an access $(\mathsf{mt}, \mathsf{AccBind})$ made by PL^v on such an I. There is an access command such that the access is generated by $\mathsf{Command}'_i = T' \Leftarrow \mathsf{mt} \Leftarrow E'$, for method mt on relation R having input positions $j_1 \ldots j_m$, with a tuple t being returned by expression E' on I being the input to the access. Let config_i be the chase configuration at which the accessibility axiom corresponding to $\mathsf{Command}'_i$ was fired, and let $R(\vec{c}_t)$ be the fact exposed by the firing of this axiom.

From the proof of Proposition 5.4 in the body of the paper we can see that E' is a projection of a temporary table T_{i-1} storing all tuples with a homomorphism from the facts exposed by accessibility axioms in the corresponding chase configuration. Thus, there is some tuple $\vec{c}_t$ in the interpretation of R in config_∞, and a homomorphism h preserving the facts generated by an accessibility axiom fired prior to config_i which maps $\vec{c}_t$ to t. Let $\mathsf{AccFacts}(\mathsf{config}_{i-1})$ denote the set of facts generated by an accessibility axiom fired prior to config_i.

From the construction and the argument above it follows that if an access command of PL^v generated an access to method mt using some tuple $\vec{c}_t$ on config_∞, then some prefix PL_j of PL also generated an access to mt using the tuple $\vec{c}_t$ as input when run on config_∞, via executing some command $\mathsf{Command}_j$ of form $T_j \Leftarrow \mathsf{mt} \Leftarrow E_j$. Further $j \leq i$, since each access command of PL generates at least one access command in PL^v. It suffices to show that when E_j is evaluated on the temporary tables produced by PL_{j-1} on I, t is in the result. This will guarantee that the access $(\mathsf{mt}, \mathsf{AccBind})$ is performed by PL on I. The expression E_j produces $\vec{c}_t$ when evaluated on the temporary tables produced by PL_{j-1} on config_∞. Thus the tuple $\vec{c}_t$ is produced by an SPJ-expression over tuples in $\mathsf{AccFacts}(\mathsf{config}_{j-1})$. But h preserves all such facts, and hence preserves SPJ queries over the facts. Since h maps $\vec{c}_t$ to t, we can conclude that t is in the result of E_j evaluated on the temporary tables produced by PL_{j-1} on I, as required.

This completes the proof of the theorem.

Optimal cost proof-based plans for GTGDs

We will show that the search algorithm for optimal proof-based plans can be instantiated with appropriate termination conditions in the case of Guarded TGDs (GTGDs below).

For a chase sequence $v = \mathsf{config}_1 \ldots \mathsf{config}_j$ let $\mathsf{RulesOf}(v)$ be the sequence $\tau_1 \ldots \tau_{j-1}$ where for $i \leq j-1$ τ_i is the TGD fired in going from config_i to config_{i+1}. We will make use of the following fact, proved in [Lukasiewicz et al. 2012]:

For any queries Q and Q' and finite collection of GTGDs Σ, there is a number k at most doubly exponential in $|Q'|$ and $|\Sigma|$ such that: for any chase sequence v starting at the canonical database of Q and ending in a match of Q', there is another sequence v' of size at most k, starting at the canonical database of Q, ending in a configuration with a match of Q', with $\mathsf{RulesOf}(v')$ a subsequence of $\mathsf{RulesOf}(v)$.

This is shown in the proof of Lemma 4 of [Lukasiewicz et al. 2012], with the bound $|Q'| \cdot |\Sigma| \cdot (2w)^w \cdot 2^{|\Sigma| \cdot (2w)^w}$, where w is the maximal width of any relation in the schema. It is shown there that when the final configuration of a chase sequence of length longer than this is arranged in a forest reflecting the relationship of guard atoms to the generated children, then there are two nodes h and h' corresponding to guard atoms $A(\vec{c})$ and $A'(\vec{d})$ that lie an ancestor relationship of the forest, such that the path between them can be "collapsed" without losing a match of Q'. Collapsing here means having the parent of h generate h' instead of h. If we identify a set of rule firings with a sequence labelled with the rule names, this collapsing operation corresponds to taking a subsequence.

Algorithm 1: plan search

```
Input: query $Q$, schema $S$
Output: plan BestPlan
1  ProofTree := an initial node $v_0$ labelled with the configuration obtained by a sufficient
   number of firings of Sch constraints.
2  Set Candidates($v_0$) = all pairs $(R(c_1 \dots c_n), \mathsf{mt})$ with $R(c_1 \dots c_n)$ a fact in the original
   configuration and mt a method on $R$.
3  BestPlan := $\perp$
4  BestCost := $\infty$
5  while there is a non-terminal node $v \in$ ProofTree do
6      Choose such a node $v$.
7      Choose a candidate fact and method $(R(c_1 \dots c_n), \mathsf{mt}) \in \mathsf{Candidates}(v)$ with
       $\mathsf{accessible}(c_{j_1}) \dots \mathsf{accessible}(c_{j_m}) \in \mathsf{config}(v)$ and mt having inputs $j_1 \dots j_m$.
8      Add a new node $v'$ as a child of $v$ with configuration formed by adding
       InfAcc$R(c_1 \dots c_n)$ a sufficiently large closure by firings of the InfAccCopy constraints.
9      Remove $(R(c_1 \dots c_n), \mathsf{mt})$ from Candidates($v$), marking $v$ as terminal if it has no more
       candidates.
10     Determine if $v'$ is successful by checking if InfAcc$Q$ holds, and if so also mark it as
       terminal.
11     if $v'$ is successful and Cost(Plan($v'$)) < BestCost then
12         BestPlan := Plan($v'$)
13         BestCost := Cost(Plan($v'$))
14 return BestPlan;
```

Recall the high-level algorithm for finding an optimal proof-based plan from the body of the paper, shown in Figure 1.

Recall also that to instantiate this algorithm for a class of constraints we require:

— A sequence v_0 formed by closing the canonical query Q under some firings of Sch constraints. We use this set in the step of chasing with the Sch constraints in line 1.

— For each chase sequence w_0, an extension $v'(w_0)$ of w_0 by firing InfAccCopy constraints, used within every step of the while loop on line 8.

Setting k as in the fact from [Lukasiewicz et al. 2012] above, with Q' being InfAcc$Q$, we let v_0 be the result of k rounds of chase steps starting with the canonical database of Q, where in each round we fire all triggers for Sch constraints, active or not. We let $v'(w)$ be the same number of rounds of chasing of w. We now claim that the algorithm instantiated with these sufficient conditions returns the optimal proof-based plan:

THEOREM 0.9. *Consider any simple cost function* Cost*, access schema* Sch *with GTGD constraints, and conjunctive query Q. Then Algorithm 1, instantiated with the sufficient sets above and the cost function* Cost*, will always return a plan with the lowest cost among all those proof-based plans that completely answer Q w.r.t.* Sch*, or return $\perp$ if there is no plan that answers Q.*

PROOF. Consider any chase proof w that Q entails InfAcc$Q$, with PL^w the resulting plan and $\mathsf{mt}_1 \dots \mathsf{mt}_j$ the sequence of methods used in access commands. By the fact above, there is another proof v of size k such that the sequence of accessibility axioms fired in v will be a subsequence of the sequence in w. In particular, the plan for v, PL^v, will use no more methods than the plan for w, and hence the cost will be no more than that of w under a simple cost function. Further PL^v will use at most k accessibility axioms. We can move all the Sch constraints in v to the beginning, where they will clearly be embedded in k rounds of chasing with the Sch constraints. Similarly, blocks of InfAccCopy constraint firings in v can be distributed so that they are performed as soon as they are applicable. Let v^* be the resulting proof.

We argue again that in every iteration of the while loop in the algorithm, if v^* has not been discovered then the while loop will not terminate, and v^* will have a prefix in ProofTree with a non-empty set of candidates. As before, since each iteration of the while loop removes a candidate, eventually such an ancestor prefix will be chosen to be expanded. Thus the algorithm will return v^*. $\square$

REFERENCES

DEUTSCH, A., LUDÄSCHER, B., AND NASH, A. 2007. Rewriting queries using views with access patterns under integrity constraints. *Theoretical Computer Science 371,* 3, 200–226.

LUKASIEWICZ, T., CALÌ, A., AND GOTTLOB, G. 2012. A general datalog-based framework for tractable query answering over ontologies. *Journal of Web Semantics 14,* 0, 57–83.