View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Helsingin yliopiston digitaalinen arkisto

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Mathematics and Statistics

Tekijä — Författare — Author

Tuukka Wahtera

Työn nimi — Arbetets titel — Title

Introduction and Comparison of Dynamic Complexity Classes

Oppiaine — Läroämne — Subject

Mathematics

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	May 2020	39 pages

Tiivistelmä — Referat — Abstract

This thesis gives some background and an introduction on dynamic complexity theory, a subfield of descriptive complexity theory in which queries on databases are maintained dynamically upon insertions and deletions to the database. The basic definitions of the dynamic complexity framework are given along with examples of queries maintainable with dynamic queries and a comparison of different dynamic complexity classes.

Avainsanat — Nyckelord — Keywords

Dynamic Complexity Theory

Säilytyspaikka — Förvaringsställe — Where deposited

Kumpula Campus Library

Muita tietoja — Övriga uppgifter — Additional information

Introduction and Comparison of Dynamic Complexity Classes

Tuukka Wahtera

University of Helsinki
Department of Mathematics and Statistics

May 10, 2020

Contents

1 Introduction **3**
1.1 Introduction 3
1.2 Complexity Theory 3
1.3 Descriptive Complexity Theory 4

2 Dynamic Complexity Framework **6**
2.1 Background 6
2.2 Definitions 7
2.2.1 Basic Notation 7
2.2.2 Structures and First-Order Logic 8
2.2.3 The Dynamic Complexity Framework 8
2.2.4 DynFO 10
2.2.5 DynProp 11
2.2.6 DynQF 12
2.3 Reachability in Undirected Graphs in DynFO 14

3 Relating Dynamic Complexity Classes **17**
3.1 Techniques for Collapsing Dynamic Classes 19
3.2 Collapsing Dynamic Complexity Classes 21
3.3 Δ-semantics 31
3.4 Relating Dynamic Complexity Classes with Static Complexity Classes 35

4 Conclusions **38**

1 Introduction

1.1 Introduction

Traditionally descriptive complexity theory has been concerned with static queries of finite structures. This is often not appropriate for databases in practice as queries are often made multiple times with modifications in between. In the traditional static context this would lead to requiring powerful query languages with computationally expensive model-checking problems.

Dynamic complexity theory defines a system that allows us to maintain a query on a database in a dynamic setting with the help of auxiliary relations. This can often be achieved with weaker logics than what would be required in the static setting.

We will start by giving an informal introduction to complexity theory, descriptive complexity theory and how dynamic complexity ties into them. We will then continue with the formal definition of the dynamic complexity framework and common dynamic complexity classes with examples on maintaining queries with dynamic programs and a more detailed example on maintaining reachability in undirected graphs in DynFO.

The rest of the thesis is dedicated to examining the relations of different dynamic complexity classes and static complexity classes.

The thesis is structured in the following way:

Chapter 1 (this chapter) gives a short introduction to complexity theory and descriptive complexity theory. Chapter 2 introduces the dynamic complexity framework, different dynamic complexity classes that use different updating logics and shows how to maintain reachability in undirected graphs In DynFO. In Chapter 3 we compare the power of different complexity classes to each other and to static complexity classes.

1.2 Complexity Theory

Computational complexity is the field of computer science that studies how many computational steps or how much memory in relation to the size of an input instance of a problem is required to determine whether it has a given property or not.

For example, reachability is a problem where, for a given directed graph and two nodes s and t, to determine whether there exists a path from s to t. Reachability can be solved in an amount of steps that is linear to the input graph, that is, in linear time, by a breadth first search starting from the node s for n iterations, where n is the amount of nodes in the graph. The node t is reachable from s if t is

encountered in the search before n iterations are reached or we run out of nodes to check, otherwise it is not.

Another well known problem is three-colourability, given an undirected graph, determine whether its vertices can be coloured with three colours with no two connected vertices sharing a colour. It can be solved by trying all ways to assign a colouring to the vertices and testing if the assignment is valid. This takes an amount of steps that is exponential in size compared to the size of the input graph and no asymptotically faster algorithm is known.

Let $\textsc{Time}[t(n)]$ be the set of problems solvable in $O(t(n))$ computational steps, that is, problems for which an algorithm which uses at most a constant multiple of $t(n)$ steps exists. Since the underlying machine model affects the amount of steps that running a computation takes, we are often interested in larger classes of problems that are more robust to variations in the underlying machine. An important complexity class like this is polynomial time, P, the class of problems solvable in $O(n^k)$ steps for some fixed k

$$\mathrm{P} = \bigcup_{k=1}^{\infty} \textsc{Time}[n^k].$$

Other important complexity classes include nondeterministic polynomial time, NP, and exponential time, Exp. Nondeterministic polynomial time is the class of problems solvable in polynomial time on a *nondeterministic* machine. A nondeterministic machine can make arbitrary decisions during the computation and if any of the combinations of decisions lead to an accepting state, the input is accepted. Exp is the class of problems solvable in $O(2^{p(n)})$ where $p(n)$ is a polynomial function of n, or

$$\textsc{Exp} = \bigcup_{k=1}^{\infty} \textsc{Time}[2^{(n^k)}].$$

The aforementioned three-colourability problem can be solved in polynomial time by the following nondeterministic algorithm: For every node, assign arbitrarily any one of three colours, then check whether each node connected by an edge is of a different colour. If so, accept the input and otherwise reject it. The three-colourability problem is thus in NP. With a deterministic model of computation, three colourability is in Exp.

1.3 Descriptive Complexity Theory

Descriptive complexity theory is the logician's take on complexity theory. Instead of asking how much of a computational resource it takes to solve a problem, descriptive complexity theory is interested in how difficult it is to specify the answer

to it in a formal language. The underlying idea is that answers to computationally difficult problems would require more sophisticated languages or longer formulas to define which, perhaps surprisingly turns out to often be the case.

In descriptive complexity theory inputs are represented as finite logical structures. For example, a graph is a structure

$$\mathcal{A}_G = (\{1, ..., v\}, E^G)$$

with the set of vertices $(1, 2, ...v)$ as the universe and the set of edges E^G. With first-order logic we can, for example, define the graphs for which there are exactly two edges leaving every vertex:

$$\forall x \exists yz \forall w(y \neq z \wedge E(x, y) \wedge E(x, z) \wedge (E(x, w) \rightarrow w = y \vee w = z))$$

or graphs in which there exists a path with length of at most two between every node:

$$\forall xy \exists z((E(x, z) \wedge E(z, y)) \vee E(x, y) \vee x = y).$$

Unlike in first-order logic where quantifiers range over the universe, in second-order logic we can also quantify over relations over the universe. Second-order formulas are much stronger in their expressive power than first-order formulas. For example, the following formula defines the three-colourable graphs, something that is not expressible with first-order formulas.

$$\begin{aligned}&\exists R \exists Y \exists B \forall x((R(x) \vee Y(x) \vee B(x)) \wedge \\ &\quad (\forall y)(E(x, y) \rightarrow \neg(R(x) \wedge R(y)) \wedge \neg(Y(x) \wedge Y(y)) \wedge \neg(B(x) \wedge B(y))))\end{aligned}$$

This formula is also a *existential second-order formula.* Existential second-order formulas (SO∃) are second-order formulas that begin with a number of existential second-order quantifiers that are followed by a first-order formula.

To tie logic to complexity theory, we can look at the complexity of the *model-checking* problem. Given a logic L and a domain $\mathcal{D}$ of finite structures the model-checking problem for L asks, given a structure $\mathcal{S} \in \mathcal{D}$ and a formula $\psi \in L$, is it the case that $\mathcal{U} \vDash \psi$? When both the structure and formula are fixed, the computational complexity of the model-checking problem is called the *combined complexity* of the model-checking problem. When the formula is fixed and we want to find the class of models that satisfy it, we get the *data complexity* for L and $\mathcal{D}$. When the

We say that a logic L *captures* a complexity class Comp over the domain of structures $\mathcal{D}$, when for every fixed sentence $\psi \in L$, the data complexity of evaluating ψ on structures from $\mathcal{D}$ is in complexity class Comp, and every property of

structures in $\mathcal{D}$ that can be decided with complexity COMP is definable in logic L [1].

The first important result of descriptive complexity theory was Fagin's theorem [2] stating that second-order existential logic captures the complexity class NP on the domain of all finite structures.

Another seminal result by Immerman and Vardi states that, on the domain of all linearly ordered structures, a problem is in P if and only if it is expressible in the extension, FO(LFP), of FO by the *least fixed point operator*, LFP. [3, 4]. Adding the least fixed point operator to FO can be thought of as adding the ability to iterate a formula polynomially many times. It is unknown, whether SO is strictly larger than FO(LFP) on domain of linearly ordered structures. This is unsurprising, as putting the two results together yields

$$\mathrm{P} = \mathrm{NP} \Leftrightarrow \mathrm{FO(LFP)} = \mathrm{SO},$$

which rewords the classic problem in computational complexity in the language of logic.

2 Dynamic Complexity Framework

2.1 Background

First-order queries on finite structures play a very interesting role in database theory as they have a one-to-one correspondence with the relational algebra, the core of SQL, the most widely used query language for databases. First-order logic over structures with "built-in" arithmetic operations also corresponds to the circuit complexity class AC^0, circuits that are of constant depth, may have polynomially many $\wedge$-, $\vee$-, and $\neg$ gates where the $\wedge$ and $\vee$ gates may have unbounded fan-in [5]. Circuits in AC^0 can be simulated in constant-time by parallel random access machines, PRAMs, with polynomially many processors [6], which in turn means that first order logic queries can be efficiently executed in parallel.

Unfortunately, many interesting queries cannot be expressed by first-order formulas. In particular, it has been shown that the transitive closure (TC) of a binary relation cannot be expressed by first order formulas [7]. This has motivated research into stronger logics such as fixed point logics or Datalog in database theory.

In practice, databases are often dynamic in nature i.e., we often have a set of queries that we wish to evaluate against a gradually changing set of relations. Therefore it would be desirable if we could take advantage of this and build up the

result to our query incrementally when relations in the database are being modified instead of having to rebuild the query from scratch on every modification.

This desire gives rise to the notion of Dynamic Complexity, independently formalized by Dong, Su and Topor [8] as first-order incremental evaluation system (FOIES) and equivalently by Patnaik and Immerman [9] as DynFO, the formulation we will be using.

In the dynamic complexity framework, the structure to be queried (called the input-relation) is updated tuple-by-tuple, and each time a tuple is inserted or deleted, a *dynamic program* may update one or more auxiliary relations, one of which is a designated *query relation* that contains the relation we are interested in.

Before we dive in to the formal definitions related to the dynamic complexity framework, we will go through an example of a query that is not in FO, but is easily maintained by a dynamic program.

Example 2.1. The parity query asks whether a unary relation U contains a number of elements divisible by two. A dynamic program can maintain the parity of U in the boolean query relation Q with the update formulas $\phi^Q_{\text{INS}_U}$ and $\phi^Q_{\text{DEL}_U}$.

When a new element gets inserted into the input relation U, the value of the query relation Q will be set to true if it was previously false. If the element was already present in U, the value of Q remains unchanged.

$$\phi^Q_{\text{INS}_U}(u) \stackrel{\text{def}}{=} (\neg U(u) \wedge \neg Q) \vee (U(u) \wedge Q)$$

The update on removal of an element is very similar:

$$\phi^Q_{\text{DEL}_U}(u) \stackrel{\text{def}}{=} (U(u) \wedge \neg Q) \vee (\neg U(u) \wedge Q)$$

2.2 Definitions

We will be using the definitions used by Thomas Zeume in his PhD thesis [10], which were introduced by Zeume and Schwentick in their joint publication, On the Quantifier-Free Dynamic Complexity of Reachability [11] and which are a variation of the definitions from Patnaik and Immerman [9].

2.2.1 Basic Notation

Let A be a finite set. We denote by A^k the set of all k-tuples over A, and by $[A]^k$ the set of all k-element subsets of A. For two tuples $\bar{a} = (a_1, ..., a_k)$ and $\bar{b} = (b_1, ..., b_l)$ over A, the $(k + l)$-tuple obtained by concatenating $\bar{a}$ and $\bar{b}$ is denoted by $(\bar{a}, \bar{b})$. We slightly abuse set-theoretic notations and write $c \in \bar{a}$ if $c = a_i$ for some i, and $\bar{a} \bigcup \bar{b}$ for the set $\{a_1, ..., a_k, b_1, ..., b_l\}$.

2.2.2 Structures and First-Order Logic

A *(relational) schema* τ consists of a set τ_{rel} of relation symbols and a set τ_{const} of constant symbols together with an arity function $\text{Ar} : \tau_{\text{rel}} \to \mathbb{N}$. A *domain* D is a finite set. A *database* $\mathcal{D}$ over a schema τ with a domain D is a mapping that assigns to every relation symbol $R \in \tau_{\text{rel}}$ a relation of arity $\text{Ar}(R)$ over D and to every constant symbol $c \in \tau_{\text{const}}$ an element (called *constant*) from D.

The set of *first-order formulas* over the schema τ is defined inductively as follows:

- Every *atomic formula* of the form $R(t_1, ..., t_2)$ or $t_1 = t_2$, where all t_i are either constant symbols or variables, is a first-order formula.
- If φ is a formula, then every *composed formula* of the form $\neg\varphi$, $\varphi \wedge \psi$, or $\exists x \varphi$ is a first-order formula.

The abbreviations $\vee$, $\to$, $\leftrightarrow$ and $\forall x$ are defined as usual.

A τ*-structure* $\mathcal{S}$ is a pair $(D, \mathcal{D})$ where $\mathcal{D}$ is a database over the schema τ and with the domain D. For a relation symbol $R \in \tau$ and a constant symbol $c \in \tau$ we denote by $R^{\mathcal{S}}$ and $c^{\mathcal{S}}$ the relation and constant, respectively, that are assigned to those symbols in $\mathcal{S}$.

Let $\mathcal{S} = (D, \mathcal{D})$ be a τ-structure, φ a first-order formula over τ with free variables $x_1, ..., x_k$, and α an assignment that maps every x_i to an element of D. By $(\mathcal{S}, \alpha) \vDash \varphi$ we indicate that $(\mathcal{S}, \alpha)$ is a model of φ. The satisfaction relation $\vDash$ is defined as usual.

An *m-ary query* $\mathcal{Q}$ on τ-structures is a mapping that is closed under isomorphisms and assigns a subset of D^m to every τ-structure over the domain D. *Closure under isomorphisms* means that $\pi(\mathcal{Q}(\mathcal{S})) = \mathcal{Q}(\pi(\mathcal{S}))$ for all isomorphisms π. Often we will denote $\mathcal{Q}(\mathcal{S})$ by $\text{ANS}(\mathcal{Q}, \mathcal{S})$.

A query $\mathcal{Q}$ is *definable* (alternatively: *expressible*) in first-order logic if there is a first-order formula $\varphi(\bar{x})$ such that $\text{Ans}(\mathcal{Q}, \mathcal{S}) = \{\bar{a} | (\mathcal{S}, \bar{a}) \vDash \varphi(\bar{x})\}$ for all structures $\mathcal{S}$.

For an arbitrary quantifier prefix $\mathbb{Q} \in \{\exists, \forall\}^*$, we will denote by $\mathbb{Q}FO$ the class of queries expressible by formulas with the quantifier prefix $\mathbb{Q}$.

2.2.3 The Dynamic Complexity Framework

A *dynamic instance* of a query $\mathcal{Q}$ is a pair $(\mathcal{D}, \alpha)$, where $\mathcal{D}$ is a database over some finite domain D and α is a sequence of modifications to $\mathcal{D}$. Here, a *modification* is either an insertion of a tuple over D into a relation of $\mathcal{D}$ or a deletion of a tuple from a relation of $\mathcal{D}$. The result of $\mathcal{Q}$ for $(\mathcal{D}, \alpha)$ is the relation that is

obtained by first applying the modifications from α to $\mathcal{D}$ and then evaluating $\mathcal{Q}$ on the resulting database. We use the Greek letters α and β to denote modifications as well as modification sequences. The database resulting from applying a modification α to a database $\mathcal{D}$ is denoted by $\alpha(\mathcal{D})$. The result $\alpha(\mathcal{D})$ of applying a sequence of modifications $\alpha \stackrel{\text{def}}{=} (\alpha_1, \ldots, \alpha_m)$ to a database $\mathcal{D}$ is defined by $\alpha(\mathcal{D}) \stackrel{\text{def}}{=} \alpha_m(\ldots(\alpha_1(\mathcal{D}))\ldots)$.

A *dynamic schema* is a tuple $(\tau_{\text{inp}}, \tau_{\text{aux}})$ where τ_{inp} and τ_{aux} are the schemas of the input database and the auxiliary database, respectively. While τ_{inp} may contain constants, we do not allow constants in τ_{aux} in the basic setting. We always let $\tau \stackrel{\text{def}}{=} \tau_{\text{inp}} \cup \tau_{\text{aux}}$.

Definition 2.2 (Update program)**.** An update program P over a dynamic schema $(\tau_{\text{inp}}, \tau_{\text{aux}})$ is a set of first-order formulas (called update formulas in the following) that contains, for every relation symbol R in τ_{aux} and every $\delta \in \{\text{INS}_S, \text{DEL}_S\}$ with $S \in \tau_{\text{inp}}$, an update formula $\varphi^R_\delta(\bar{x}, \bar{y})$ over the schema τ where $\bar{x}$ and $\bar{y}$ have the same arity as S and R, respectively.

A *program state* $\mathcal{S}$ over the dynamic schema $(\tau_{\text{inp}}, \tau_{\text{aux}})$ is a structure $(D, \mathcal{I}, \mathcal{A})$ where D is a finite domain, $\mathcal{I}$ is a database over the input schema (the *current database*) and $\mathcal{A}$ is a database over the auxiliary schema (the *auxiliary database*).

The semantics of update programs is as follows. Let P be an update program, $\mathcal{S} = (D, \mathcal{I}, \mathcal{A})$ be a program state and $\alpha = \delta(\bar{a})$ a modification where $\bar{a}$ is a tuple over D and $\delta \in \text{INS}_S, \text{DEL}_S$ for some $S \in \tau_{\text{inp}}$ wath arity equal to that of $\bar{a}$. If P is in state $\mathcal{S}$ then the application of α yields the new state $\mathcal{P}_\alpha(\mathcal{S}) \stackrel{\text{def}}{=} (D, \alpha(\mathcal{I}), \mathcal{A}')$ where, in $\mathcal{A}'$, relation symbols $R \in \tau_{\text{aux}}$ are interpreted by $\{\bar{b} | \mathcal{S} \models \varphi^R_\delta(\bar{a}, \bar{b})\}$. The effect $P_\alpha(\mathcal{S})$ of applying a modification sequence $\alpha \stackrel{\text{def}}{=} (\alpha_1, \ldots, \alpha_m)$ to a state $\mathcal{S}$ is the state $\mathcal{P}_{\alpha_m}(\ldots(\mathcal{P}_{\alpha_1}(\mathcal{S}))\ldots)$.

Definition 2.3 (Dynamic program)**.** A dynamic program is a triple $(P, \textsc{Init}, Q)$, where

- P is an update program over some dynamic schema $(\tau_{\text{inp}}, \tau_{\text{aux}})$
- $\textsc{Init}$ is a mapping that maps τ_{inp}-databases to τ_{aux}-databases, and
- $Q \in \tau_{\text{aux}}$ is a designated query symbol.

A dynamic program $\mathcal{P} = (P, \textsc{Init}, Q)$ *maintains* a query $\mathcal{Q}$ if, for every dynamic instance $(\mathcal{D}, \alpha)$, the relation $\textsc{Ans}(\mathcal{Q}, \alpha(\mathcal{D}))$ coincides with the query relation $Q^{\mathcal{S}}$ in the state $\mathcal{S} = P_\alpha(\mathcal{S}_{\textsc{Init}(\mathcal{D})})$ where $\mathcal{S}_{\textsc{Init}(\mathcal{D})}$ is the initial state for $\mathcal{D}$, that is, $\mathcal{S}_{\textsc{Init}(\mathcal{D})} \stackrel{\text{def}}{=} (D, \mathcal{D}, \textsc{Init}(\mathcal{D}))$.

Definition 2.4 (Dyn$\mathcal{C}$). For some fragment of FO $\mathcal{C}$, we call Dyn$\mathcal{C}$ the class of dynamic queries that can be maintained by dynamic programs with updates formulas in $\mathcal{C}$.

2.2.4 DynFO

Definition 2.5 (DynFO). DynFO is the class of queries that can be maintained by dynamic programs defined by first-order update formulas formulas and arbitrary initialization mappings.

DynFO is the most extensively studied of the dynamic complexity classes and it is the one we will also pay the most attention to.

We will follow the argument from [9] and construct a DynFO-program with one binary auxiliary relation T which is intended to store the transitive closure of an acyclic graph.

Example 2.6. Insertions can be handled straightforwardly: after inserting an edge (u, v) there is a path from x to y if, before the insertion, there has been a path from x to y or there have been paths from x to u and from v to y. There is a path p from x to y after deleting an edge (u, v) if and only if there was a path from x to y before the deletion and 1. there was no such path via (u, v), or 2. there is an edge (z, z') on p such that u can be reached from z but not from z', as if such an edge would not exist, u would be reachable from y which would contradict acyclicity. All conditions can be checked using the transitive closure of the graph before deletion of (u, v). The update formulas for T are as follows:

$$\phi^T_{\text{INS}_E}(u, v; x, y) \stackrel{\text{def}}{=} T(x, y) \lor (T(x, u) \land T(v, y))$$

$$\begin{aligned}\phi^T_{\text{DEL}_E}(u, v; x, y) \stackrel{\text{def}}{=} & T(x, y) \land ((\neg T(x, u) \lor \neg T(v, y)) \\ & \lor \exists z \exists z' (T(x, z) \land E(z, z') \land (z \neq u \lor z' \neq v) \\ & \land T(z', y) \land T(z, u) \land \neg T(z', u))).\end{aligned}$$

With this, we have the required dynamic program $\mathcal{P} = (P, \textsc{Init}, Q)$ where the update program P contains the update formulas $\phi^T_{\text{INS}_E}$ and $\phi^T_{\text{DEL}_E}$, Init is a mapping that takes the initial state of the input database to its transitive closure, and $Q = T$.

In the upcoming proofs, we will generally only show the update formulas, since we allow arbitrary initialization functions, constructing the update program from the formulas is trivial.

2.2.5 DynProp

A rather natural restriction to DynFO is to disallow the use of quantifiers in update formulas, and exactly this limitation yields DynProp.

Definition 2.7 (DynProp)**.** DynProp is the class of all queries that can be maintained by dynamic programs with quantifier-free first-order update formulas and arbitrary initialization mappings.

It would first seem that disallowing all quantification would severly limit the usefulness of DynProp, but it turns out that many interesting queries are in DynProp. As Example 2.1 shows, no quantification is needed to express parity.

Example 2.8. We will now show how the boolean query NonEmptySet can be implemented by a DynProp program $\mathcal{P}$. NonEmptySet is a query that says whether a unary relation U is empty or not.

The input schema for $\mathcal{P}$ will be $\tau_{\text{inp}} = \{U\}$, where U is the unary relation that is subject to insertions and deletions. The auxiliary schema $\tau_{\text{aux}} = \{Q, \textsc{First}, \textsc{Last}, \textsc{List}\}$ consists of the boolean query relation Q, which will contain $\top$ if U is non-empty and $\bot$ if it's empty. First, Last and List will be used to maintain a list of elements in U, First and Last are unary relations that contain the first and last elements respectively and List contains all pairs (a, b) where a and b are adjacent in the list.

Inserting an element a into U: The element a is added to the end of the list. If it already occurs somewhere in the list, the old instance of it will be removed so the list only contains unique occurences of each element. Since we take care of duplicate entries on insertion, we will only need to handle removing one element at a time. After adding an element, the input relation is always non-empty and Q will be set to true.

$$\begin{aligned}
\phi^{\textsc{First}}_{\textsc{ins}_U}(a; x) &\stackrel{\text{def}}{=} (\neg Q \wedge a = x) \vee (Q \wedge \textsc{First}(x) \wedge x \neq a) \vee (Q \wedge \textsc{First}(x) \wedge \textsc{Last}(x)) \\
\phi^{\textsc{Last}}_{\textsc{ins}_U}(a; x) &\stackrel{\text{def}}{=} a = x \\
\phi^{\textsc{List}}_{\textsc{ins}_U}(a; x, y) &\stackrel{\text{def}}{=} x \neq a \wedge y \neq a \wedge (\textsc{List}(x, y) \vee (\textsc{List}(x, a) \wedge \textsc{List}(a, y)) \\
&\qquad \vee (\textsc{Last}(x) \wedge a = y)) \\
\phi^{\text{Q}}_{\textsc{ins}_U}(a; x) &\stackrel{\text{def}}{=} \top
\end{aligned}$$

Deleting the element a from U: Deleting works much like insertion except that no nodes are inserted and the last element needs a bit more consideration. The

query relation will be true unless the first and the last element are the element to be removed or the query was false to begin with.

$$
\begin{aligned}
\phi^{\textsc{First}}_{\textsc{del}_U}(a;x) &\stackrel{\text{def}}{=} (\textsc{First}(x) \wedge a \neq x) \vee (\textsc{First}(a) \wedge \textsc{List}(a,x)) \\
\phi^{\textsc{Last}}_{\textsc{del}_U}(a;x) &\stackrel{\text{def}}{=} (\textsc{Last}(x) \wedge a \neq x) \vee (\textsc{Last}(a) \wedge \textsc{List}(x,a)) \\
\phi^{\textsc{List}}_{\textsc{del}_U}(a;x,y) &\stackrel{\text{def}}{=} x \neq a \wedge y \neq a \wedge (\textsc{List}(x,y) \vee (\textsc{List}(x,a) \wedge \textsc{List}(a,y))) \\
\phi^{\text{Q}}_{\textsc{del}_U}(a;x) &\stackrel{\text{def}}{=} \neg((\textsc{First}(a) \wedge \textsc{Last}(a)) \vee \neg Q).
\end{aligned}
$$

2.2.6 DynQF

While programs in DynFO have great freedom in accessing tuples during updates using quantifiers, DynProp programs can only access the tuple being inserted or removed and the tuple that is currently being accessed. DynQF falls between the two by allowing auxiliary functions to be maintained in addition to auxiliary relations. These auxiliary functions allow quantifier-free formulas to access additional tuples that they otherwise could not, working as a sort of weakened version of quantification.

To define DynQF, we need to first expand the definition of a dynamic program to include *update terms*, expressions that may be used as parts of the update formulas.

Definition 2.9 (Update term). Update terms are inductively defined as follows:

1. Every variable and every constant is an update term.
2. If f is a k-ary function symbol and $t_1, ..., t_k$ are update terms, then $f(t_1, ..., t_k)$ is an update term.
3. If ϕ is a quantifier-free update formula (possibly using update terms) and t_1 and t_2 are update terms, then $\text{ITE}(\phi, t_1, t_2)$ is an update term.

The semantics of update terms associates with every update term t and interpretation $I = (\mathcal{S}, \beta)$, where $\mathcal{S}$ is a state and β a variable assignment, a value $[\![t]\!]_I$ from S. The semantics of 1) and 2) are straightforward. If $I \vDash \phi$ holds, then $[\![\text{ITE}(\phi, t_1, t_2)]\!]_I$ is $[\![t_1]\!]_I$, otherwise $[\![t_2]\!]_I$.

The extension of the notion of update programs for auxiliary schemas with function symbols is now straightforward. An update program still has an update formula ϕ^R_δ (possibly using update terms instead of only variables and constants)

for every relation symbol $R \in \tau_{\text{aux}}$ and every $\delta \in \text{INS}_S, \text{DEL}_S$ with $S \in \tau_{\text{inp}}$. Furthermore, it has, for every such δ and every function symbol $f \in \tau_{\text{aux}}$, an update term $t_\delta^f(\bar{x}, \bar{y})$. For a modification $\delta(\bar{a})$ it redefines f for each tuple $\bar{b}$ by evaluating $t_\delta^f(\bar{a}, \bar{b})$ in the current state.

Definition 2.10 (DynQF)**.** DynQF is the class of queries maintainable by quantifier-free update programs with (possibly) auxiliary functions and arbitrary initialization mappings.

Example 2.11. We will now construct a DynQf program that maintains a graph query $\mathcal{Q}$ that returns the set of nodes with maximal outdegree in the input graph G.

With the help of two functions Succ and Pred, we will treat the domain as if were of the form $D = \{0, ..., n-1\}$. For all the states $\mathcal{S}$, $\text{Succ}^{\mathcal{S}}$ is the standard successor function on D with $\text{Succ}^{\mathcal{S}}(n-1) = n-1$ and $\text{Pred}^{\mathcal{S}}$ the standard predecessor function with $\text{Pred}^{\mathcal{S}}(0) = 0$. Both functions are initialized for the whole domain and neither will be updated on modifications.

When we refer to numbers, 0 is a constant with a value from D (the only value $x \in D$ for which $\text{Pred}(x) = x$) and any other number n refers to the element $x \in D$ so that

$$\underbrace{\text{Succ}(\text{Succ}(...}_{n \text{ applications of Succ}} (0)...)) = x$$

The program will maintain the unary functions #Edges and #nodes. The function #edges maintains the outdegree for all the nodes in the graph so that $\#\text{edges}(a) = b$ if b is the number of outgoing edges from a and #nodes maintains the amount of nodes in the graph with a certain number of outgoing edges, that is $\#\text{nodes}(a) = b$ if there are b nodes with a outgoing edges, for all $a, b \in D$. A constant Max is maintained so that it always points to the number i such that i is the maximal number of outgoing edges for any node in the graph.

When a new edge (u, v) is inserted for the node u that has a outgoing edges, $\#edges(u)$ is updated from a to $a+1$, $\#\text{nodes}(a)$ is decremented and $\#\text{nodes}(a+1)$ is incremented. Max is incremented if u was a node with maximal outgoing edges before the insertion. With this, we get the following update terms:

$$t^{\#\text{EDGES}}_{\text{INS}_E}(u,v;x) \stackrel{\text{def}}{=} \text{ITE}(\neg E(u,v) \wedge x = u, \textsc{Succ}(\#\text{EDGES}(x)), \#\text{EDGES}(x))$$

$$\begin{aligned} t^{\#\text{NODES}}_{\text{INS}_E}(u,v;x) \stackrel{\text{def}}{=} \text{ITE}(&\neg E(u,v) \wedge x = \#\text{EDGES}(u), \textsc{Pred}(\#\text{NODES}(x)), \\ &\text{ITE}(\neg E(u,v) \wedge x = \textsc{Succ}(\#\text{EDGES}(u)), \\ &\quad \textsc{Succ}(\#\text{NODES}(x)), \#\text{NODES}(x))) \end{aligned}$$

$$t^{\textsc{Max}}_{\text{INS}_E}(u,v) \stackrel{\text{def}}{=} \text{ITE}(\textsc{Max} = \#\text{EDGES}(u) \wedge \neg E(u,v), \textsc{Succ}(u), \textsc{Max})$$

The update formula for the designated query symbol Q:

$$\phi^{Q}_{\text{INS}_E}(u,v;x) \stackrel{\text{def}}{=} t^{\#\text{EDGES}}_{\text{INS}_E}(u,v;x) = t^{\textsc{Max}}_{\text{INS}_E}(u,v)$$

The update terms for deletion are similar:

$$t^{\#\text{EDGES}}_{\text{DEL}_E}(u,v;x) \stackrel{\text{def}}{=} \text{ITE}(E(u,v) \wedge x = u, \textsc{Pred}(\#\text{EDGES}(x)), \#\text{EDGES}(x))$$

$$\begin{aligned} t^{\#\text{NODES}}_{\text{DEL}_E}(u,v;x) \stackrel{\text{def}}{=} \text{ITE}(&E(u,v) \wedge x = \#\text{EDGES}(u), \textsc{Pred}(\#\text{NODES}(x)), \\ &\text{ITE}(E(u,v) \wedge x = \textsc{Pred}(\#\text{EDGES}(u)), \\ &\quad \textsc{Succ}(\#\text{NODES}(x))), \#\text{NODES}(x))) \end{aligned}$$

$$\begin{aligned} t^{\textsc{Max}}_{\text{DEL}_E}(u,v) \stackrel{\text{def}}{=} \text{ITE}(&\textsc{Max} = \#\text{EDGES}(u) \wedge E(u,v) \wedge \#\text{NODES}(\textsc{Max}) = \textsc{Succ}(0), \\ &\textsc{Pred}(\textsc{Max}), \textsc{Max}) \end{aligned}$$

And the update formula for the designated query symbol for deletions:

$$\phi^{Q}_{\text{DEL}_E}(u,v;x) \stackrel{\text{def}}{=} t^{\#\text{EDGES}}_{\text{DEL}_E}(u,v;x) = t^{\textsc{Max}}_{\text{DEL}_E}(u,v).$$

2.3 Reachability in Undirected Graphs in DynFO

It was shown by Patnaik and Immerman in [9] that reachability in acyclic graphs is in DynFO.

The proof is an adaptation of the one shown by Patnaik and Immerman, it works by constructing a dynamic program that maintains a spanning forest of the input graph, that is, a subgraph with vertices removed so that it is acyclic but has the same connected components as the original graph. Therefore, if there is a path in forest from node x to node y, a path must also exist in the input graph.

Theorem 2.12. *$\textsc{Reach}_u$ is in DynFO*

Proof. Our input schema τ_{inp} will consist only of the edge relation E. Since our graph is undirected, we will interpret the addition of an edge (x, y) to also add the edge (y, x).

We will use two auxiliary relations, F which will contain a tuple (x, y) if the edge (x, y) is in the spanning forest and PV, which will contain tuples (x, y, u) if there exists a path from x to y via the vertex u, which may be equal to x or y.

For convenience, we will define the abbreviations $P(x, y)$ and $Eq(x, y, c, d)$ for testing whether a path exists from x to y and testing if the edges (x, y) and (c, d) are equal, respectively.

$$P(x, y) \stackrel{\text{def}}{=} (x = y \vee PV(x, y, x))$$
$$Eq \stackrel{\text{def}}{=} (x = c \wedge y = d) \vee (x = d \wedge y = c).$$

Maintaining the input graph on insertions is easy, we simply insert both u, v and v, u.

$$\phi^{E}_{\text{INS}_E}(u, v; x, y) \stackrel{\text{def}}{=} E(x, y) \vee Eq(u, v, x, y)$$

If the added vertices u and v were already in the same connected component, the edges in the forest stay the same. If u and v were not connected, the edge (u, v) is added to the forest.

$$\phi^{F}_{\text{INS}_E}(u, v; x, y) \stackrel{\text{def}}{=} F(x, y) \vee (Eq(x, y, u, v) \wedge \neg P(u, v))$$

PV changes if and only if the added edge (u, v) connects two trees that were previously disconnected. If that is the case, all the tuples x, y, z where x and y are in the previously disjoint trees and z is in either one will be added to PV

$$\begin{aligned}\phi^{PV}_{\text{INS}_E}(u, v; x, y, z) \stackrel{\text{def}}{=} & PV(x, y, z) \vee \\ & (\exists ab)(Eq(a, b, u, v) \wedge P(x, a) \wedge P(b, y) \wedge (PV(x, a, z) \vee PV(b, y, z)))\end{aligned}$$

When deleting the edge (u, v), if it is not in the forest, the relations stay the same. If it is, we identify the vertices of the two trees created by the deletion, and if there exists an edge in the input graph between nodes of the two trees, we pick one and insert it into the forest.

We will use $T(x, y, z)$ as an abbreviation for the PV relation after the node (u, v) has been deleted. Since PV is acyclic, if there were paths from x to y via u and v, the forest cannot have any path between x and y after the removal of (u, v).

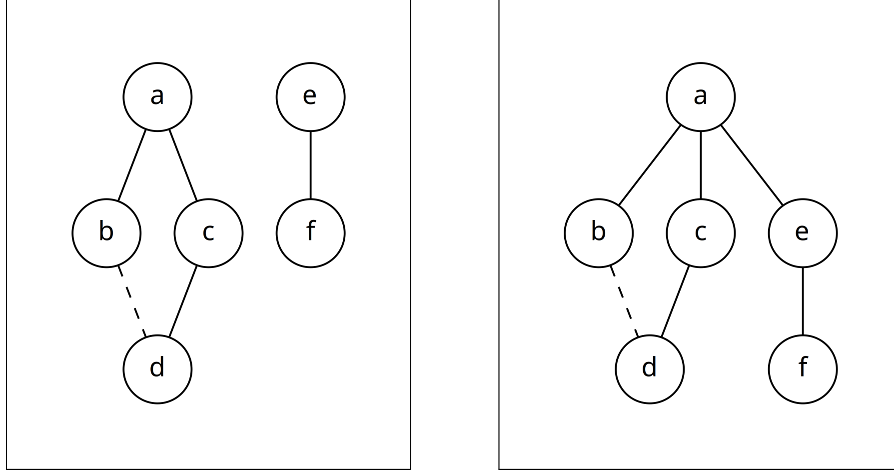

Figure 1: Inserting a new edge (a, e). Solid lines are edges both in E and in F, while dashed lines are only in E.

$$T(x, y, z) \stackrel{\text{def}}{=} PV(x, y, z) \wedge \neg(PV(x, y, u) \wedge PV(x, y, v))$$

Now, using T we can define another abbreviation, $New(x, y)$ which will either be empty or contain the single edge x, y from T that is needs to be added to the tree in the forest to make it connected after the removal of the node (u, v).

$$New(x, y) \stackrel{\text{def}}{=} E(x, y) \wedge T(u, x, u) \wedge T(v, y, v) \wedge$$
$$(\forall ab)((E(a, b) \wedge T(u, a, u) \wedge T(v, b, v)) \rightarrow (x < a \vee (x = a \wedge (y = b \vee y < b)))$$

Note that here we use an ordering on the nodes. If no such ordering exists, we can easily construct one with a new auxiliary relation into which all new nodes are added upon insertion into E.

We can now define the update formulas for deletion:

$$\phi^{E}_{\text{DEL}_E}(u, v; x, y) \stackrel{\text{def}}{=} E(x, y) \wedge \neg Eq(x, y, u, v)$$

We remove the edge u, v from the forest and possibly add the new edge:

$$\phi^{F}_{\text{DEL}_E}(u, v; x, y) \stackrel{\text{def}}{=} (F(x, y) \wedge \neg Eq(x, y, u, v)) \vee New(x, y) \vee New(y, x)$$

If a path from x to y via z did not go through the edge between u and v, it is still valid. If New contained an edge, paths passing through it need to be added.

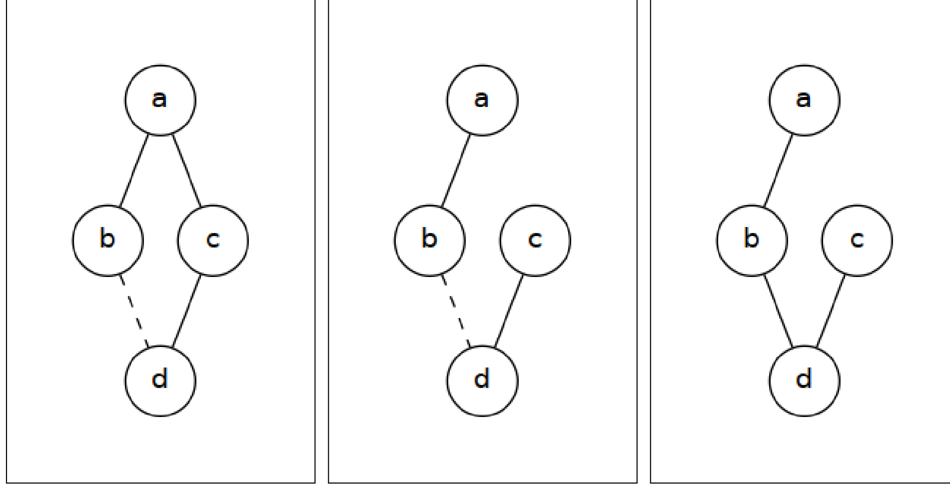

Figure 2: Deleting the edge (a, c). Solid lines are edges both in E and in F, while dashed lines are only in E.

$$\phi^{PV}_{\text{DEL}_E}(u, v; x, y, z) \stackrel{\text{def}}{=} T(x, y, z) \vee ((\exists ab)(New(a, b) \vee New(b, a)) \\ \wedge T(x, a, x) \wedge T(y, b, y) \wedge (T(x, a, z) \vee T(y, b, z)))$$

□

3 Relating Dynamic Complexity Classes

One of the major goals in descriptive complexity theory is to compare the power of different logics, whether queries expressible in one are expressible in another. Sometimes, like in the case of propositional logic and first-order logic, the relation is obvious but other times, like in the case of SO and FO+LFP, it is not. If we know how logics relate to each other in terms of expressive power, we can more easily say if a given query is expressible in a certain logic or not, and how computationally demanding it's evaluation is.

In a similar manner, we can compare different complexity classes with each other. Knowing that two complexity classes, Dyn$\mathcal{C}$ and Dyn$\mathcal{C}'$ induced by static complexity classes $\mathcal{C}$ and $\mathcal{C}'$ are equal means that queries maintainable by in one are also maintainable in the other.

We are also interested in comparing dynamic classes to static ones, of specific interest are cases of a dynamic class Dyn$\mathcal{C}'$ and a static class $\mathcal{C}$ where the static

class $\mathcal{C}'$ that induces DynC$'$ is smaller than $\mathcal{C}$. Finding instances of this would mean that we can maintain queries using a simpler logic, which can in turn lead to lower computational complexity.

Most of the dynamic complexity classes that have been studied are fragments of DynFO that have been obtained from DynFO by restricting update programs in one of the following ways: restricting the arity of auxiliary relations used, restricting the syntax of update formulas or restricting the initialization mapping.

We will now informally introduce some of the restrictions of FO that have been studied as update languages in dynamic complexity.

Conjunctive queries (CQs) or first-order queries with a prefix of existential quantifiers and whose quantifier-free part consists of a conjunction of atoms, that is formulas of the form $\exists x_1 ... x_n \bigwedge_i \varphi_i$ Unions of conjunctive queries (UCQs) are formulas of the form $q_1 \cup ... \cup q_m$, where each $q_i, i \in \mathbb{N}$ is a conjunctive query. Conjunctive queries with negations (CQ$^\neg$s) are conjunctive queries but atoms can be negated, and unions of conjunctive queries with negations (UCQ$^\neg$s) are like UCQs but atoms can again be negated.

If existential quantification from FO is disallowed, we get the corresponding classes PropCQ, PropUCQ, PropCQ$^\neg$ and PropUCQ$^\neg$. It is thought that all of these static complexity classes except UCQ$^\neg$ and Σ_1^0, the fragment of FO that allows only existential quantifiers, are distinct for relational databases. Figure 3 shows how these different fragments of FO relate to each other.

Another variation of the dynamic framework that has been studied can be obtained by using Δ-*semantics*. So far, all updates to relations have worked by re-defining them upon every insertion or deletion. This will be referred to as *absolute semantics*, in contrast to Δ-semantics, where the new state of a relation R is defined in terms of a set of tuples R^+ to be added to R and R^- that will be removed from it. For update languages that are closed under Boolean operations, Δ-semantics and absolute semantics coincide trivially but some of the query languages that have been studied, like conjunctive queries, are not closed under all Boolean operations.

Zeume gives the hierarchy shown in Figure 4 showing the relations between dynamic classes induced by different fragments of FO under absolute and Δ-semantics. It is notable that so many of the dynamic complexity classes turn out to be equal.

We will not go through all of the proofs necessary to obtain the equivalences and separations shown in Figure 4 but we will introduce some of the techniques used in them and give examples that demonstrate their use.

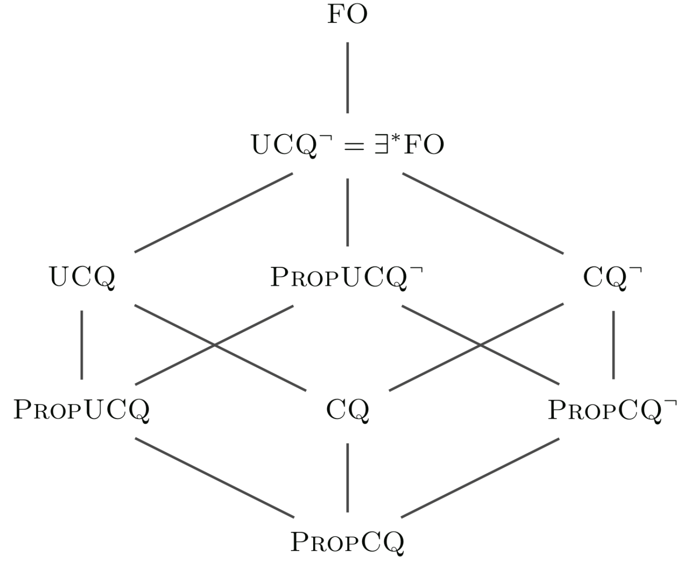

Figure 3: Hierarchy of fragments of first-order logic. Solid lines are strict separations. Figure from [10].

3.1 Techniques for Collapsing Dynamic Classes

Many of the proofs that show that a class Dyn$\mathcal{C}$ is contained in a class Dyn$\mathcal{C}'$ are pretty similar in structure, showing that for all programs with update queries from class $\mathcal{C}$ an equivalent program with update queries from class $\mathcal{C}'$ can be constructed. Many of the proofs also use one on more of the following three techniques.

The *replacement technique* is used to replace certain types of subformulas with additional auxiliary relations that do something equivalent. The replacement technique can be used to remove negations or disjunctions in formulas.

The *preprocessing technique* simplifies complicated subformulas in update formulas by performing a part of it in the initialization step and storing it in an additional auxiliary relation that is used in the simpler update formula. This technique can be used to remove unions from dynamic unions of conjunctive queries and proving that dynamic conjunctive queries with negations are equally expressive under absolute and Δ-semantics.

The *squirrel technique* maintains additional auxiliary relations that reflect on the state of some auxiliary relation after every possible single modification (or short modification sequence). For a relation R, a new relation symbol R_{INS} can be maintained so that the interpretation of R_{INS} contains what R would contain

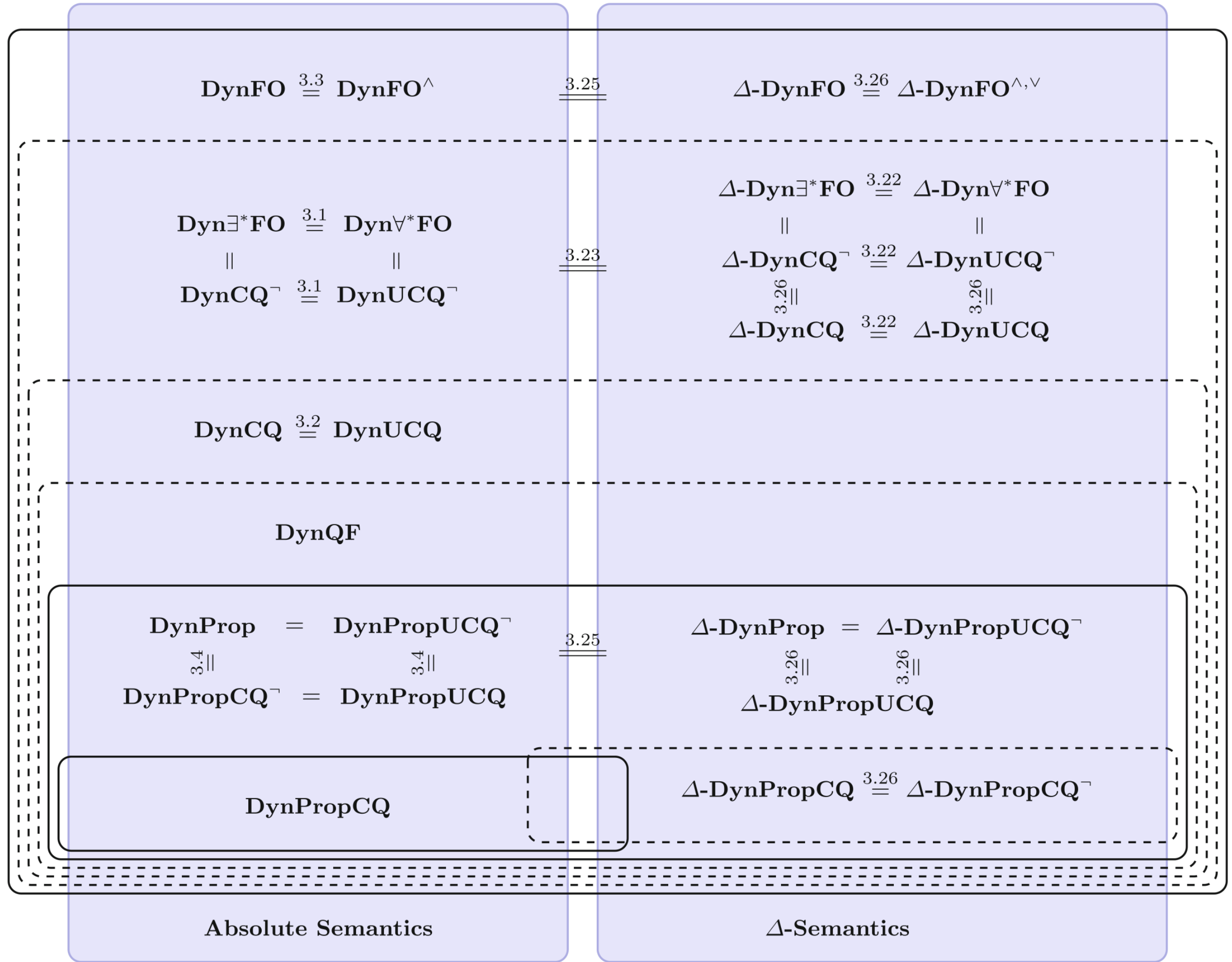

Figure 4: Hierarchy of fragments of DynFO. Solid lines are strict separations. Classes above a solid or dashed line contain those under it. Figure from [10]. Numbers on equalities refer to theorems in [10].

after every insertion modification.

3.2 Collapsing Dynamic Complexity Classes

We will now start working towards proving the equalities that form the left side of Figure 4. As most of the theorems we are working to are about showing the equality of various dynamic classes, most of our proofs revolve around taking a dynamic program $\mathcal{P}$ with update formulas from a static class $\mathcal{C}$, and transforming it to an equivalent program $\mathcal{P}'$ with update formulas from a different static class $\mathcal{C}'$.

We will start by giving formal definitions of the covered restrictions of FO that induce the DynFO fragments we will be examining.

Definition 3.1.

- CQ is the class of conjunctive queries, that is, first-order queries of the form $\varphi(\bar{x}) = \exists\bar{y}\psi$ where ψ is a conjunction of atomic formulas.
- UCQ is the class of all unions of conjunctive queries, that is conjunctive queries expressible by formulas of the form $\varphi(\bar{x}) = \bigvee_i \exists\bar{y}\psi_i$ where each ψ_i is a conjunction of atomic formulas.
- $\text{CQ}^{\neg}$ and $\text{UCQ}^{\neg}$ are the same as CQ and UCQ but atoms are allowed to be negated.
- PropCQ, PropUCQ, $\text{PropCQ}^{\neg}$ and $\text{PropUCQ}^{\neg}$ are as above, but quantifiers are disallowed.

Lemma 3.2. *Let $\mathcal{Q}$ be an arbitrary quantifier prefix. For every $\text{Dyn}\mathcal{Q}$FO-program there is an equivalent $\text{Dyn}\mathcal{Q}$FO-program $\mathcal{P}$ such that the update formulas for the designated query symbol of $\mathcal{P}$ consists of a single atom.*

Proof. In this proof, the squirrel technique will be used. For simplicity, the input schema will be fixed to be $\tau_{\text{inp}} = \{E\}$, where E is a binary relation symbol, but the proof can easily be adapted to arbitrary input schemas.

Let $\mathcal{P}$ be a $\text{Dyn}\mathcal{C}$-program over the auxiliary schema τ with the designated query symbol Q. We will construct an equivalent $\text{Dyn}\mathcal{C}$ program over the schema τ' where τ' contains a designated query symbol $\mathcal{Q}'$ and a k+2-ary relation symbol R_δ for every k-ary $R \in \tau$ and every $\delta \in \{\text{INS}, \text{DEL}\}$.

The idea is that R_δ will reflect the state of R in the next state, for each possible modification of kind δ. Let $G = (E, V)$ be a graph , α a sequence of modifications, $\beta = \delta(\bar{e})$ a modification with $\delta \in \{\text{INS}, \text{DEL}\}$ and $\bar{e} \in V^2$. If $\mathcal{S}$ is the state obtained

by $\mathcal{P}$ after applying $\alpha\beta$ to G, that is, $\mathcal{S} = \mathcal{P}_{\alpha\beta}(\textsc{Init}(G))$, and $\mathcal{S}'$ is the state obtained by $\mathcal{P}$ after applying α to G, i.e. $\mathcal{S}' = \mathcal{P}'_\alpha(\textsc{Init}'(G))$, then

$$\bar{a} \in R^{\mathcal{S}} \text{ if and only if } (\bar{e}, \bar{a}) \in R_\delta^{\mathcal{S}'}. \tag{1}$$

Thus for every $\delta(\bar{e})$ the relation $R_\delta(\bar{e}, \cdot)$ stores $R(\cdot)$ after the application of $\delta(\bar{e})$.

Then the update formula for Q' after a modification δ can be written with a single-atomic formula as follows:

$$\phi_\delta^{Q'}(\bar{u}, \bar{x}) \stackrel{\text{def}}{=} R_\delta(\bar{u}, \bar{x}).$$

It remains to show how to update the relations R_δ. Therefore it will be convenient to assume that the edge relation E is updated by formulas ϕ^E_{INS} and ϕ^E_{DEL} that express the impact of a modification to E, for example, $\phi^E_{\text{INS}}(a, b; x, y) = E(x, y) \vee (a = x \wedge b = y)$.

By $\phi^R_{\delta_i}[\tau \to \tau_{\delta_0}](\bar{u}_0; \bar{u}_1, \bar{x})$ we denote the formula obtained from $\phi^R_{\delta_i}(\bar{u}_1; \bar{x})$ by replacing every atom $S(\bar{z})$ with $S \in \tau$ by $S_{\delta_0}(\bar{u}_o, \bar{z})$. Then the update formula for R is

$$\phi_{\delta_0}^{R_{\delta_1}}(\bar{u}_0; \bar{u}_1, \bar{x}) \stackrel{\text{def}}{=} \phi^R_{\delta_1}[\tau \to \tau_{\delta_0}](\bar{u}_0; \bar{u}_1, \bar{x}).$$

We observe that all quantifier prefixes of formulas thus obtained have been used by the program $\mathcal{P}$ already.

The initialization mapping of $\mathcal{P}'$ is as follows. The query symbol Q' is initialized like Q in $\mathcal{P}$. For every graph G the relation symbol $R_\delta \in \tau'$ is initialized as

$$\bigcup_{\bar{e} \in V^2} \bar{e} \times \mathcal{P}_{\delta(\bar{e})}(\textsc{Init}(G)) \restriction R_\delta$$

where $\mathcal{P}_{\delta(\bar{e})}(\textsc{Init}(G)) \restriction R_\delta$ denotes the relation R_δ in the state $\mathcal{P}_{\delta(\bar{e})}(\textsc{Init}(G))$.

The correctness of this construction is proved inductively over the length of modification sequences by showing that states of $\mathcal{P}'$ simulate states of $\mathcal{P}$ as specified by Equation 1.

Therefore, let G be a graph and $\alpha = \alpha_1 \ldots \alpha_i$ a modification sequence with $\alpha_i = \delta_i(\bar{e}_i)$. Further, let $\mathcal{S}_i$ and $\mathcal{S}'_i$ be the states obtained by $\mathcal{P}$ and $\mathcal{P}'$ respectively, after the application of $\alpha_1 \ldots \alpha_i$.

If α is of length 0 and β is an arbitrary modification with $\beta = \delta(\bar{e})$ then $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}_\beta(\mathcal{S}_0)$ and $\mathcal{S}' \stackrel{\text{def}}{=} \mathcal{S}'_0$ satisfy Equation 1 due to the definition of the initialization mapping of $\mathcal{P}'$. If α is of length $i \geq 1$ then, by the induction hypothesis, the states $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{S}_i = \mathcal{P}_{\alpha_i}(\mathcal{S}_{i-1})$ and $\mathcal{S}' \stackrel{\text{def}}{=} \mathcal{S}'_{i-1}$ satisfy Equation 1, that is

$$\bar{a} \in R^{\mathcal{S}} \text{ if and only if } (\bar{e}_i, \bar{a}) \in R_{\delta_i}^{\mathcal{S}'} \tag{2}$$

for all relations R and R_{δ_i}.

Now, let $\beta = \delta(\bar{e})$ be an arbitrary modification. Further, let $\mathcal{T} \stackrel{\text{def}}{=} \mathcal{P}_\beta(\mathcal{S})$ and $\mathcal{T}' \stackrel{\text{def}}{=} \mathcal{P}'_{\alpha_i}(\mathcal{S}')$. By definition, $\bar{b} \in R^{\mathcal{T}}$ if and only if

$$(R^{\mathcal{S}}, \{\bar{u}_1 \mapsto \bar{e}, \bar{x} \mapsto \bar{b}\}) \vDash \phi_\delta^R(\bar{u}_1; \bar{x}).$$

Thanks to Equation 2 and the definition of $\phi_{\delta_0}^{R_{\delta_i}}$, this is equivalent to

$$(R^{\mathcal{S}'}, \{\bar{u}_0 \mapsto \bar{e}_i, \bar{u}_1 \mapsto \bar{e}, \bar{x} \mapsto \bar{b}\}) \vDash \phi_{\delta_i}^R[\tau \to \tau_{\delta_0}](\bar{u}_0, \bar{u}_1, \bar{x}).$$

And by definition, this is equivalent to $(\bar{e}, \bar{b}) \in R^{\mathcal{T}}$. $\square$

The next two lemmas will be about removing negations and inverting quantifiers.

Lemma 3.3.

(a) Every DynFO-program has an equivalent negation-free DynFO-program and

(b) Every DynProp-program has an equivalent DynPropUCQ-program

Proof. This lemma is a generalization of Theorem 6.6 from [12] and is an example of using the replacement technique. Given a dynamic program $\mathcal{P}$, the idea is to maintain, for every auxiliary relation R of $\mathcal{P}$, an additional auxiliary relation $\hat{R}$ for the complement of R, and replace all negations with them. This technique of replacing negations with negated relations will be used again in multiple proofs.

In the following we prove (a), and as the proof does not introduce quantifiers, it can also be used as a proof for (b).

Let $\mathcal{P} = (P, \textsc{Init}, Q)$ be a DynFO-program over the schema τ. We will assume without loss of generality that $\mathcal{P}$ is in negation normal form and, for ease of presentation, that the input relations also have update formulas.

We will construct a negation-free DynFO-program that is equivalent to $\mathcal{P}$ and that uses the schema $\tau \cup \hat{\tau} \cup \{\hat{=}\}$ where $\hat{\tau}$ contains, for every symbol R, a fresh relation symbol $\hat{R}$ of equal arity. The idea is to maintain the negation of $R^{\mathcal{S}}$ for every state $\mathcal{S}$ in $\hat{R}^{\mathcal{S}}$, and the complement of $=$ in $\hat{=}$.

We will start by constructing a DynFO-program $\mathcal{P}' = (P', \textsc{Init}', Q)$ in negation normal form over the schema $\tau \cup \hat{\tau} \cup \{\hat{=}\}$ that maintains $R^{\mathcal{S}}$ and $\hat{R}^{\mathcal{S}}$ but still uses negations. The update formulas for relation symbols $R \in \tau$ are as in $\mathcal{P}$. For every $\hat{R} \in \hat{\tau}$ and every modification δ, the update formula $\phi_\delta^{\hat{R}}(\bar{x}, \bar{y})$ is

the negation normal form of $\neg\phi_\delta^R(\bar{x}, \bar{y})$. The relation $\hat{=}$ does not change after initialization. The initialization mapping $\textsc{Init}'$ initializes $\hat{R}$ with the complement of R.

We can now construct a negation-free DynFO-program $\mathcal{P}'' = (P'', \textsc{Init}', Q)$ from $\mathcal{P}'$ in the following way: an update formula $\phi_\delta^R(\bar{x}; \bar{y})$ for $\mathcal{P}''$ is obtained from the update formula $\phi_\delta^R(\bar{x}; \bar{y})$ for $\mathcal{P}'$ by replacing all negative literals $\neg S$ by $\hat{S}$. The initialization mapping of $\mathcal{P}''$ is the same as that of $\mathcal{P}'$. The equivalence of $\mathcal{P}$ and $\mathcal{P}''$can be proved by induction over the length of modification sequences.

□

Definition 3.4. If $\mathbb{Q}$ is a quantifier prefix, $\bar{\mathbb{Q}}$ is the quantifier prefix that is obtained by replacing every instance of $\exists$ in $\mathbb{Q}$ by $\forall$ and every instance of $\forall$ by $\exists$.

Lemma 3.5. *Let $\mathbb{Q}$ be an arbitrary quantifier prefix. A query can be maintained in* $\textsc{Dyn}\mathbb{Q}$*FO if and only if it can be maintained in* $\textsc{Dyn}\bar{\mathbb{Q}}$*FO.*

Proof. Let $\mathcal{P} = (P, \textsc{Init}, Q)$ be an arbitrary dynamic $\textsc{Dyn}\mathbb{Q}$FO-program over the schema τ. By Lemma 3.2 we can assume without loss of generality that the update formulas of Q are atomic. We will construct a dynamic $\textsc{Dyn}\bar{\mathbb{Q}}$FO-program $\mathcal{P}'$ over the schema $\hat{\tau} \cup \{Q'\}$ where $\hat{\tau}$ contains a k-ary relation symbol $\hat{R}$ for every k-ary $R \in \tau$. The intention is that $\hat{R}$ is always equal to the complement of R. This is achieved in a similar way as in the proof above.

We denote by $\phi[\tau \to \hat{\tau}]$ the formula obtained from ϕ by replacing every atom $S(\bar{z})$ in ϕ by $\neg\hat{S}(\bar{z})$. Then the update formulas of $\mathcal{P}'$ are obtained as $\phi_\delta^{\hat{R}} \stackrel{\text{def}}{=} \neg\phi_\delta^R[\tau \to \hat{\tau}]$ for every $\hat{R} \in \hat{\tau}$. Observe that this formula can easily be transformed into an $\bar{\mathbb{Q}}$FO-formula. Further $\phi_\delta^{Q'} = \phi_\alpha^{\hat{Q}}$ which is a $\bar{\mathbb{Q}}$FO-formula since $\phi_\alpha^{\hat{Q}}$ is quantifier-free. The initialization mapping of $\mathcal{P}'$ is straightforward. □

The following lemma uses the replacement technique to remove disjunctions from quantifier-free programs.

Lemma 3.6. *Every* $\textsc{DynProp}$*-program has an equivalent* $\textsc{DynPropCQ}^{\neg}$*-program*

Proof. Let $\mathcal{P} = (P, \textsc{Init}, Q)$ be a $\textsc{DynProp}$-program over the schema τ. We will assume without loss of generality that τ contains, for every relation symbol R a relation symbol $\hat{R}$ and that $\mathcal{P}$ ensures that $\hat{R}^{\mathcal{S}}$ is the complement of $R^{\mathcal{S}}$ for every state $\mathcal{S}$. This can be achieved with the technique used in Lemma 3.3. Further, we will assume that all update formulas of $\mathcal{P}$ are in conjunctive normal form.

The conjunctive $\textsc{DynProp}$-program we are going to construct will use the schema $\tau \cup \tau'$, where τ' contains a fresh relation symbol $R_{\neg C}$ for every clause C occuring in any update formula of $\mathcal{P}$. The goal of the construction is to ensure

that $R^{\mathcal{S}}_{\neg C}(\bar{z})$ holds if and only if $\neg C(\bar{z})$ is true in the state $\mathcal{S}$. Now update formulas $\phi = C_1(\bar{x}_1) \wedge ... \wedge C_k(\bar{x}_k)$ with clauses $C_1(\bar{x}_1), ..., C_k(\bar{x}_k)$ can be replaced by the conjunctive formula $\neg R_{\neg C_1}(\bar{x}_1) \wedge ... \wedge \neg R_{\neg C_k}(\bar{x}_k)$.

We start by constructing a DynProp-program $\mathcal{P}' = (P', \text{Init}, Q)$ in conjunctive normal form, that maintains the relations $R^{\mathcal{S}}_{\neg C}$. To this end, let C be a clause with k variables and let $\bar{z}$ be the k-tuple that contains the variables of C in the order in which they occur. Assume that $C = L_1(\bar{z}_1) \vee ... \vee L_l(\bar{z}_l)$. The relation symbol $R_{\neg C}$ is of arity k. For a modification δ, the update formula for $R_{\neg C}$ is

$$\phi^{R_{\neg C}}_{\delta}(\bar{x};\bar{z}) = \phi^{X_1}_{\delta}(\bar{x};\bar{z}_1)^{... \phi^{X_l}_{\delta}}(\bar{x};\bar{z}_l)$$

where X_i is the relation symbol R if $L_i = \neg R$ and $\hat{R}$ if $L_i = R$. Observe that $\phi^{R_{\neg}C}_{\delta}(\bar{x}, \bar{z})$ is in conjunctive normal form since each $\phi^{X_i}_{\delta}(\bar{x}, \bar{z}_i)$ is in conjunctive normal form. Further, $\phi^{R_{\neg C}}_{\delta}(\bar{x};\bar{z})$ does not use new clauses. The initialization mapping Init' extends the initialization mapping Init to the schema τ' in the following way. For a clause C and an input database $\mathcal{I}$, a tuple $\bar{a}$ is in $\text{Init}'(R_{\neg C})$ if and only if C evaluates to false in $\text{Init}(\mathcal{I})$ for $\bar{a}$.

We will then construct the desired conjunctive DynProp-program P'' by replacing every clause C in every update formula of $\mathcal{P}'$ by $\neg R_{\neg C}$. The initialization mapping for P'' is the same as that of P'.

Since $\mathcal{P}'$ updates relations from τ exactly as $\mathcal{P}$ does, and we can demonstrate by induction over the length of modification sequences that for all tuples $\bar{a}$, $R^{\mathcal{S}}_{\neg C}(\bar{a})$ holds if and only if $\neg C(\bar{a})$ is true in state $\mathcal{S}$, the corresponding formulas from $\mathcal{P}$ and $\mathcal{P}''$ always yield the same results. □

Definition 3.7. A DynFO$^{\wedge}$-program is a dynamic program with update formulas that are in prenex normal form where the quantifier-free part is a conjunction of atoms.

The following lemma will be needed for Lemma 3.9. The proof will not be covered here but it can be found in [10] under Lemma 3.1.13.

Lemma 3.8. *Let $k \geq 0$. For every k-ary DynUCQ-program $\mathcal{P}$ there is a k'-ary DynUCQ-program $\mathcal{P}'$ with $k' \overset{def}{=} max\{k, 2\}$ and with a query symbol Q' which is equivalent for domains of size at least 2 and satisfies*

1. *in every possible state all auxiliary relations $\neq Q'$ of $\mathcal{P}'$ are neither empty nor do they contain all tuples and*
2. *no update formula uses the relation symbol $\mathcal{Q}'$.*

Analogously for $DynUCQ^{\neg}$- and negation-free DynFO-programs.

Lemma 3.9.

(a) For every DYNUCQ$^{\neg}$*-program there is an equivalent* DYNCQ$^{\neg} - program$

(b) For every DYNUCQ*-program there is an equivalent* DYNCQ*-program*

(c) For every DYNFO*-program there is an equivalent* DYNFO$^{\wedge}$*-program*

Proof. We will only prove the statement for domains with at least two elements. To see how this restriction can be lifted, see the proof of Lemma 3.1.14 from [10].

The construction will be presented for (a), but since it does not introduce any negation operators, it also works for (b). For (c), we can assume by Lemma 3.3 that we start from a negation-free DYNFO-program and use the construction from (a) on that, the quantifier-prefix $\exists\bar{y}$ just needs to be substituted by an arbitrary quantifier-prefix.

Let $\mathcal{P} = (P, \textsc{Init}, Q)$ be a DYNUCQ$^{\neg}$-program over the schema τ. Since we assume the domain is of size at least 2, due to Lemma 3.8 we can assume that all auxiliary relations of $\mathcal{P}$ except for $\mathcal{Q}$ are not empty, do not contain all tuples and that $\mathcal{Q}$ is not used in any update formula. Further, without loss of generality, we will assume that the quantifier-free parts of all update formulas of $\mathcal{P}$ are in disjunctive normal form.

We will convert $\mathcal{P}$ into an equivalent DYNCQ$^{\neg}$-program $\mathcal{P}'$ with update formulas in prenex normal form and quantifier-free parts of the form $T(\bar{w}) \wedge \bigwedge_i L_i(\bar{w}_i)$, where L_i is an arbitrary literal over τ for all i and the symbols T are fresh auxiliary relation symbols. Now let $\mathcal{P}' = (P', \textsc{Init}, Q)$ be a program over the schema $\tau' = \tau \cup \tau_T$, where τ_T contains a relation symbol $T_{R,\delta}$ for every relation symbol $R \in \tau$ and every modification δ. The intent is that corresponding states for $\mathcal{P}$ and $\mathcal{P}'$ agree on the relations from τ.

Now we will construct the update formulas for the program $\mathcal{P}'$. Let $R \in \tau$ and let δ be a modification. Further, let

$$\phi^R_\delta(\bar{u};\bar{x}) = \exists\bar{y}(C_1(\bar{u},\bar{x},\bar{y}) \vee ... \vee C_k(\bar{u},\bar{x},\bar{y}))$$

be the update formula of R with respect to δ in $\mathcal{P}$, where every C_i is a conjunction of literals.

For every

$$C_i(\bar{u},\bar{x},\bar{y}) = L^1_i(\bar{v}^1_i) \wedge ... \wedge L^l_i(\bar{v}^l_i)$$

we define

$$\hat{C}_i(\bar{z}_i) \stackrel{\text{def}}{=} L^1_i(\bar{z}^1_i) \wedge ... \wedge L^l_i(\bar{z}^l_i)$$

where all $\bar{z}_i^j$ contain pairwise different, fresh variables and $\bar{z}_i \stackrel{\text{def}}{=} (\bar{z}_i^1, ..., \bar{z}_i^l)$. Also, let $\bar{v}_i \stackrel{\text{def}}{=} (\bar{v}_i^1, ..., \bar{v}_i^l)$ and let X be the set of variables appearing in $\bar{u}$, $\bar{x}$, $\bar{y}$ and in the tuples $\bar{z}_i$.

The update formula $\psi_\delta^R(\bar{u}, \bar{x})$ for $R \in \tau$ in $\mathcal{P}'$ is as follows:

$$\psi_\delta^R(\bar{u}; \bar{x}) \stackrel{\text{def}}{=} \exists \bar{y} \exists \bar{z}_1 ... \exists \bar{z}_k (\hat{C}_1(\bar{z}_1) \land ... \land \hat{C}_k(\bar{z}_k) \land T_{R,\delta}(\bar{u}, \bar{x}, \bar{y}, \bar{z}_1, ..., \bar{z}_k))$$

The relations $T_{R,\delta}$ are fixed, that is, their update formulas always reproduce the current value of $T_{R,\delta}$. A tuple $\bar{a}$ is in the relation $T_{R,\delta}$ if there is an assignment $\pi : X \to D$ with $\bar{a} = \pi(\bar{u}, \bar{x}, \bar{y}, \bar{z}_1, ..., \bar{z}_k)$ and $\pi(\bar{z}_i) = \pi(\bar{v}_i)$ for some i. Here, the tuple $\bar{v}_i$ consists of elements from $\bar{u}$, $\bar{x}$ and $\bar{y}$ as specified by the definition of $\bar{v}_i$ above.

Auxiliary relation symbols $R \in \tau$ are initialized as in $\mathcal{P}$. The relations $T_{R,\delta}$ are initialized as intended by simple quantifier-free formulas (but with a disjunction for selecting i).

We will now outline the proof of why $\mathcal{P}$ and $\mathcal{P}'$ are equivalent. The proof is by induction over the length of modification sequences. It is sufficient to show that the formulas ϕ_δ^R and ψ_δ^R yield the same result for states $\mathcal{S}$ and $\mathcal{S}'$, where $\mathcal{S}'$ contains the relation $T_{R,\delta}$ in addition to the relations of $\mathcal{S}$.

If $(S, \bar{a}, \bar{b}) \models \phi_\delta^R(\bar{u}; \bar{x})$ then there is a tuple $\bar{c}$ so that $(\mathcal{S}, \bar{a}, \bar{b}, \bar{c}) \models C_i(\bar{u}, \bar{x}, \bar{y})$ for some i. Now to show that $(\mathcal{S}', \bar{a}, \bar{b}) \models \psi_\delta^R(\bar{u}, \bar{x})$ one can choose $\bar{y}$ in ψ_δ^R as $\bar{c}$ and the values for $\bar{z}_i$ accordingly. This will satisfy $\hat{C}_i(\bar{z}_i)$ and $T_{R,\delta}$. The values for each $\bar{z}_j$ with $j \neq i$ are chosen so that all literals in $\hat{C}_j(\bar{z}_j)$ are satisfied, which is possible because all auxiliary relations are neither empty nor do they contain all tuples.

If $(\mathcal{S}', \bar{a}, \bar{b}) \models \psi_\delta^R(\bar{u}; \bar{x})$ then there are tuples $\bar{c}$ and $\bar{d}_1, ..., \bar{d}_k$ so that $(\mathcal{S}', \bar{d}_i) \models \hat{C}_i(\bar{z}_i)$ for some i and $(\mathcal{S}', \bar{a}, \bar{b}, \bar{c}, \bar{d}_1, ..., \bar{d}_k) \models T_{R,\delta}(\bar{u}, \bar{x}, \bar{y}, \bar{z}_1, ..., \bar{z}_k)$. But then, due to the definition of $T_{R,\delta}$, there is a tuple $\bar{c}''$ so that $(\mathcal{S}, \bar{c}') \models C_i(\bar{v}_i)$. Therefore also $(\mathcal{S}, \bar{a}, \bar{b}) \models \phi_\delta^R(\bar{u}; \bar{x})$. □

From the lemmas 3.3, 3.5, 3.6 and 3.9 we get the following four theorems:

Theorem 3.10. *Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:*

(a) $\mathcal{Q}$ can be maintained in DynUCQ$^\neg$.

(b) $\mathcal{Q}$ can be maintained in DynCQ$^\neg$.

(c) $\mathcal{Q}$ can be maintained in Dyn$\exists^*$FO.

(d) $\mathcal{Q}$ can be maintained in Dyn$\forall^*$FO.

Proof.

(a) ⇔ (b): follows from Lemma 3.9 and the fact that every DYNCQ$^{\neg}$-program is a DYNUCQ$^{\neg}$-program.

(c) ⇔ (d): follows from Lemma 3.5.

(a) ⇔ (c): follows from the definition of DYNUCQ$^{\neg}$ and DYN$\exists^*$FO.

□

Theorem 3.11. *Let Q be a query. Then the following statements are equivalent:*

(a) Q can be maintained in DYNUCQ.

(b) Q can be maintained in DYNCQ.

Proof. Follows from Lemma 3.9 and the fact that every DYNCQ-program is a DYNUCQ-program. □

Theorem 3.12. *Let Q be a query. Then the following statements are equivalent:*

(a) Q can be maintained in DYNFO.

(b) Q can be maintained in DYNFO$^{\wedge}$.

Proof. **(a) ⇒ b:** by definition.

(b) ⇒ (a): follows from Lemma 3.9.

□

Theorem 3.13. *Let Q be a query. Then the following statements are equivalent:*

(a) Q can be maintained in DYNPROP.

(b) Q can be maintained in DYNPROPUCQ$^{\neg}$.

(c) Q can be maintained in DYNPROPCQ$^{\neg}$.

(d) Q can be maintained in DYNPROPUCQ.

Proof.

(a) ⇒ (c): follows from Lemma 3.6.

(c) ⇒ (b): follows from the definitions of DYNPROPCQ$^{\neg}$ and DYNPROPUCQ$^{\neg}$

(a) ⇒ (d): follows from Lemma 3.3.

(b), (c), (d) ⇒ (a): follow from the definitions.

□

Theorem 3.14.

(a) The class DYNPROPCQ *is a strict subclass of* DYNPROP*.*

(b) The class DYNPROP *is a strict subclass of* DYNCQ*.*

Proof. Part (a) follows from the of the fact that s-t-REACH cannot be maintained in DYNPROPCQ (Theorem 4.1.14 in [10]) and part (b) follows from the next theorem and the fact that DYNQF can express the equal cardinality query while DYNPROP cannot [13]. □

Theorem 3.15. *DYNQF is contained in DYNCQ.*

Proof. Let $\mathcal{P} = (P, \text{INIT}, Q)$ be a dynamic DYNQF-program over the schema $\tau = \tau_{\text{rel}} \cup \tau_{\text{fun}}$. As in Lemma 3.3, we will assume, without loss of generality that $\mathcal{P}$ is in negation normal form and that input relations also have update formulas.

We will now prove that there is a DYNUCQ-program $\mathcal{P}''$ that is equivalent to $\mathcal{P}$. Then, by Lemma 3.9, there is an equivalent DYNCQ-program.

As a preparatory step, we will construct, from $\mathcal{P}$, a DYNQF-program $\mathcal{P}'$ over the schema $\tau' \stackrel{\text{def}}{=} \tau_{\text{rel}} \cup \hat{\tau}_{\text{rel}} \cup \{\hat{=}\} \cup \tau_{\text{fun}}$ where $\hat{\tau}_{\text{rel}}$ contains, for every $R \in \tau$, a relation symbol $\hat{R}$ intended to contain the complement of R and $\hat{=}$ that contains the complement of the relation $=$, as in Lemma 3.3.

From $\mathcal{P}'$ we construct a DYNUCQ-program $\mathcal{P}''$ over the schema $\tau'' \stackrel{\text{def}}{=} \tau_{\text{rel}} \cup \hat{\tau}_{\text{rel}} \cup \tau_F$, where τ_F contains a $k{+}1$-ary relation symbol R_f for every k-ary function symbol $f \in \tau_{\text{fun}}$. The intention is that R_f simulates f in the sense that $(\bar{a}, b) \in R_f^{\mathcal{S}'}$ if and only if $f^{\mathcal{S}''}(\bar{a}) = b$ in states $\mathcal{S}'$ and $\mathcal{S}''$ reached in $\mathcal{P}'$ and $\mathcal{P}''$ reached by the same modification sequence. The initialization of R_f can easily be obtained from the initialization of f.

We say that two states $\mathcal{S}'$ and $\mathcal{S}''$ over τ' and τ'' *correspond*, if

1. $(\bar{a}, b) \in R^{\mathcal{S}'}$ if and only if $f^{\mathcal{S}''}(\bar{a}) = b$, and

2. $R^{\mathcal{S}'} = R^{\mathcal{S}''}$ for all $R \in \tau_{\text{rel}} \cup \widehat{\tau_{\text{rel}}}$.

We will now explain how to update relations from τ_F. To this end, we will define CQ-formulas $\varphi_t(\bar{x}, z)$ and $\varphi_\phi(\bar{x})$ over τ'' for every update term $t(\bar{x})$ and every update formula $\phi(\bar{x})$ over τ so that the following conditions are satisfied for all corresponding states $\mathcal{S}'$, $\mathcal{S}''$, all tuples $\bar{a}$ and all elements b:

- $(\mathcal{S}'', \bar{a}, b) \vDash \varphi_t(\bar{x}, z)$ if and only if $t^{\mathcal{S}'}(\bar{a}) = b$, and
- $(\mathcal{S}'', \bar{a}) \vDash \varphi_\phi(\bar{x})$ if and only if $(\mathcal{S}', \bar{a}) \vDash \phi(\bar{x})$

Then the update formulas in $\mathcal{P}''$ after a modification δ can be defined as follows. For every $R_f \in \tau_F$, define the update formula as $\phi_\delta^{R_f} \stackrel{\text{def}}{=} \varphi_t$ where t is the update term for $f \in \tau_{\text{fun}}$ in $\mathcal{P}'$. For every $R \in \tau_{\text{rel}} \cup \hat{\tau}_{\text{rel}}$define the update formula as $\phi_\delta^R \stackrel{\text{def}}{=} \varphi_\phi$ where ϕ is the update formula of R in $\mathcal{P}'$. An easy induction shows that $\mathcal{P}'$ and $\mathcal{P}''$ yield corresponding states when the same modification sequence is applied. This proves the claim.

It remains to define the CQ-formulas $\varphi_t(\bar{x}, z)$ and $\varphi_\phi(\bar{x})$ for every update term $t(\bar{x})$ and every formula $\phi(\bar{x})$. Those formulas are defined inductively as follows:

1. If $t(\bar{x}) = y$ for some variable y occuring in $\bar{x}$, then

$$\varphi_t(\bar{x}, z) \stackrel{\text{def}}{=} y = z$$

2. If $t(\bar{x}) = f(t_1(\bar{x}_1), ..., t_k(\bar{x}_k))$ with $\bar{x}_i \subseteq \bar{x}$, then

$$\varphi_t(\bar{x}, z) \stackrel{\text{def}}{=} \exists z_1 ... z_k (R_f(z_1, ..., z_k, z) \wedge \bigwedge_i \varphi_{t_i}(\bar{x}, z_i))$$

3. If $t(\bar{x}) = \text{ITE}(\phi(\bar{y}), t_1(\bar{x_1}), t_2(\bar{x}_2))$ with $\bar{y}, \bar{x_1}, \bar{x_2} \subseteq \bar{x}$, a quantifier-free update formula ϕ and update terms t_1, t_2, then

$$\varphi_t(\bar{x}, z) \stackrel{\text{def}}{=} (\varphi_\phi(\bar{y}) \wedge \varphi_{t_i}(\bar{x}_1, z)) \vee (\varphi_{\neg\phi}(\bar{y}) \wedge \varphi_{t2}(\bar{x}_2, z))$$

4. If $\phi(\bar{x})$ contains the maximal update terms $t_1(\bar{x}_1), ..., t_k(\bar{x}_k)$ then let

$$\varphi_t(\bar{x}) \stackrel{\text{def}}{=} \exists z_1 ... \exists z_k (\phi' \bigwedge_i \varphi_{t_i}(\bar{x}_i, z_i))$$

   where ϕ' is obtained from ϕ by replacing t_i by z_i, transforming the resulting formula into negation normal form and then replacing every literal of the form $\neg R(s_1, ..., s_l)$ by $\hat{R}(s_1, ..., s_l)$. Here, a term t_i is maximal if it is not contained in another update term.

Observe that the formula ϕ in 4. contains only relation symbols from $\tau_{\text{rel}} \cup \hat{\tau}_{\text{rel}}$, and therefore no symbols from τ_{fun} need to be replaced in ϕ'. The correctness of this construction can be proved inductively.

□

With theorems 3.10, 3.11, 3.12, 3.13, 3.14 and 3.15 we have now covered the left side of Figure 4.

3.3 Δ-semantics

We will now consider an alternative semantics to dynamic programs, called the Δ-semantics. Commonly, when updating the input relation, only some of the tuples in the auxiliary relations will change. It therefore makes sense to express the updated version of relations as the previous version with some elements added and some elements removed.

We will now give a formal definition of Δ-semantics and compare the expressive power of Δ-semantics with different logics for update formulas against each other and their counterparts with absolute semantics.

Definition 3.16 (Δ-Update program)**.** A Δ-update program $\mathcal{P}$ over the dynamic schema $(\tau_{\text{inp}}, \tau_{\text{aux}})$ is a set of first-order formulas (called Δ-update formulas in the following) that contains, for every $R \in \tau_{\text{aux}}$ and every $\delta \in \{\text{INS}_S, \text{DEL}_S\}$ with $S \in \tau_{\text{inp}}$ two formulas $\phi_\delta^{R^+}(\bar{u}; \bar{x})$ and $\phi_\delta^{R^-}(\bar{u}; \bar{x})$ over the schema τ where $\bar{u}$ and S have the same arity, $\bar{x}$ and R have the same arity, and $\phi_\delta^{R^+} \wedge \phi_\delta^{R^-}$ is unsatisfiable.

For a modification $\delta = \delta(\bar{a})$ and a program state $\mathcal{S} = (D, \mathcal{I}, \mathcal{A})$ we denote by $P_\delta(\mathcal{S})$ the state $(D, \delta(\mathcal{I}), \mathcal{A}')$ where the relations R' of $\mathcal{A}'$ are defined by

$$R' \stackrel{\text{def}}{=} (R \cup \{\bar{b} | \mathcal{S} \vDash \phi_\delta^{R^+}(\bar{a}; \bar{b})\}) \backslash \{\bar{b} | \mathcal{S} \vDash \phi_\delta^{R^-}(\bar{a}; \bar{b})\}.$$

The effect of a modification sequence on a state, dynamic Δ-programs and so on are defined like their counterparts in absolute semantics except for using Δ-update programs instead of update programs.

Definition 3.17 (Δ-DYN$\mathcal{C}$)**.** For a class $\mathcal{C}$ of formulas, let Δ-DYN$\mathcal{C}$ be the class of all dynamic queries that can be maintained by dynamic Δ-programs with formulas from $\mathcal{C}$ and arbitrary initialization mappings.

Lemma 3.18. *Let Δ-DYN$\mathcal{C}$ be one of the dynamic complexity classes Δ-DYNPROPCQ$^\neg$, Δ-DYNCQ$^\neg$, Δ-DYNUCQ$^\neg$ or Δ-DYN$\mathbb{Q}$FO for an arbitrary quantifier $\mathbb{Q}$. If a query Q can be maintained in Δ-DYN$\mathcal{C}$ then Q can be maintained in negation-free Δ-DYN$\mathcal{C}$.*

Proof. Given a dynamic Δ-program $\mathcal{P}$ over the schema τ, we will construct a dynamic Δ-program $\mathcal{P}'$ over the schema $\tau \cup \hat{\tau}$, where $\hat{\tau}$ again contains, for every k-ary relation symbol $R \in \tau$, a fresh k-ary relation symbol $\hat{R}$ with the intent that $\hat{R}$ always stores the complement of R.

The update formulas for $R \in \tau$ are as in $\mathcal{P}$. For a relation symbol $R \in \tau$, let $\phi_\delta^{R^+}(\bar{u}; \bar{x})$ and $\phi_\delta^{R^-}(\bar{u}; \bar{x})$ be the update formulas for R. Now the update formulas from $\hat{R}$ can be defined in the following way:

$$\phi_\delta^{\hat{R}^+}(\bar{u};\bar{x}) \stackrel{\text{def}}{=} \phi_\delta^{R^-}(\bar{u};\bar{x})$$
$$\phi_\delta^{\hat{R}^-}(\bar{u};\bar{x}) \stackrel{\text{def}}{=} \phi_\delta^{R^+}(\bar{u};\bar{x})$$

As $\phi_\delta^{R^+} \wedge \phi_\delta^{R^-}$ is unsatisfiable, all tuples will be contained either in R or $\widehat{R}$. From $\mathcal{P}'$, a negation-free dynamic Δ-program $\mathcal{P}''$ can be constructed by replacing all occurences of $\neg R(\bar{x})$ in it's update formulas by $\widehat{R}(\bar{x})$. □

The following lemma will not be proved here. The proof can be found in Lemma 3.2.8 in [10].

Lemma 3.19. *Let $\mathbb{Q}$ be an arbitrary quantifier prefix. If a query can be maintained in* Dyn$\mathbb{Q}$FO*, then it can be maintained in* Δ-Dyn$\mathbb{Q}$FO *as well.*

Lemma 3.20.

(a) If a query can be maintained in Δ-DynUCQ$^\neg$*, then it can be maintained in* DynUCQ$^\neg$ *as well.*

(b) If a query can be maintained in Δ-Dyn$\forall^*$FO*, then it can be maintained in* Dyn$\forall^*$FO *as well.*

Proof. We will only prove (a), the proof for (b) follows the same construction. Let $\mathcal{P} = (P, \textsc{Init}, \mathcal{Q})$ be a dynamic Δ-DynUCQ$^\neg$-program over the schema τ. By Lemma 3.18 we can assume without loss of generality that the update formulas of $\mathcal{P}$ are negation-free. For ease of presentation we will assume that the input schema is E, where E is a binary relation.

We will construct an equivalent DynUCQ$^\neg$-program $\mathcal{P}'$ in the following way.

Consider some update formulas $\phi_\delta^{R^+}(\bar{u};\bar{x})$ and $\phi_\delta^{R^-}(\bar{u};\bar{x})$ of a relation $R \in \tau$ for a modification δ in $\mathcal{P}$. The naïve translation into a DynFO-update formula $\phi_\delta^{R}(\bar{u};\bar{x})$ would yield

$$\phi_\delta^{R}(\bar{u},\bar{x}) = (R(\bar{x}) \vee \phi_\delta^{R^+}(\bar{u};\bar{x})) \wedge \neg\phi_\delta^{R^-}(\bar{u};\bar{x})$$

which is not necessarily in UCQ$^\neg$ since $\neg\phi_\delta^{R^-}(\bar{u};\bar{x})$ is not in UCQ$^\neg$ if $\phi_\delta^{R^-}(\bar{u};\bar{x})$ is not atomic. Therefore, $\mathcal{P}'$ will maintain a relation R_δ^- that contains all tuples $(\bar{a},\bar{b})$ so that $\bar{a}$ would be removed from R after applying the modification $\delta(\bar{b})$. Those relations are maintained using the squirrel technique.

The dynamic program $\mathcal{P}'$ is over the schema $\tau \cup \tau_\Delta$ where τ_Δ contains a $k+2$-ary relation symbol $R_\delta^- \in \tau$ for every k-ary relation symbol $R \in \tau$ and every modification $\delta \in \{\textsc{ins}, \textsc{del}\}$ of the input relation E.

The update formula for a relation symbol $R \in \tau$ is

$$\phi_\delta^R(\bar{u}, \bar{x}) \stackrel{\text{def}}{=} (R(\bar{x}) \vee \phi_\delta^{R^+}(\bar{u}, \bar{x})) \wedge R_\delta^-(\bar{u}; \bar{x}).$$

This formula can be translated into an existential formula in a straightforward manner.

For updating a relation $R_{\delta_1}^-$ after a modification δ_0, the update formula $\phi_{\delta_1}^{R^-}$ for R^- is used. However, since $R_{\delta_1}^-$ will store tuples that have to be deleted after applying δ_1, the formula $\phi^{R_{\delta_1}^-}$ has to be adapted to use the content of relational symbols $S \in \tau$ after modification δ_0.

For this purpose, relation symbols $S \in \tau$ in $\phi^{R^-}{}_{\delta_1}$ need to be replaced by their update formulas as defined above.

The update formula for $R_{\delta_1}^-$ is

$$\phi_{\delta_1}^{R_{\delta_1}^-}(\bar{u}_0; \bar{u}_1, \bar{x}) \stackrel{\text{def}}{=} \phi_{\delta_1}^{R_{\delta_1}^-}[\tau \to \phi^\tau](\bar{u}_0; \bar{u}_1, \bar{x})$$

where $\phi_{\delta_1}^{R_{\delta_1}^-}[\tau \to \phi^\tau](\bar{u}_0; \bar{u}_1, \bar{x})$ is obtained from $\phi_{\delta_1}^{R_{\delta_1}^-}$ by replacing every atom $S(\bar{z})$ by $\phi_{\delta_0}^S(\bar{u}_0, \bar{z})$, as constructed above. Since by our initial assumption, $\phi_{\delta_1}^{R^-}$ itself is an UCQ-formula and all update formulas $\phi_{\delta_0}^S$ for $S \in \tau$ are UCQ-formulas, the formula $\phi_{\delta_1}^{R_{\delta_1}^-}$ can easily be converted into an UCQ-formula as well. □

The following lemma will not be proved here. The proof can be found in Lemma 3.2.11 in [10].

Lemma 3.21.

(a) For every Δ-DynUCQ$^\neg$-program there is an equivalent Δ-DynUCQ-program.

(b) For every Δ-DynFO-program, there is an equivalent Δ-DynFO$^\wedge$-program.

Theorem 3.22. *Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:*

(a) $\mathcal{Q}$ can be maintained in Δ-DynUCQ$^\neg$.

(b) $\mathcal{Q}$ can be maintained in Δ-DynUCQ.

(c) $\mathcal{Q}$ can be maintained in Δ-DynCQ$^\neg$.

(d) $\mathcal{Q}$ can be maintained in Δ-DynCQ.

(e) $\mathcal{Q}$ can be maintained in Δ-Dyn$\exists^*$FO.

*(f) $\mathcal{Q}$ can be maintained in Δ-DynForall*FO.*

Proof.

(a) ⇔ (b) and (c) ⇔ (d): follow from Lemma 3.18.

(a) ⇔ (c): follows from Lemma 3.21.

(a) ⇔ (e): (a) and (e) are equivalent by definition.

(e) ⇔ (f): follows by combining Lemmas 3.19 and 3.20 with Theorem 3.10.

□

Theorem 3.23. *Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:*

(a) $\mathcal{Q}$ can be maintained in $\text{DynUCQ}^{\neg}$.

(b) $\mathcal{Q}$ can be maintained in $\Delta\text{-DynUCQ}^{\neg}$.

Proof.

(a) ⇒ (b): follows from Lemma 3.19.

(b) ⇒ (a): follows from Lemma 3.20.

□

Theorem 3.24. *Let $\mathcal{Q}$ be a query. Then the following statements are equivalent:*

(a) $\mathcal{Q}$ can be maintained in Δ-DynFO.

(b) $\mathcal{Q}$ can be maintained in $\Delta\text{-DynFO}^{\wedge}$.

Proof. **(a) ⇒ (b):** Follows from Lemma 3.21.

(b) ⇒ (a): is by definition.

□

3.4 Relating Dynamic Complexity Classes with Static Complexity Classes

In this section we are interested in finding cases where we can maintain all queries from a static descriptive complexity class using updates from a weaker class.

Prior to [10], the only known such cases were that MSO-queries on strings can be maintained in DynPROP and that on the general structures, $\exists^*$FO is captured by DynQF [13].

Zeume presents two further results in [10]. The first result if that all first-order definable queries are maintainable using conjunctive queries with negations as update formulas.

The second result presented by Zeume is that when restricting modifications to be insertions only, queries definable by unions of conjunctive queries with negated equality atoms can be maintained in DynProp.

Definition 3.25. The *dependency graph* of a dynamic program P with the auxiliary schema τ_{aux} is a graph with the vertex set $V = \tau_{\text{aux}}$ and all edges (R, R') where R' occurs in one of the update formulas for R.

Definition 3.26. A *non-recursive dynamic program* is a dynamic program with an acyclic dependency graph.

For every class $\mathcal{C}$, *non-recursive* Dyn$\mathcal{C}$ refers to the set of queries that can be maintained by a non-recursive Dyn$\mathcal{C}$ program.

We say that a formula φ is in *existential prefix form* if it has a prefix of $((\neg\exists)|\exists)^*$ and no quantifiers after this prefix. As all FO formulas have an equivalent formula in prenex normal form, and universal quantifiers can be replaced by existential quantifiers and negations, all FO formulas have an equivalent formula in existential prefix form.

Lemma 3.27. *If a query is definable in FO, then it can be maintained in non-recursive Dyn$\exists^1$FO.*

Proof. Let φ be a FO formula in existential prefix form. We will prove by induction over the length of the prefix of φ that, for every finite sequence $\delta_1, ..., \delta_j$, the query defined by $\varphi_{\delta_1 ... \delta_j}$ is maintainable in non-recursive Dyn$\exists^1$FO. The claim follows by setting $j = 0$. We construct dynamic programs where the result of the query defined by $\varphi_{\delta_1 ... \delta_j}$ is stored in the relation $R^{\varphi}_{\delta_1 ... \delta_j}$.

For a formula φ with a prefix of length 0, we define

$$\phi_{\delta_0}^{R^{\varphi}_{\delta_1 ... \delta_j}}(\bar{v}_0; \bar{y}, \bar{v}_1, ..., \bar{v}_j) \stackrel{\text{def}}{=} \varphi^{E}_{\delta_0 ... \delta j}(\bar{y}, \bar{v}_1, ..., \bar{v}_j)$$

where $\varphi^E_{\delta_0 \ldots \delta j}$ is as defined above.

For the induction step, let φ be a formula of prefix length i. By induction hypothesis, every query defined by $\psi_{\delta_1 \ldots \delta_j}$ where ψ has prefix length $i-1$ can be maintained in non-recursive DYN$\exists^1$FO for every sequence $\delta_1 \ldots \delta_j$ of modifications.

We distinguish the two cases $\varphi(\bar{y}) = \exists x \psi(x, \bar{y})$ and $\varphi(\bar{y}) = \neg\gamma(\bar{y})$. If $\varphi(\bar{y}) = \exists x \psi(x, \bar{y})$ then the dynamic program for φ and $\delta_1 \ldots \delta_j$ has auxiliary relations $R^\psi_{\delta_0 \ldots \delta_j}$ for $\delta \in$ INS, DEL containing the result of the query $\psi_{\delta_0 \ldots \delta_j}$. Further

$$\phi_{\delta_0}^{R^\varphi_{\delta_1 \ldots \delta_j}}(\bar{v}_0; \bar{y}, \bar{v}_1, \ldots, \bar{v}_j) \stackrel{\text{def}}{=} \exists x R^\psi_{\delta_0 \ldots \delta_j}(x, \bar{y}, \bar{v}_0, \ldots, \bar{v}_j).$$

If $\varphi(\bar{y}) = \neg\gamma(\bar{y})$, then the dynamic program for δ and $\delta_1 \ldots \delta_j$ has auxiliary relations $R^\gamma_{\delta_0 \ldots \delta_j}$ for $\delta_0 \in$ INS, DEL containing the result of the query $\gamma_{\delta_0 \ldots \delta_j}$. Further,

$$\phi_{\delta_0}^{R^\varphi_{\delta_1 \ldots \delta_j}}(\bar{v}_0; \bar{y}, \bar{v}_1, \ldots, \bar{v}_j) \stackrel{\text{def}}{=} \neg R^\gamma_{\delta_0 \ldots \delta_j}(\bar{y}, \bar{v}_0, \ldots, \bar{v}_j).$$

This yields a non-recursive $\exists^1$FO-program for every $\varphi_{\delta_1 \ldots \delta j}$. □

Lemma 3.28. *If a query can be maintained in non-recursive DYNFO, then it can be expressed in FO.*

Proof. Let $\mathcal{Q}$ be a query which can be maintained by a non-recursive DYNFO program $\mathcal{P} = (P, \text{INIT}, Q)$ over the schema $\tau = \tau_{\text{inp}} \cup \tau_{\text{aux}}$. For simplicity, we will assume that $\tau_{\text{inp}} = E$, for a binary symbol E. We let $R_0 \stackrel{\text{def}}{=} E$ and assume that the auxiliary relations $R_1, \ldots, R_m$ are enumerated with respect to a topological sorting of the dependency graph of $\mathcal{P}$ with $R_m = Q$.

We define inductively, by i, for every sequence $\delta_1 \ldots \delta_j$ with $j \geq i$, first-order formulas $\varphi^{R_i}_{\delta_1 \ldots \delta_j}(\bar{y}, \bar{x}, \ldots, \bar{x})$ over the schema $\tau_{\text{inp}} = E$ so that $\varphi^{R_i}_{\delta_1 \ldots \delta_j}$ defines R_i after modifications $\delta_1(\bar{x_1}) \ldots \delta_j(\bar{x_j})$. More precisely $\varphi^{R_i}_{\delta_1 \ldots \delta_j}$ will be defined so that for every state $\mathcal{S} = (V, E^{\mathcal{S}}, \mathcal{A}^{\mathcal{S}})$ of $\mathcal{P}$ and every sequence $\delta = \delta_1(\bar{a}_{1 \ldots} \delta_j(\bar{a}_j))$ of modifications the following holds:

$$\mathcal{P}_\delta(\mathcal{S}) \restriction R_i = \{\bar{b} | (V, E) \vDash \varphi^{R_i}_{\delta_1 \ldots \delta_j}(\bar{b}, \bar{a}_1 \ldots \bar{a}_j)\} \tag{3}$$

Here $\mathcal{P}_\delta(\mathcal{S})_E \restriction R_i$ denotes the relation stored in R_i in state $\mathcal{P}_\delta(S)$. For $R_0 = E$ the formula $\varphi^E_{\delta_1 \ldots \delta_j}(y, \bar{x}_1, \bar{\ldots}, \bar{x}_j)$ is obtained from the update formula $\phi^{R_i}_{\delta_j}(\bar{x}; \bar{y})$

of R_i by substituting all occurences of $R_{i'}(\bar{z})$ by $\varphi^{R'_i}_{\delta_1 \dots \delta_{j-1}}(\bar{x}_1, \dots, \bar{x}_{j-1}, \bar{z})$ for all $i' \leq i$. Using induction over i, one can prove that the formulas $\varphi^{R_i}_{\delta_i \dots \delta_j}$ satisfy Equation 3. As $\mathcal{P}$ is non-recursive, each formula $\delta^{R_i}_{\delta_1 \dots \delta_j}$ with $j \geq i$ is over the schema $\{E\}$.

The first-order formula φ for $\mathcal{Q}$ over the schema $\tau_{\text{inp}} = \{E\}$ can be constructed in the following way. The formula "guesses" a tuple $\bar{a} \in E$, deletes it and inserts it m times and applies $\varphi^{R_m}_{(\text{INS},\text{DEL})^m}$ to the result.

More precisely, φ for $\mathcal{Q}$ is defined by

$$\varphi(\bar{y}) \stackrel{\text{def}}{=} \exists \bar{x}((E(\bar{x}) \wedge \varphi^{R_m}_{(\text{DEL INS})^m}(\bar{y}, \underbrace{\bar{x}, \dots, \bar{x}}_{2m\text{-times}})) \vee (\neg E(\bar{x}) \wedge \varphi^{R_m}_{(\text{INS DEL})^m}(\bar{y}, \underbrace{\bar{x}, \dots \bar{x}}_{2m\text{-times}})))$$

□

And with these lemmas, we get the following result.

Theorem 3.29. *For every query $\mathcal{Q}$ the following statements are equivalent*

(a) $\mathcal{Q}$ can be expressed in FO.

(b) $\mathcal{Q}$ can be maintained in non-recursive DynFO.

(c) $\mathcal{Q}$ can be maintained in non-recursive Dyn$\exists^1$FO.

(d) $\mathcal{Q}$ can be maintained in non-recursive Dyn$\forall^1$FO.

Proof.

(a) ⇒ (c): follows from Lemma 3.27.

(c) ⇒ (d): follows from Theorem 3.10 since the proof does not introduce recursion when applied to a non-recursive program.

(d) ⇒ (b): by definition.

(b) ⇒ (a): follows from Lemma 3.28.

□

Combining Theorem 3.29 with Theorem 3.10 yields the following corollary:

Corollary 3.30. *Every first-order query can be maintained in DynCQ$^\neg$.*

4 Conclusions

We have introduced the dynamic complexity framework, defined multiple different dynamic complexity classes both with absolute semantics and Δ-semantics and shown how to maintain queries with dynamic programs.

We have also compared the relative power of different complexity classes against each other and against static complexity classes. Although we covered quite a few fragments of DynFO, more have been studied. Maybe most notable are classes where the arity of auxiliary relations for update formulas has been restricted.

Although many equalities between dynamic complexity classes have been drawn in this thesis, the amount of strict separations made is very small. Finding separations between classes requires finding lower bounds for them, an endeavour that has proven to be difficult but nonetheless, some results have been established. For a number of them, see Chapter 4 in [10].

References

[1] Erich Graedel, Phokion Kolaitis, Leonid Libkin, Maarten Marx, Joel Spencer, Moshe Vardi, Yde Venema, and Scott Weinstein. *Finite Model Theory and Its Applications.* Springer Science & Business Media, 01 2007.

[2] Siam-ams Proceedingn and Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets'. *SIAM-AMS Proc.*, 7, 01 1974.

[3] Neil Immerman. Upper and lower bounds for first order expressibility. *J. Comput. Syst. Sci.*, 25(1):76–98, 1982.

[4] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). *Proceedings of the fourteenth annual ACM symposium on Theory of computing - STOC '82*, 1982.

[5] David A. Mix Barrington, Neil Immerman, and Howard Straubing. On uniformity within nc^1. *J. Comput. Syst. Sci.*, 41(3):274–306, 1990.

[6] Heribert Vollmer. *Introduction to Circuit Complexity - A Uniform Approach.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 1999.

[7] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[8] Guozhu Dong and Jianwen Su. First-order incremental evaluation of datalog queries. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages, Manhattan, New York City, USA, 30 August - 1 September 1993*, Workshops in Computing, pages 295–308. Springer, 1993.

[9] Sushant Patnaik and Neil Immerman. Dyn-fo: A parallel, dynamic complexity class. In Victor Vianu, editor, *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 24-26, 1994, Minneapolis, Minnesota, USA*, pages 210–221. ACM Press, 1994.

[10] Thomas Zeume. *Small Dynamic Complexity Classes: An Investigation into Dynamic Descriptive Complexity*, volume 10110 of *Lecture Notes in Computer Science*. Springer, 2017.

[11] Thomas Zeume and Thomas Schwentick. On the quantifier-free dynamic complexity of reachability. *CoRR*, abs/1306.3056, 2013.

[12] William M Hesse and Neil Immerman. *Dynamic computational complexity.* PhD thesis, University of Massachusetts Amherst, 2003.

[13] Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, February 26-28, 2009, Freiburg, Germany, Proceedings*, volume 3 of *LIPIcs*, pages 481–492. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.