Chapter 2
Dynamic Complexity: Definitions and Examples

In this chapter the dynamic complexity framework is defined formally; and many examples for its expressive power are presented. As the dynamic complexity framework is based on notions from database theory and finite model theory, we will shortly review necessary foundations from those two areas first. Afterwards we will introduce the formal dynamic complexity framework, define three basic dynamic complexity classes and present simple examples for each of them. The formal framework has several aspects that can be varied; those will be discussed subsequently.

In order to get some familiarity with the setting, this chapter will be concluded by an extensive case study on path queries for graph databases. Recently query languages for graph databases have drawn considerable attention from the database theory community (see e.g. [MW95, AV99, ACP12, LM13, Woo12, Bae13] for surveys). Graph databases can contain huge amounts of data and therefore evaluating queries in parallel and, if possible, dynamically is highly desired. This motivates to study graph query languages from the dynamic complexity perspective. We will see how to maintain regular path queries as well as certain non-regular path queries using dynamic programs. The maintenance of regular path queries relies on the recent result that reachability can be maintained in by first-order update formuals. An overview for the proof of this result is presented. Rather than giving a complete picture for graph queries, we aim at developing some intuition of the capabilities of dynamic programs. This intuition will be of help in Chaps. 3 and 4.

Parts of this chapter originated in joint work with Samir Datta, Raghav Kulkarni, Anish Mukherjee and Thomas Schwentick; and discussions with Katja Losemann. For detailed bibliographic remarks we refer to the end of this chapter.

2.1 Preliminaries

In this section we review basic definitions in order to fix notations. We list definitions that are used throughout the whole work; specific notations are introduced in later chapters.

T. Zeume, *Small Dynamic Complexity Classes*, LNCS 10110
DOI: 10.1007/978-3-662-54314-6_2

Basic Notations

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 $\bar{a} = (a_1, \ldots, a_k)$ and $\bar{b} = (b_1, \ldots, b_\ell)$ over A, the $(k+\ell)$-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 $\bar{a} \cup \bar{b}$ for the set $\{a_1, \ldots, a_k, b_1 \ldots, b_\ell\}$. The tuple $\bar{a}$ is $\prec$-*ordered* with respect to a linear order $\prec$ of A, if $a_1 \prec \ldots \prec a_k$. If π is a function on A, we denote $(\pi(a_1), \ldots, \pi(a_k))$ by $\pi(\bar{a})$.

Structures and First-order Logic

We shortly review basic notions from finite model theory. We emphasize that in this work we are solely interested in finite structures. For a detailed introduction to the field we refer the reader to [EF05, Lib04].

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 schema τ with 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.

A τ-*structure* $\mathcal{S}$ is a pair $(D, \mathcal{D})$ where $\mathcal{D}$ is a database over schema τ and D is a domain. 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 $\mathcal{S}$. The substructure $\mathcal{S}'$ of $\mathcal{S}$ induced by some $D' \subseteq D$ is denoted by $\mathcal{S} \restriction D'$. In this work we always assume that the domain of a substructure contains all constants from the structure itself.

Let $\mathcal{S}$ and $\mathcal{T}$ be two structures over schema τ with domains S and T, respectively. A mapping $\pi : S \to T$ *preserves* a relation symbol $R \in \tau$ of arity m, when $\bar{a} \in R^{\mathcal{S}}$ if and only if $\pi(\bar{a}) \in R^{\mathcal{T}}$ for all m-tuples $\bar{a}$. It preserves a constant symbol c if and only if $\pi(c^{\mathcal{S}}) = c^{\mathcal{T}}$. The structures $\mathcal{S}$ and $\mathcal{T}$ are *isomorphic via* π , denoted by $\mathcal{S} \simeq_\pi \mathcal{T}$, if π is a bijection from S to T that preserves all relation and constant symbols in τ. The bijection $id[\bar{a}, \bar{b}]$ from S to S with $\bar{a} = (a_1, \ldots, a_k)$ (where all a_i are pairwise distinct) and $\bar{b} = (b_1, \ldots, b_k)$ (where all b_i are pairwise distinct) will be used a couple of times; it maps a_i to b_i, b_i to a_i and every other element of S to itself.

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

- Every *atomic formula* of the form $R(t_1, \ldots, t_k)$ or $t_1 = t_2$, where all t_i are either constant symbols or variables, is a first-order formula.
- 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.

Let $\mathcal{S} \stackrel{\text{def}}{=} (D, \mathcal{D})$ be a τ-structure, φ a first-order formula over τ with free variables $x_1, \ldots, x_k$, and α an assignment that maps every x_i to an element from D. By $(\mathcal{S}, \alpha) \models \varphi$ we indicate that $(\mathcal{S}, \alpha)$ is a model of φ. Often we write also

$(\mathcal{S}, \bar{a}) \models \varphi$ or $\mathcal{S} \models \varphi(\bar{a})$ where $\bar{a} \stackrel{\text{def}}{=} (\alpha(x_1), \ldots, \alpha(x_k))$. The model relation $\models$ 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 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 $\text{ANS}(\mathcal{Q}, \mathcal{S}) = \{\bar{a} \mid (\mathcal{S}, \bar{a}) \models \varphi(\bar{x})\}$ for all structures $\mathcal{S}$.

The *k-ary atomic type* $\langle \mathcal{S}, \bar{a} \rangle$ of $\bar{a} \in D^k$ with respect to a structure $\mathcal{S}$ over τ is the conjunction of all atomic formulas $\varphi(\bar{x})$ over τ for which $\mathcal{S} \models \varphi(\bar{a})$.

Standard Structures and Queries

The following structures and queries will be used throughout this work. A (directed) *graph* G is a pair (V, E) where V is a finite set and E is a subset of V^2. Graphs can be encoded as structures over schema $\{E\}$ where E is a binary relation symbol (to be interpreted by the set of edges). Usually we identify graphs and their corresponding structures.

An *s-t-graph* is a graph with two distinguished nodes s and t. Such graphs can be encoded by structures over schema $\{E, s, t\}$ where E is as before and s and t are two constant symbols (to be interpreted by two distinguished nodes). A *k-layered s-t-graph* G is an s-t-graph in which $V - \{s, t\}$ is partitioned into k layers $A_1, \ldots, A_k$ such that every edge is from s to A_1, from A_k to t or from A_i to A_{i+1} for some $i \in \{1, \ldots, k-1\}$.

The *reachability query* REACH, the *k-clique query* k-CLIQUE and the *k-colorability query* k-COL are defined as usual. A tuple (a, b) is in $\text{REACH}(G)$ if b can be reached from a in G. The *s-t-reachability query* s-t-REACH is a Boolean query which is true for an s-t-graph G, if and only if $(s, t) \in \text{REACH}(G)$. A graph $G = (V, E)$ is in k-CLIQUE if V contains k nodes $v_1, \ldots, v_k$ such that $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$ for all $1 \leq i < j \leq k$.

A *k-node-coloring col* of G is a mapping that assigns to every node of V a color from $\{1, \ldots, k\}$. Such a coloring is admissible, if all nodes a and b with $(a, b) \in E$ are colored by different colors. A graph is k-node-colorable, if it admits a k-node-coloring. The graph $G = (V, E)$ is in k-COL if it is k-node-colorable.

2.2 The Dynamic Complexity Framework

After the informal discussion of the dynamic complexity framework in the introduction chapter, we present the basic formal framework now. In Example 1.1, the structure subjected to modifications was a graph and modifications were restricted to be insertions. In the general dynamic complexity framework, arbitrary structures

are subject to both tuple insertions and tuple deletions. We present a variant of the framework introduced by Patnaik and Immerman [PI94].

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 $\alpha \stackrel{\text{def}}{=} \alpha_1 \dots \alpha_m$ to a database $\mathcal{D}$ is defined by $\alpha(\mathcal{D}) \stackrel{\text{def}}{=} \alpha_m(\dots(\alpha_1(\mathcal{D}))\dots)$.

Dynamic programs, to be defined next, consist of an initialization mechanism and an update program. The former yields, for every (input) database $\mathcal{D}$, an initial state with initial auxiliary data. The latter defines how the new state of the dynamic program is obtained from the current state when applying a modification.

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 2.2.1 (Update program). *An* update program P *over a dynamic schema* (τ_{inp}, τ_{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* $S \in \tau_{inp}$*, an update formula* $\phi^R_\delta(\bar{x}; \bar{y})$ *over the schema* τ *where* $\bar{u}$ *and* $\bar{x}$ *have the same arity as* S *and* R*, respectively.*

A *program state* $\mathcal{S}$ over 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 $S \in \tau_{\text{inp}}$. 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 $\mathcal{A}'$, a relation symbol $R \in \tau_{\text{aux}}$ is interpreted by $\{\bar{b} \mid \mathcal{S} \models \phi^R_\delta(\bar{a}; \bar{b})\}$. The effect $P_\alpha(\mathcal{S})$ of applying a modification sequence $\alpha \stackrel{\text{def}}{=} \alpha_1 \dots \alpha_m$ to a state $\mathcal{S}$ is the state $P_{\alpha_m}(\dots(P_{\alpha_1}(\mathcal{S}))\dots)$.

Definition 2.2.2 (Dynamic program). *A* dynamic program *is a triple* (P, INIT, Q)*, where*

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

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

In Example 1.1 we have already seen how to maintain reachability in graphs under insertions. Now we present a simple dynamic program for maintaining the parity of a unary relation as another example. This already gives a glimpse of the power of dynamic programs with first-order update formulas, as parity and reachability are not expressible in first-order logic (even with arbitrary built-in predicates) [Ajt83, FSS84].

Example 2.2.3. The parity query asks whether the number of elements in a unary relation U is divisible by two. A dynamic program can maintain the parity of U using a single 0-ary auxiliary relation Q as follows:

$$\phi^{Q}_{\mathrm{INS}_U}(u; x, y) \stackrel{\text{def}}{=} (\neg U(u) \wedge \neg Q) \vee (U(u) \wedge Q)$$
$$\phi^{Q}_{\mathrm{DEL}_U}(u; x, y) \stackrel{\text{def}}{=} (U(u) \wedge \neg Q) \vee (\neg U(u) \wedge Q)$$

□

The following notions will be useful at several places. Two programs $\mathcal{P}$ and $\mathcal{P}'$ with the same input schema and with designated query symbols Q and Q' of the same arity are *equivalent* if Q and Q' store the same relation after the application of every modification sequence.

The *dependency graph* of a dynamic program $\mathcal{P}$ with auxiliary schema τ has the vertex set $V = \tau$ and an edge (R, R') if the relation symbol R' occurs in one of the update formulas for R. The *deletion dependency graph* is defined as the dependency graph except that only update formulas for delete operations are taken into account.

2.3 Three Basic Dynamic Complexity Classes

It is natural to look at the classes of queries that can be maintained by dynamic programs. In the following we introduce the dynamic complexity classes DynFO, DynProp and DynQF. These are the basic complexity classes studied in this work. They have been introduced in [PI94, Hes03b].

2.3.1 The Class DynFO

We start with the class of queries maintainable by dynamic programs with first-order updates.

Definition 2.3.1 (DynFO). DynFO *is the class of all dynamic queries that can be maintained by dynamic programs with first-order update formulas and arbitrary initialization mappings.*

The role of the initialization mapping will be discussed later in this section. We remark that the class DynFO is the prototype of dynamic complexity classes defined by referring to a static class $\mathcal{C}$; in this case FO. More generally, in the dynamic class Dyn$\mathcal{C}$ updates from class $\mathcal{C}$ are allowed. In Chap. 3 we will encounter several other classes of the form Dyn$\mathcal{C}$ where $\mathcal{C}$ is a fragment of FO.

The dynamic complexity class DynFO and all other classes presented in the following also come in a variant where the arity of the auxiliary relations is restricted. A dynamic program is *k-ary* if the arity of its auxiliary relation symbols is at most k. By k-ary DynFO we refer to dynamic queries that can be maintained with k-ary DynFO-programs.

As we already have seen above, the parity query is in DynFO; even in 0-ary DynFO. Many other queries have been shown to be in DynFO as well, e.g. two-colorability of graphs [DS98] and all context-free languages are in DynFO [GMS12]. Very recently reachability for arbitrary (directed) graphs has been shown to be maintainable in DynFO [DKM+15].

As an example we show how reachability on acyclic graphs can be maintained in DynFO. The technique used in this example will be the foundation for some results for maintaining graph queries on acyclic graphs in Sect. 2.6.

Example 2.3.2. We follow the argument from [PI97] and construct a dynamic DynFO-program with one binary auxiliary relation T which is intended to store the transitive closure of an acyclic graph.

Insertions can be handled straightforwardly as in the introductory example at the beginning of this section: 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 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'. If there is still a path p from x to y, such an edge (z, z') must exist on p, as otherwise u would be reachable from y contradicting acyclicity. All conditions can be checked using the transitive closure of the graph before the deletion of (u, v). The update formulas for T are as follows:

$$\phi^T_{\mathrm{INS}_E}(u, v; x, y) \stackrel{\text{def}}{=} T(x, y) \vee \big(T(x, u) \wedge T(v, y)\big)$$

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

□

Those examples already show that DynFO is very powerful. This is the reason why several restrictions of DynFO have been studied.

2.3.2 *The Class* DynProp

Disallowing quantifiers in update formulas yields the class DynProp. Thus, in DynProp, update formulas can only access the inserted or deleted tuple $\bar{a}$ and the currently updated tuple $\bar{b}$ of an auxiliary relation.

Definition 2.3.3 (DynProp). DynProp *is the class of all dynamic queries that can be maintained by dynamic programs with quantifier-free first-order update formulas and arbitrary initialization mappings.*

By k-ary DynProp we refer to dynamic queries that can be maintained with k-ary quantifier-free dynamic programs.

At first glance the restriction to quantifier-free programs appears severe. However, Example 2.2.3 shows that the parity query is also in DynProp, and Example 1.1 shows that reachability can be maintained in DynProp under insertions. Using a slightly more involved construction, Hesse proved that reachability on deterministic graphs can be maintained in DynProp [Hes03b]. Gelade et al. showed that on strings, all regular languages can be maintained in DynProp [GMS12]. Thus quantifier-free programs are already quite expressive; and, as we will see later, proving lower bounds for quantifier-free programs is non-trivial.

The following example illustrates a technique to maintain lists with quantifier-free dynamic programs which is used in some of our upper bound results. This technique was introduced in [GMS12, Proposition 4.5].

Example 2.3.4. We provide a DynProp-program $\mathcal{P}$ for the dynamic variant of the Boolean query NonEmptySet. This query asks, for a unary relation U subject to insertions and deletions of elements, whether U is empty. Of course, this query is trivially expressible in first-order logic, but not without quantifiers.

The program $\mathcal{P}$ uses the auxiliary schema $\tau_{\text{aux}} = \{Q, \text{First}, \text{Last}, \text{List}\}$, where Q is the query bit (i.e. a 0-ary relation symbol), First and Last are unary relation symbols, and List is a binary relation symbol. The idea is to store in a program state $\mathcal{S}$ a list of all elements currently in U. The list structure is stored in the binary relation $\text{List}^{\mathcal{S}}$ such that a tuple (a, b) is in $\text{List}^{\mathcal{S}}$ if a and b that are adjacent in the list. The first and last element of the list are stored in $\text{First}^{\mathcal{S}}$ and $\text{Last}^{\mathcal{S}}$, respectively. We note that the order in which the elements of U are stored in the list depends on the order in which they are inserted into the set.

For a given instance of NonEmptySet the initialization mapping initializes the auxiliary relations accordingly.

In the following we assume for simplicity that only elements that are not already in U are inserted. The given formulas can be extended easily to the general case. A similar assumption is made for deletions.

Insertion of a into U. A newly inserted element is attached to the end of the list. Therefore the First-relation does not change except when the first element is inserted into an empty set U. Furthermore, the inserted element is the new last element of

the list and has a connection to the former last element. Finally, after inserting an element into U, the query result is 'true':

$$\begin{aligned}
\phi^{\text{FIRST}}_{\text{INS}_U}(a; x) &\stackrel{\text{def}}{=} (\neg Q \wedge a = x) \vee (Q \wedge \text{FIRST}(x)) \\
\phi^{\text{LAST}}_{\text{INS}_U}(a; x) &\stackrel{\text{def}}{=} a = x \\
\phi^{\text{LIST}}_{\text{INS}_U}(a; x, y) &\stackrel{\text{def}}{=} \text{LIST}(x, y) \vee (\text{LAST}(x) \wedge a = y) \\
\phi^{Q}_{\text{INS}_U}(a) &\stackrel{\text{def}}{=} \top.
\end{aligned}$$

Deletion of a from U. How a deleted element a is removed from the list, depends on whether a is the first element of the list, the last element of the list or some other element of the list. The query bit remains 'true', if a was not the first *and* last element of the list.

$$\begin{aligned}
\phi^{\text{FIRST}}_{\text{DEL}_U}(a; x) &\stackrel{\text{def}}{=} (\text{FIRST}(x) \wedge a \neq x) \vee (\text{FIRST}(a) \wedge \text{LIST}(a, x)) \\
\phi^{\text{LAST}}_{\text{DEL}_U}(a; x) &\stackrel{\text{def}}{=} (\text{LAST}(x) \wedge a \neq x) \vee (\text{LAST}(a) \wedge \text{LIST}(x, a)) \\
\phi^{\text{LIST}}_{\text{DEL}_U}(a; x, y) &\stackrel{\text{def}}{=} x \neq a \wedge y \neq a \wedge \big(\text{LIST}(x, y) \vee (\text{LIST}(x, a) \wedge \text{LIST}(a, y))\big) \\
\phi^{Q}_{\text{DEL}_U}(a) &\stackrel{\text{def}}{=} \neg(\text{FIRST}(a) \wedge \text{LAST}(a))
\end{aligned}$$

□

2.3.3 The Class DynQF

Quantifier-free programs, as the one above, can only access the inserted or deleted tuple and the currently updated tuple of an auxiliary relation. Dynamic programs with first-order update formulas, on the other hand, have great freedom in choosing tuples to access. In the following we consider a class of queries maintainable by programs that lay in between those two extremes.

To define this class of dynamic programs, we extend the class of programs with quantifier-free update formulas as follows. In addition to auxiliary relations, we allow auxiliary functions to be stored as auxiliary data. With auxiliary functions further elements might be accessed via function terms over the modified tuple and the currently updated tuple. Thus, in a sense, auxiliary functions can be seen as adding weak quantification to quantifier-free formulas. When full first-order updates are available, auxiliary functions can be simulated in a straightforward way by auxiliary relations. However, without quantifiers this is not possible.

The class of queries maintainable by quantifier-free programs extended by auxiliary functions is called DynQF. Before giving the formal definition of DynQF, we formalize the extension of the basic dynamic complexity framework by functions. We follow the approach from [GMS12].

When talking about DynQF, the auxiliary schema τ_{aux} is the union of a relational schema τ_{rel} and a *functional schema* τ_{fun}. It has an associated arity function Ar :

$\tau_{\text{rel}} \cup \tau_{\text{fun}} \mapsto \mathbb{N}$. A database $\mathcal{D}$ over such a schema with 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 k-ary function symbol $f \in \tau_{\text{fun}}$ a k-ary function from D^k to D. We observe that constants can be modeled by 0-ary functions.

In the following, we extend our definition of update programs with relational auxiliary schemas to programs with auxiliary schemas with function symbols. It is straightforward to extend the definition of update formulas for auxiliary relations: they simply can make use of function terms. However, following the spirit of DynProp, a more powerful update mechanism for auxiliary functions is used which allows case distinctions in addition to composition of function terms.

Definition 2.3.5 (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 $t_1, \ldots, t_k$ are update terms, then $f(t_1, \ldots, t_k)$ 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) is straightforward. If $I \models \phi$ 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_δ^R (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 2.3.6 (DynQF). DynQF *is the class of queries maintainable by quantifier-free update programs with (possibly) auxiliary functions and arbitrary initialization mappings.*

The class k-ary DynQF is defined via update programs that use auxiliary functions and relations of arity at most k.

We remark that our definition of DynQF is slightly stronger than the usual definition. Here we allow for using update terms in update formulas for relations whereas in [GMS12] only terms are allowed. This strengthens several of our later results.

Auxiliary functions are quite powerful. While only regular languages can be maintained in DynProp, all Dyck languages, among other non-regular languages, can be maintained in DynQF [GMS12]. Furthermore, undirected reachability can be maintained in DynQF with built-in arithmetic [Hes03b, Corollary 4.9].

The following example provides an impression of the expressive power of DynQF.

Example 2.3.7. Consider the unary graph query $\mathcal{Q}$ that returns all nodes a of a given graph G with maximal outdegree.

We construct a unary DynQF-program $\mathcal{P}$ that maintains $\mathcal{Q}$ in a unary relation denoted by the designated query symbol Q. The program uses two unary functions Succ and Pred that shall encode a successor and its corresponding predecessor relation on the domain. For simplicity, but without loss of generality, we therefore assume that the domain is of the form $D = \{0, \ldots, n-1\}$. For every state $\mathcal{S}$, the function $\text{Succ}^{\mathcal{S}}$ is then the standard successor function on D (with $\text{Succ}^{\mathcal{S}}(n-1) = n-1$), and $\text{Pred}^{\mathcal{S}}$ is the standard predecessor function (with $\text{Pred}^{\mathcal{S}}(0) = 0$). Both functions are initialized accordingly. In the following we refer to *numbers* and mean the position of elements in Succ. The program uses constants for representing the numbers 0 and 1.

The program $\mathcal{P}$ maintains two unary functions #edges and #nodes. The function #edges counts, for every node a, the number of outgoing edges of a; more precisely $\#\text{edges}(a) = b$ if and only if b is the number of outgoing edges of a. The function #nodes counts, for every number a, the number of nodes with a outgoing edges; more precisely $\#\text{nodes}(a) = b$ if and only if b is the number of nodes with a outgoing edges. A constant Max shall always point to the number i such that i is the maximal number of outgoing edges from some node in the current graph.

When inserting an outgoing edge (u, v) for a node u that already has a outgoing edges, the counter #edges of u is incremented from a to $a+1$ and all other edge-counters remain unchanged. The counter #nodes of a is decremented, the counter of $a+1$ is incremented, and all other node-counters remain unchanged. The number Max increases if, before the insertion, u was a node with maximal number of outgoing edges. This yields the following update terms:

$$t^{\#\text{edges}}_{\text{ins}_E}(u, v; x) \stackrel{\text{def}}{=} \text{ite}\Big(\neg E(u, v) \wedge x = u, \text{Succ}(\#\text{edges}(x)), \#\text{edges}(x)\Big)$$

$$t^{\#\text{nodes}}_{\text{ins}_E}(u, v; x) \stackrel{\text{def}}{=} \text{ite}\Big(\neg E(u, v) \wedge x = \#\text{edges}(u), \text{Pred}(\#\text{nodes}(x)), \text{ite}\big(\neg E(u, v) \wedge x = \text{Succ}(\#\text{edges}(u)), \text{Succ}(\#\text{nodes}(x)), \#\text{nodes}(x)\big)\Big)$$

$$t^{\text{Max}}_{\text{ins}_E}(u, v) \stackrel{\text{def}}{=} \text{ite}\Big(\text{Max} = \#\text{edges}(u) \wedge \neg E(u, v), \text{Succ}(u), \text{Max}\Big)$$

The update formula for the designated query symbol Q is as follows:

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

The update terms for deletions are very similar:

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

$$t^{\#\text{NODES}}_{\text{DEL}_E}(u, v; x) \stackrel{\text{def}}{=} \text{ITE}\Big(E(u, v) \wedge x = \#\text{EDGES}(u), \text{PRED}(\#\text{NODES}(x)),$$
$$\text{ITE}\big(E(u, v) \wedge x = \text{PRED}(\#\text{EDGES}(u)),$$
$$\text{SUCC}(\#\text{NODES}(x)), \#\text{NODES}(x)\big)\Big)$$

$$t^{\text{MAX}}_{\text{DEL}_E}(u, v) \stackrel{\text{def}}{=} \text{ITE}\Big(\text{MAX} = \#\text{EDGES}(u) \wedge E(u, v) \wedge \#\text{NODES}(\text{MAX}) = 1,$$
$$\text{PRED}(\text{MAX}), \text{MAX}\Big)$$

The update formula for the designated query symbol Q under deletion is as follows:

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

2.4 Variants of the Dynamic Complexity Framework

Many variations of the basic dynamic complexity framework presented above have been studied. In the following we discuss three aspects of the basic framework that can be varied: the type of the domain; the initial state of a dynamic program and the power of the initialization mapping; and, closely related, whether built-in relations are available. For each of the aspects we also say which variants we consider in this work and explain why we chose those variants.

Choosing a Domain

While in the framework of Patnaik and Immerman as well as in our framework fixed and finite domains are used, the first-order incremental evaluation system framework (short: FOIES) introduced by Dong, Su and Topor [DT92, DS93] uses active domains, where the domain accessible by the update formulas contains only elements currently used in input relations. The choice of domains is the main diference between our basic framework and the FOIES framework. We discuss both choices in the following.

In this work, we will almost exclusively use finite and fixed domains and thus follow the approach taken by Patnaik and Immerman [PI94]. Here, fixed means that domains do not change while modifications are applied. Yet a dynamic program has to work uniformly for all domain sizes in order to maintain a query. Choosing fixed domains for a dynamic setting might appear counterintuitive at first sight.

After all, dynamic complexity is about dynamically changing structures, in particular structures might grow without any a priori bound on the size of the domain when tuples are inserted. However, it turns out that fixed and finite domains are sufficient to study the underlying dynamic mechanisms of dynamic programs. They also offer a strong connection to logics and circuit complexity, which will be used a couple of times in this work. Very often upper and lower bounds obtained for fixed, finite domains can be easily transferred to settings where the domain can depend on the input database, and vice versa.

When using active domains, only elements used in the input database are contained in the domain. When a tuple containing a so far unused element a is inserted into the input database, then a is also inserted into the domain accessible to the update formulas. This setting is a little closer to real database systems but most results in dynamic complexity hold equally for both fixed, finite domains and active domains.

Initializations

How the initial state of a dynamic program looks like depends on two choices: the set of permissible input databases and the power of the initialization mapping. We first highlight some possible options for those two choices, how those options have been combined in previous work and how they relate to each other; then we discuss which initialization settings we use in this work and why.

In our basic framework the initial input database can be arbitrary. Always starting from scratch, that is from an initially empty input database, is another option.

The initialization mapping in our basic dynamic complexity classes may assign arbitrary complex auxiliary relations. Other options are to restrict the mapping to be computable in some static complexity class, to be permutation-invariant (short: invariant initialization), or to always assign empty initial auxiliary relations. Intuitively a permutation-invariant initialization mapping maps isomorphic databases to isomorphic auxiliary databases. A particular case of permutation-invariant initialization mappings, studied in [GS12], is when the initialization is specified by some logical formalism.

Several combinations of those options have been used in the literature. We state some examples only. In [PI97], DYN- FO is defined as the class of (Boolean) queries that can be maintained for empty initial input databases with first-order update formulas and a first-order definable initialization mapping. Furthermore, a larger class DYN- FO^+ that extends DYN- FO by polynomial-time computable initialization mapping has been studied by Patnaik and Immerman. Also [Ete98] considers empty initial databases. In [WS07], general instances (with non-empty initial databases) are allowed, and auxiliary data is initialized by a mapping computable in some given complexity class. In [GS12], also general instances are allowed, but the initialization mapping has to be defined by logical formulas and is thus always permutation-invariant.

Some of the initialization settings are the same. First we note that for arbitrary initialization mappings, the same queries can be maintained regardless of whether

one starts from an empty or from a non-empty initial database. This is because the initialization for a non-empty database can be obtained as the auxiliary relations obtained after inserting all tuples of the database into the empty one. Furthermore, it is easy to see that applying an invariant initialization mapping to an empty database is pretty much useless, as, all tuples with the same constants at the same positions are treated in the same way. Therefore, queries maintainable in DynFO with empty initial database and invariant initialization can also be maintained from initially empty input and auxiliary database. We do not formally prove this here.

We shortly discuss which settings are used in this work. For upper bounds, we try to make the initialization as weak as possible. Yet, the weakest setting we use is the original setting of Patnaik and Immerman, that is, the setting with initially empty input database and first-order definable initialization mapping [PI97]. If a query can be maintained, for example, in this setting with first-order update formulas, we say that it can be maintained *in* DynFO *from scratch*.

When proving lower bounds, we try to make the initialization as strong as possible. This is motivated by the fact that lower bounds in settings with restricted initialization might depend on the restriction. Thus our goal is to use the setting with arbitrary initial input databases and arbitrary initialization mappings for lower bounds whenever possible. An inexpressibility result in this setting shows that a query cannot be maintained dynamically at all.

In this work the basic dynamic complexity classes use the setting with arbitrary initial input databases and arbitrary initialization mapping as default. We explicitly state when a result does not adhere to this setting.

Built-In Relations

When studying fragments of DynFO with restricted arity, it is sometimes useful to allow for special built-in relations of larger arity. For example, a lower bound proof for unary DynProp could look like it depends on the fact that no linear order was present on the domain (as binary relations are not allowed in unary DynProp). Then proving a lower bound for unary DynProp with a built-in linear order can refute this.

For the study of dynamic complexity classes with built-in relations we extend dynamic schemas to triples $(\tau_{\text{inp}}, \tau_{\text{aux}}, \tau_{\text{bi}})$ where τ_{bi} is the schema for the built-in database. The built-in relations are initialized by an initialization mapping that only depends on the domain (and not on the input database). In contrast to the auxiliary database, the built-in database never changes throughout a "computation", that is, update formulas for built-in relations R are of the form $\phi_\delta^R(\bar{u}; \bar{x}) = R(\bar{x})$ for all modifications δ.

In this work, the main dynamic classes do not allow built-in relations. At times we also consider dynamic programs with non-empty built-in schemas. We denote the extension of a dynamic complexity class that allows programs with non-empty built-in schemas by a superscript *, as in DynProp*. For classes with restricted arity of the auxiliary schema, the restriction does not apply to the built-in schema. For

example, dynamic programs for queries in binary DynProp* can only use binary auxiliary relations, but built-in relations of arbitrary arity.

We use built-in databases only to strengthen some results in one of two possible ways, (1) by showing upper bounds in which (some) auxiliary relations or functions need not to be updated or (2) by showing inexpressibility results that hold for auxiliary schemas of bounded arity but with built-in relations of unbounded arity. In general, built-in data can be "simulated" by auxiliary data. However, this needs not to hold, e.g., if the auxiliary schema is more restricted than the built-in schema.

2.5 A Case Study: Graph Queries

The goal of this section is to develop a better intuition for the capabilities of dynamic programs. To this end we will study graph query languages from the perspective of dynamic complexity. We refer to the introduction of this chapter for a motivation for studying graph query languages in this context.

Formally, a *graph database* is a labeled graph $G = (V, E)$ where V is the set of nodes and E is a set of labeled edges $(x, \sigma, y) \subseteq V \times \Sigma \times V$ for some alphabet Σ. Here σ is called the *label* of edge (x, σ, y). The graph underlying a labeled graph is obtained by removing all labels from the database. A labeled graph is acyclic if its underlying graph is acyclic; it is undirected if, for each $\sigma \in \Sigma$, the projection of G to edges labeled by σ is undirected.

A path p in G is labeled with a word $w = \sigma_1 \dots \sigma_\ell$ if the edges $e_1, \dots, e_\ell$ along p are labeled with $\sigma_1, \dots, \sigma_\ell$. We also say that p is a *w-path*. A path is an *L-path*, for a formal language L, if it is a w-path for some $w \in L$.

Here we focus on path-based graph query languages. Every formal language L can be seen as a path query that selects a pair (x, y) of nodes from a given labeled graph if there is an L-path from x to y. Usually we identify a path query with its defining formal language. A path query is called *regular path query* if L is regular, similarly for context-free path queries and other classes of formal languages. A conjunctive regular path query is a query of the form $\mathcal{Q}(\bar{\S}) \stackrel{\text{def}}{=} \exists \bar{\ddagger} \bigwedge_{\rangle} \dagger_{\rangle} \mathcal{L}_{\rangle} \dagger'_{\rangle}$ where each L_i is a regular language, and each y_i, y'_i is either contained in $\bar{x}$ or in $\bar{z}$. Informally, a tuple $\bar{x}$ is selected by the query if there is a tuple $\bar{z}$ such that for each i there is an L_i-path from y_i to y'_i. For a more thorough introduction to graph queries we refer to [Bae13].

In dynamic complexity, even maintaining path queries as simple as $L(a^*)$ was, until recently, not possible using first-order update formulas because this requires to maintain reachability. Very likely this is the reason why graph queries have (almost) not been studied at all in dynamic complexity.

Some related work has been done before. Already Patnaik and Immerman pointed out that regular languages can be maintained in DynFO [PI97]. Later Gelade et al. systematically studied formal languages in the dynamic complexity framework [GMS12]. They showed, among other results, that regular languages coincide with DynProp, and that all context-free languages can be maintained in DynFO. A result

for path queries has been obtained by Weber and Schwentick in [WS07]: the Dyck language D_2 can be maintained in DynFO on acyclic graphs.

Here we present some further results for maintaining graph queries. As our main goal is to develop some intuition, those results are not meant to be exhaustive.

It turns out that regular path queries can be evaluated dynamically in a highly parallel fashion.

Theorem 2.5.1. *(a) When only insertions are allowed then every regular path query can be maintained from scratch in* DynProp.
(b) Every conjunctive regular path query can be maintained from scratch in DynFO.

The proof of the second part relies on the very recent result that reachability can be maintained from scratch in DynFO [DKM+15]. This result will not be proved in detail here.

Theorem 2.5.2. *Reachability can be maintained in* DynFO *from scratch.*

Proof overview. The proof has three main steps.

Step 1. Show that reachability can be maintained in DynFO if the rank of a matrix subject to modifications can be maintained in DynFO.
Step 2. Construct a DynFO-program with built-in arithmetic that maintains the rank of a matrix.
Step 3. Show that if a domain independent query is in DynFO with built-in arithmetic then it can be maintained in DynFO from scratch (and, in particular, without built-in arithemtic).

The statement of the theorem follows from Steps (1)–(3) since the reachability query is domain independent. In Section 3.4 we present Step 3 in detail. □

Capturing non-regular path queries by one of our basic dynamic complexity classes seems to be significantly harder. We provide only some preliminary results for restricted classes of graphs and modifications.

Theorem 2.5.3. *(a) Context-free path queries can be maintained from scratch in* DynFO *on acyclic graphs.*
(b) There is a non-context-free path query that can be maintained from scratch in DynFO *on acyclic graphs.*
(c) There is a non-context-free path query that can be maintained from scratch in DynFO *when only insertions are allowed.*

Part (a) of the theorem generalizes results and techniques from [GMS12, WS07].

2.5.1 Regular Path Queries

In this subsection we prove Theorem 2.5.1. For proving the first part of the theorem, the following notion will be useful. Let $\mathcal{A}$ be a deterministic finite state automaton

(short: DFA) and let G be a labeled graph. Then a path p in G can be *read by* $\mathcal{A}$ *starting in a state* q *and ending in a state* r if $\mathcal{A}$ can reach state r from state q by reading the label sequence of p.

Proposition 2.5.4. *When only insertions are allowed then every regular path query can be maintained in* DynProp *from scratch.*

Proof. Let L be a regular path query and let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be a DFA with $L = L(\mathcal{A})$. We construct a DynProp-program $\mathcal{P}$ that maintains L.

The program $\mathcal{P}$ has input schema $\{E_\sigma \mid \sigma \in \Sigma\}$ and an auxiliary schema that contains a binary relation symbol $R_{q,r}$ for every tuple $(q, r) \in Q^2$, as well as a binary designated query symbol R. The simple idea is that in a state $\mathcal{S}$ with underlying labeled graph G, the relation $R^{\mathcal{S}}_{q,r}$ contains all tuples $(a, b) \in V^2$ such that $\mathcal{A}$, for some labeled path p from a to b, can read p by starting in state q and ending in state r.

The update formulas for the relations $R_{q,r}$ are slightly more involved than the formulas for maintaining reachability under insertions (see Example 1.1). This is because $\mathcal{A}$ might reach a state r from a state q only by reading a labeled path from a to b that contains one or more loops. The crucial observation is, however, that for deciding whether (a, b) is in $R_{q,r}$ it suffices to consider paths that contain the node a at most $|Q|$ times (as paths that contain a more than $|Q|$ times can be shortened). This suffices to maintain the relations $R_{q,r}$ dynamically.

The update formulas for $R_{q,r}$ and R are as follows:

$$\phi^{R_{q,r}}_{\mathrm{INS}_{E_\sigma}}(u, v; x, y) \stackrel{\text{def}}{=} R_{q,r}(u, v) \vee \bigvee_{q',r'} \Big(R_{q,q'}(x, u) \wedge \varphi^{|Q|}_{q',r'}(u, v) \wedge R_{r',r}(v, y) \Big)$$

$$\phi^{R}_{\mathrm{INS}_{E_\sigma}}(u, v; x, y) \stackrel{\text{def}}{=} \bigvee_{f \in F} \phi^{R_{s,f}}_{\mathrm{INS}_{E_\sigma}}(u, v; x, y)$$

Here the formula $\varphi^{|Q|}_{q',r'}(u, v)$ shall only be satisfied by tuples (a, b) for which there exists a path p from a to b such that $\mathcal{A}$ can read p by starting in q' and ending in r'. It shall be satisfied by all such tuples with a witness path p that contains node a at most $|Q|$ times.

We inductively define, for every $1 \leq i \leq |Q|$ and all $q, r \in Q$, the slightly more general formulas $\varphi^i_{q,r}(u, v)$ as follows:

$$\varphi^1_{q,r}(u, v) \stackrel{\text{def}}{=} [(q, \sigma, r) \in \delta] \vee R_{q,r}(u, v)$$

$$\varphi^i_{q,r}(u, v) \stackrel{\text{def}}{=} \varphi^{i-1}_{q,r}(u, v) \vee \bigvee_{q',r'} \Big(\varphi^1_{q,q'}(u, v) \wedge R_{q',r'}(v, u) \wedge \varphi^{i-1}_{r',r}(u, v) \Big)$$

□

We conjecture that DynProp cannot maintain regular path queries for both insertions and deletions. This would imply that reachability can be maintained in DynProp which is very unlikely. A first step towards verifying this conjecture is done in Sect. 4.1.2 where we show that reachability cannot be maintained in binary DynProp.

However, regular path queries with insertions and deletions can be maintained in DynFO. The main ingredient for the proof is that reachability can be maintained in DynFO.

Proposition 2.5.5. *Every conjunctive regular path query can be maintained from scratch in* DynFO.

Proof sketch. Since DynFO is closed under conjunctions and existential quantification, it suffices to show that regular path queries can be maintained in DynFO. We reduce the maintenance of regular path queries to reachability. The approach is very similar to the approach taken in [KW03] for showing that dynamic LTL model checking can be maintained by a TC^0-update mechanism.

The product graph $G \times \mathcal{A}$ of a labeled graph $G = (V, E)$ and an DFA $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ has nodes $V \times Q$, and the edge set contains a tuple $((u, q), (v, r))$ if and only if there is a $\sigma \in \Sigma$ such that both $(u, \sigma, v) \in E$ and $(q, \sigma, r) \in \delta$. It can be easily verified that there is a path from node a to node b in G labeled by $w \in L(\mathcal{A})$ if and only if there is a path from (a, s) to (b, f) in $G \times \mathcal{A}$ for some accepting state $f \in F$ of $\mathcal{A}$.

Now, let L be a regular path query and let $\mathcal{A} = (Q, \Sigma, \delta, s, F)$ be an DFA with $L = L(\mathcal{A})$. We construct a DynFO-program $\mathcal{P}$ that maintains L. The program $\mathcal{P}$ has input schema $\{E_\sigma \mid \sigma \in \Sigma\}$, yet it also stores an encoding of $G \times \mathcal{A}$ in its state. This can be easily achieved, for example, by using a 4-ary auxiliary relation $R_{G\times\mathcal{A}}$ that contains the tuples of $G \times \mathcal{A}$. For this it has to be assumed, without loss of generality, that G contains at least $|Q|$ nodes and that each $q \in Q$ is identified with a unique node. The result of a fixed regular path query for labeled graphs G with less than $|Q|$ nodes can be easily encoded by a first-order formula.

The crucial observation is that the modification of a single labeled edge in G leads to at most $|Q|$ modifications in $G \times \mathcal{A}$. Thus, due to Reach $\in$ DynFO [DKM+15] and the closure of DynFO under bounded expansion first-order reductions (see [PI97]), the transitive closure of $G \times \mathcal{A}$ can be maintained in a relation T.

Now, this relation can be used to maintain the regular path query via

$$\phi_\delta^Q(u, v; x, y) = \bigvee_{f \in F} T'((x, s), (y, f))$$

where updating T with modification δ yields T'. □

2.5.2 Beyond Regular Path Queries

The maintenance of non-regular path queries is more difficult. Here we prove the results stated in Theorem 2.5.3. All of them are either for restricted graph classes or for restricted modification sequences. We start with proving part (a).

Proposition 2.5.6. *Context-free path queries can be maintained from scratch in* DynFO *on acyclic graphs.*

It is known that context-free languages are in DynFO [GMS12] and that the Dyck language with two types of parentheses can be maintained on acyclic graphs [WS07]. We combine the techniques used in those two proofs in order to prove Proposition 2.5.6.

In the following we fix a context-free language L and a grammar $\mathcal{G} = (V, \Sigma, S, P)$ for L. We assume, without loss of generality, that $\mathcal{G}$ is in Chomsky normal form, that is, it has only rules of the form $X \to YZ$ and $X \to \sigma$. Further if $\epsilon \in L$ then $S \to \epsilon$ and no right-hand side of a rule contains S. We write $V \Rightarrow^* w$ if $w \in (\Sigma \cup V)^*$ can be derived from $Z \in V$ using rules of G.

The dynamic program maintaining L on acyclic graphs will use 4-ary auxiliary relation symbols $R_{Z \to Z'}$ for all $Z, Z' \in V$. The intention is that in every state $\mathcal{S}$ with input database G, the relation $R^{\mathcal{S}}_{Z \to Z'}$ contains a tuple (x_1, y_1, x_2, y_2) if and only if there are strings $s_1, s_2 \in \Sigma^*$ such that $Z \Rightarrow^* s_1 Z' s_2$ and there is an s_i-path p_i from x_i to y_i for $i \in \{1, 2\}$. The paths p_1 and p_2 are called *witnesses* for $(x_1, y_1, x_2, y_2) \in R^{\mathcal{S}}_{Z \to Z'}$. Later we will see that whether two nodes are connected by an L-path after an update can be easily verified using those relations.

It turns out that for updating the relations $R^{\mathcal{S}}_{Z \to Z'}$ it is necessary to have access to $(2k + 2)$-ary relations $R^{\mathcal{S}}_{X \to Y_1, \ldots, Y_k}$, for $k \in \{1, 2, 3\}$, which contain a tuple $(x_1, y_1, \ldots, x_{k+1}, y_{k+1})$ if and only if there are strings $s_1, \ldots, s_{k+1} \in \Sigma^*$ such that $X \Rightarrow^* s_1 Y_1 s_2 \ldots s_k Y_k s_{k+1}$ and there is an s_i-path p_i from x_i to y_i in the input database underlying $\mathcal{S}$.

Next, in Lemma 2.5.7, we prove that every relation $R^{\mathcal{S}}_{X \to Y_1, \ldots, Y_k}$ is first-order definable from the relations $R^{\mathcal{S}}_{Z \to Z'}$ (and thus only relations $R^{\mathcal{S}}_{Z \to Z'}$ have to be stored as auxiliary data). This lemma is inspired by Lemma 7.3 from [WS07], and its proof is a generalization of the technique used in the proof of Theorem 4.1 in [GMS12]. Afterwards we prove Proposition 2.5.6 by showing how to use the relations $R^{\mathcal{S}}_{Z \to Z'}$ to maintain L and how to update the relations $R^{\mathcal{S}}_{Z \to Z'}$ using the formulas that define relations of the form $R^{\mathcal{S}}_{X \to Y_1, Y_2}$ and $R^{\mathcal{S}}_{X \to Y_1, Y_2, Y_3}$.

Lemma 2.5.7. *For a grammar $\mathcal{G}$ in Chomsky normal form, $k \geq 2$ and variables $X, Y_1, \ldots, Y_k$ there is a first-order formula $\varphi_{X \to Y_1, \ldots, Y_k}$ over schema $\tau = \{R_{Z \to Z'} \mid Z, Z' \in V\}$ that defines $R_{X \to Y_1, \ldots, Y_k}$ in states $\mathcal{S}$ where the relations $R^{\mathcal{S}}_{Z \to Z'}$ are as described above.*

Proof sketch. We explain how $\varphi_{X \to Y_1, Y_2, Y_3}$ tests whether a tuple is contained in $R^{\mathcal{S}}_{X \to Y_1, Y_2, Y_3}$. The construction for general k is analogous.

If a tuple $(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)$ is contained in $R^{\mathcal{S}}_{X \to Y_1, Y_2, Y_3}$ witnessed by s_i-paths p_i from x_i to y_i such that $X \Rightarrow^* s_1 Y_1 s_2 Y_2 s_3 Y_3 s_4$, then in the derivation tree of $s_1 Y_1 s_2 Y_2 s_3 Y_3 s_4$ from X there is a variable U such that $U \to U_1 U_2$ and either (1) Y_1 and Y_2 are derived from U_1, and Y_3 is derived from U_2; or (2) Y_1 is derived from U_1, and Y_2 and Y_3 are derived from U_2. In case (1), the derivation subtree starting from U_1 contains a variable W such that $W \to W_1 W_2$ and Y_1 is derived from W_1 and

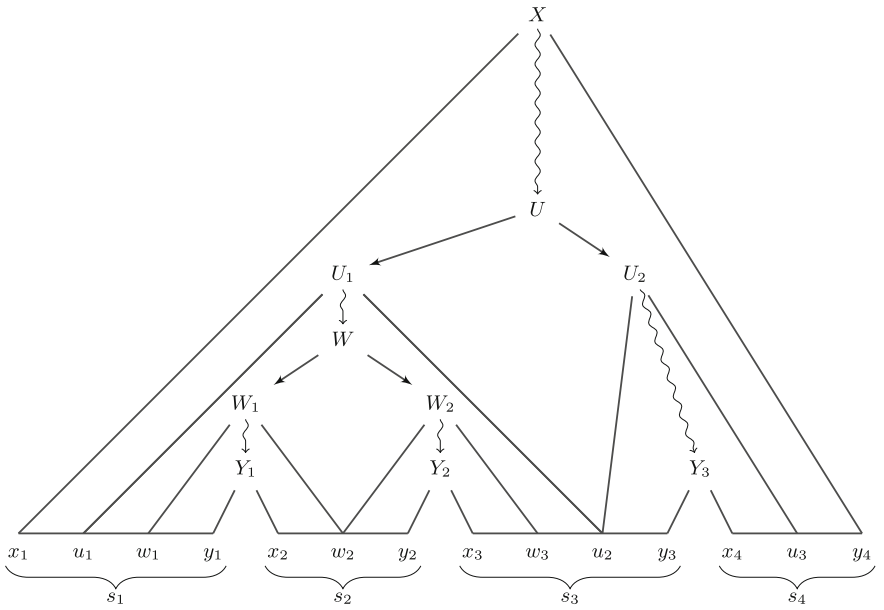

Fig. 2.1 Illustration of when a tuple $(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)$ is contained in $R_{X \to Y_1, Y_2, Y_3}$ in Lemma 2.5.7.

Y_2 is derived from W_2. Analogously for case (2). The derivation tree of X for case (1) is illustrated in Fig. 2.1.

The formula $\varphi_{X \to Y_1, Y_2, Y_3}$ is the disjunction of formulas ψ_1 and ψ_2, responsible for dealing with the cases (1) and (2) respectively. We only exhibit ψ_1, the formula ψ_2 can be constructed analogously. The formula ψ_1 guesses the variables U, U_1, U_2, W, W_1 and W_2, and the start and end positions of strings derived from those variables. Whether $(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)$ is contained in $R^{\mathcal{S}}_{X \to Y_1, Y_2, Y_3}$ can then be tested using the relations $R_{Z \to Z'}$. For simplicity the formula ψ_1 reuses the element names x_i and y_i as variable names and is defined as follows:

$$
\begin{aligned}
\psi(x_1, y_1, \ldots, x_4, y_4) = \exists u_1 \exists u_2 \exists u_3 \bigvee_{\substack{U, U_1, U_2 \in V \\ U \to U_1 U_2 \in P}} \exists w_1 \exists w_2 \exists w_3 \bigvee_{\substack{W, W_1, W_2 \in V \\ W \to W_1 W_2 \in P}} \\
\Big(R_{X \to U}(x_1, u_1, u_3, y_4) \wedge R_{U_1 \to W}(u_1, w_1, w_3, u_2) \\
\wedge R_{W_1 \to Y_1}(w_1, y_1, x_2, w_2) \wedge R_{W_2 \to Y_2}(w_2, y_2, x_3, w_3) \\
\wedge R_{U_2 \to Y_3}(u_2, y_3, x_4, u_3) \Big)
\end{aligned}
$$

□

We now use the relations $R_{Z \to Z'}$ and the formulas $\varphi_{X \to Y_1, Y_2, Y_3}$ for maintaining context-free path queries on acyclic graphs.

Proof sketch (of Proposition 2.5.6). Let L be an arbitrary context-free language and let $\mathcal{G} = (V, \Sigma, S, P)$ be a grammar for L in Chomsky normal form. We provide a DynFO-program $\mathcal{P}$ with designated binary query symbol Q that maintains L on acyclic graphs. The input schema is $\{E_\sigma \mid \sigma \in \Sigma\}$ and the auxiliary schema is $\tau_{\text{aux}} = \{R_{X\to Y} \mid X, Y \in V\} \cup \{T\}$. The intention of the auxiliary relation symbols $R_{X\to Y}$ has already been explained above; the relation symbol T shall store the transitive closure of the input graph (where the input graph is the union of all E_σ).

Before showing how to update the relations $R_{X\to Y}$, we state the update formulas for the query relation Q. The update formulas distinguish whether the witness path is of length 0 or of length at least 2. The updated relations $R_{X\to Y}$ are used for the latter case.

$$\begin{aligned}
\phi^{Q}_{\text{INS}_{E_\sigma}}(u, v; x, y) \stackrel{\text{def}}{=} &([S \to \epsilon \in P] \wedge x = y) \\
&\vee \exists z_1 \exists z_2 \bigvee_{\substack{U \in V \\ U \to \tau \in P}} \left(\phi^{R_{S\to U}}_{\text{INS}_{E_\sigma}}(u, v; x, z_1, z_2, y) \wedge E_\tau(z_1, z_2)\right) \\
\phi^{Q}_{\text{DEL}_{E_\sigma}}(u, v; x, y) \stackrel{\text{def}}{=} &([S \to \epsilon \in P] \wedge x = y) \\
&\vee \exists z_1 \exists z_2 \bigvee_{\substack{U \in V \\ U \to \tau \in P}} \left(\phi^{R_{S\to U}}_{\text{DEL}_{E_\sigma}}(u, v; x, z_1, z_2, y) \wedge E_\tau(z_1, z_2)\right)
\end{aligned}$$

It remains to present update formulas for each $R_{X\to Y}$. For simplicity we identify names of variable and elements.

After inserting a σ-edge (u, v), a tuple (x_1, y_1, x_2, y_2) is contained in $R_{X\to Y}$ if there are two witness paths p_1 and p_2 such that (1) p_1 and p_2 have already been witnesses before the insertion, or (2) only p_1 uses the new σ-edge, or (3) only p_2 uses the new σ-edge, or (4) both p_1 and p_2 use the new σ-edge. In case (2) the path p_1 can be split into a path from x_1 to u, the edge (u, v) and a path from v to y_1. Similarly in the other cases and for p_2. Using the formulas from Lemma 2.5.7 this can be expressed as follows:

$$\begin{aligned}
\phi^{R_{X\to Y}}_{\text{INS}_{E_\sigma}}(u, v; x_1, y_1, x_2, y_2) \stackrel{\text{def}}{=} &\; R_{X\to Y}(x_1, y_1, x_2, y_2) \vee && (2.1) \\
&\bigvee_{\substack{U_1, U_2 \in V \\ U_1 \to \sigma \in P \\ U_2 \to \sigma \in P}} \Big(\varphi_{X\to U_1, Y}(x_1, u, v, y_1, x_2, y_2) && (2.2) \\
&\qquad \vee \varphi_{X\to Y, U_2}(x_1, y_1, x_2, u, v, y_2) && (2.3) \\
&\qquad \vee \varphi_{X\to U_1, Y, U_2}(x_1, u, v, y_1, x_2, u, v, y_2)\Big) && (2.4)
\end{aligned}$$

After deleting a σ-edge (u, v) a tuple (x_1, y_1, x_2, y_2) is in $R_{X\to Y}$ if it still has witness paths p_1 and p_2 from x_1 to y_1 and from x_2 to y_2, respectively. The update formula for $R_{X\to Y}$ verifies that such witness paths exist. Therefore, similar to Example 2.3.2, the formula distinguishes for each $i \in \{1, 2\}$ whether (1) there was

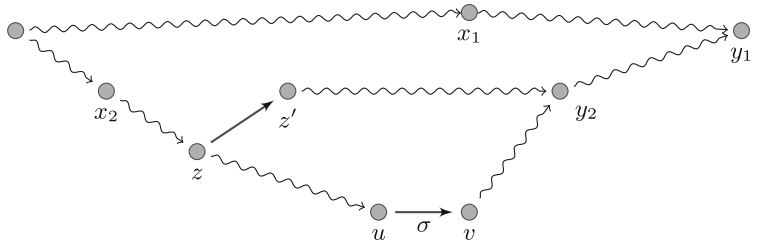

Fig. 2.2 Illustration of the update of $R_{X\to Y}$ after deletion of σ-edge (u, v) in the proof of Proposition 2.5.6. The nodes x_1 and y_1 satisfy Condition (1), whereas nodes x_2 and y_2 satisfy Condition (2).

no path from x_i to y_i via (u, v) before deleting the σ-edge (u, v), or (2) there was a path from x_i to y_i via (u, v). See Fig. 2.2 for an illustration.

In case (1) all paths present from x_i to y_i before the deletion of the σ-edge (u, v) are also present after the deletion. In particular the set of possible witnesses p_i remains the same. For case (2), the update formula has to check that there is still a witness path p_i. Such a path p_i has the options (a) to still use the edge (u, v) but for a $\tau \neq \sigma$, and (b) to not use the edge (u, v) at all.

The update formula for $R_{X\to Y}$ is a disjunction over all those cases for the witnesses for (x_1, y_1) and (x_2, y_2). Instead of presenting formulas for all those cases, we explain the idea for two representative cases. All other cases are analogous.

We first look at the case where (x_1, y_1) satisfies (1), (x_2, y_2) satisfies (2) and there are witness paths p_1 and p_2 where p_2 satisfies (a). The following formula deals with this case:

$$(\neg T(x_1, u) \vee \neg T(v, y_1)) \wedge T(x_2, u) \wedge T(v, y_2) \\ \wedge \bigvee_{\substack{\tau\neq\sigma, U_2\in V \\ U_2\to\tau\in P}} \big(\varphi_{X\to Y, U_2}(x_1, y_1, x_2, u, v, y_2) \wedge E_\tau(u, v)\big)$$

In the first line the premises for this case are checked, in the second line it is verified that p_2 uses τ-edge (u, v) for $\sigma \neq \tau$.

Now we consider the case where both (x_1, y_1) as well as (x_2, y_2) satisfy (2), and where there are witness paths p_1 and p_2 where p_1 satisfies (a) and p_2 satisfies (b). The existence of such a path p_1 can be verified as above. For verifying the existence of such a path p_2, a path not using (u, v) has to be found. This is achieved by relying on the same technique as for maintaining reachability for acyclic graphs (see Example 2.3.2). The following formula verifies the existence of such p_1 and p_2:

$$T(x_1, u) \wedge T(v, y_1) \wedge T(x_2, u) \wedge T(v, y_2)$$
$$\wedge \exists z \exists z' \bigvee_{\substack{\tau \neq \sigma, U_1, U_2 \in V \\ U_1 \to \tau \in P \\ U_2 \to \tau' \in P}} \Big(\varphi_{X \to U_1, Y, U_2}(x_1, u, v, y_1, x_2, z, z', y_2) \wedge E_\tau(u, v)$$
$$\wedge \big(T(x_2, z) \wedge E_{\tau'}(z, z') \wedge (z \neq u \vee z' \neq v)$$
$$\wedge T(z', y_2) \wedge T(z, u) \wedge \neg T(z', u) \big) \Big)$$

Again, in the first line the premises for this case are checked. In the second line z and z' are chosen with the purpose to find an alternative path p_2 (as in Example 2.3.2), and it is verified that p_1 and p_2 are witness paths. The third and forth lines verify that z and z' yield an alternative path. □

Finally we exhibit examples of non-context-free languages that can be maintained in restricted settings.

Proposition 2.5.8. *There is a non-context-free path query that can be maintained from scratch in* DynFO *on acyclic graphs.*

Proof. Let $L \stackrel{\text{def}}{=} \{a^n b^n c^n \mid n \in \mathbb{N}\}$. We provide a DynFO-program $\mathcal{P} = (P, \text{Init}, Q)$ that maintains L on acyclic graphs. The input schema τ_{inp} contains a binary relation symbol E_σ for each $\sigma \in \{a, b, c\}$.

We assume that arithmetic is available on the elements that have been used in modifications so far, that is, that there is a linear order relation $<$ as well as its corresponding addition relation $+$ on those elements. In [Ete98], Etessami showed that basic arithmetic relations can be maintained by a DynFO-program (see also Sect. 3.4). The linear order $<$ can be used to interpret elements of the active domain as numbers, i.e. an element x is interpreted as the number k if x is the kth element with respect to $<$.

For each $\sigma \in \{a, b, c\}$ the program $\mathcal{P}$ has a binary auxiliary relation T_σ and a ternary auxiliary relation D_σ. The intention is as follows. In a given state $\mathcal{S}$, the relation $T_\sigma^{\mathcal{S}}$ shall contain the transitive closure of $E_\sigma^{\mathcal{S}}$; and the relation $D_\sigma^{\mathcal{S}}$ shall contain a tuple (x, y, k) if and only if there is a σ^*-path from x to y of length k.

Already in Example 2.3.2 we have seen how the transitive closure of an acyclic graph can be maintained in DynFO. Thus the relations T_σ can be easily maintained. Their update formulas can be easily extended to also keep track of the length of paths:

$$\phi^{D_\sigma}_{\mathrm{INS}_{E_\sigma}}(u,v;x,y,k) \stackrel{\text{def}}{=} D_\sigma(x,y,k) \vee \exists\ell\exists\ell'\big(\ell+\ell'+1=k \wedge D_\sigma(x,u,\ell) \wedge D_\sigma(v,y,\ell')\big)$$

$$\begin{aligned}\phi^{D_\sigma}_{\mathrm{DEL}_{E_\sigma}}(u,v;x,y,k) \stackrel{\text{def}}{=}\ & D_\sigma(x,y,k) \wedge \Big(\big(\neg T_\sigma(x,u) \vee \neg T_\sigma(v,y)\big) \\ & \vee \exists z\exists z'\exists\ell\exists\ell'\big(\ell+\ell'+1=k \wedge D_\sigma(x,z,\ell) \wedge E_\sigma(z,z') \\ & \wedge (z \neq u \vee z' \neq v) \wedge D_\sigma(z',y,\ell') \\ & \wedge T_\sigma(z,u) \wedge \neg T_\sigma(z',u)\big)\Big)\end{aligned}$$

For updates to $E_{\sigma'}$ for $\sigma' \neq \sigma$, the relations D_σ and T_σ remain unchanged. Finally, whether there is an L-path between to nodes after an update δ can be expressed by the following formulas:

$$\phi^{Q}_{\delta}(u,v;x,y) \stackrel{\text{def}}{=} \exists z\exists z'\exists k\big(\phi^{D_a}_{\delta}(u,v;x,z,k) \wedge \phi^{D_b}_{\delta}(u,v;z,z',k) \wedge \phi^{D_c}_{\delta}(u,v;z',y,k)\big) \qquad \square$$

Proposition 2.5.9. *There is a non-context-free path query that can be maintained from scratch in* DynFO *when only insertions are allowed.*

Proof sketch. Consider the language $L \stackrel{\text{def}}{=} \{a^{n!+n} \mid n \in \mathbb{N}\}$. Observe that L is not context-free (because its Parikh image is not semi-linear). Furthermore L has the property that if there is a path from x to y containing a loop, then there is an L-path from x to y. To see this, let p be a path from x to y containing a loop of length ℓ, and let k be the length of the path p without the loop. Then there is an a^*-path of length $m\ell + k$ between x and y for every m, since the loop can be repeated m many times. Now, choosing $n \stackrel{\text{def}}{=} k\ell + k$ shows that there is an L-path from x to y since there is an m such that $m\ell + k = (k\ell + k)! + k\ell + k$ (because ℓ divides $(k\ell + k)! + k\ell$).

Thus, in order to maintain L under insertions, it is sufficient to maintain for all nodes x and y auxiliary data that indicates (1) whether there is a path with a loop between x and y, and (2) the lengths of loop-free paths between x and y.

This can be achieved by using the technique used for maintaining $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ in the acyclic case. Hence, again, the dynamic program maintains a linear order $<$, as well as its corresponding addition and multiplication relations on the elements that have been used in modifications so far [Ete98]. Further, the program maintains a relation D which stores all tuples (x, y, k) for which there is an a^*-path from x to y of length k.

This suffices to check condition (1). For checking (2), additionally a unary relation F is maintained, which is supposed to store all elements k such that $k = m! + m$ for some $m \in \mathbb{N}$. This can be easily achieved by extending Etessami's construction.

Now, when an a-edge (u, v) is added, then there is an L-path between x and y if there was such a path before; or there is a node z such that there are a^*-paths from x to z, from z to z and from z to y; or there is an a^*-path from x to y of length k where k is in F. This can be easily expressed by a first-order update formula. $\square$

2.6 Outlook and Bibliographic Remarks

In this chapter the basic dynamic complexity framework and the basic dynamic complexity classes have been introduced. We presented examples for the power of dynamic programs and discussed variants of the framework used in the literature. Furthermore several small results for maintaining queries on graph databases have been obtained.

The most promising direction for future work is to conduct a deeper, more systematic study of graph queries. Here, we only looked at very basic graph query languages. Extensions, such as the query language ECRPQ [BLLW12], might be worth studying as well. As an example, the length of paths used in a conjunctive regular path query can be compared in ECRPQ. It is conceivable that such queries can be maintained in DynFO. This would also settle the open question whether a context-free path query can be maintained in DynFO when both insertions and deletions are allowed.

As a minimal prerequisite for maintaining such queries, it is necessary to be able to maintain distances in a graph. Extending the result that reachability is in DynFO in this direction seems not to be trivial, but also not out of reach.

Another possible research direction is motivated from the discussion of the framework. The discussion highlighted that many different settings have been looked at and that some variants actually collapse. In order to be able to transfer results easily between different settings, it would be very convenient to have a systematic overview of how the settings relate to each other. This is, however, probably not easy to achieve.

Bibliographical Remarks

The dynamic complexity framework presented in Sect. 2.2 has originally been introduced by Patnaik and Immerman in [PI94]. In this form it was introduced in joint work with Thomas Schwentick in [ZS13], and reused in [ZS14, Zeu14a]. A variant of the class DynFO was introduced as Dyn- FO in [PI94]; the classes DynProp and DynQF have been introduced in Hesse's thesis [Hes03b]. The examples for those classes are attributed as follows. Example 2.3.2 has been independently presented in [DS95][1] and [PI94]. The technique used in Example 2.3.4 has been introduced in [GMS09], the example itself is from [ZS13]. Example 2.3.7 will also be contained in the full versions of [ZS14, Zeu14a].

With the only exception of Theorems 2.5.1 and 2.5.2 and Proposition 2.5.9, the results for graph queries are solely by the author and have not been published before. Theorems 2.5.1 and 2.5.2 are joint work with Samir Datta, Raghav Kulkarni, Anish Mukherjee and Thomas Schwentick [DKM+15]. Proposition 2.5.9 originated from related discussions with Katja Losemann.

[1] In the cited work the result is attributed to earlier work by the same authors from 1993. I was not able to obtain this previous work.

http://www.springer.com/978-3-662-54313-9

Small Dynamic Complexity Classes
An Investigation into Dynamic Descriptive Complexity
Zeume, Th.
2017, VIII, 149 p. 17 illus., Softcover
ISBN: 978-3-662-54313-9