Static Analysis for Logic-based Dynamic Programs*

Thomas Schwentick, Nils Vortmeier, and Thomas Zeume

TU Dortmund University
Germany
{thomas.schwentick,nils.vortmeier,thomas.zeume}@tu-dortmund.de

Abstract

A dynamic program, as introduced by Patnaik and Immerman (1994), maintains the result of a fixed query for an input database which is subject to tuple insertions and deletions. It can use an auxiliary database whose relations are updated via first-order formulas upon modifications of the input database. This paper studies static analysis problems for dynamic programs and investigates, more specifically, the decidability of the following three questions. Is the answer relation of a given dynamic program always empty? Does a program actually maintain a query? Is the content of auxiliary relations independent of the modification sequence that lead to an input database? In general, all these problems can easily be seen to be undecidable for full first-order programs. Therefore the paper aims at pinpointing the exact decidability borderline for programs with restricted arity (of the input and/or auxiliary database) and restricted quantification.

1998 ACM Subject Classification F.4.1. Mathematical Logic

Keywords and phrases Dynamic descriptive complexity, algorithmic problems, emptiness, history independence, consistency

Digital Object Identifier 10.4230/LIPIcs.CSL.2015.308

1 Introduction

In modern database scenarios data is subject to frequent changes. In order to avoid costly re-computation of queries from scratch after each small modification of the data, one can try to use previously computed auxiliary data. This auxiliary data then needs to be updated dynamically whenever the database changes.

The descriptive dynamic complexity framework (short: dynamic complexity) by Patnaik and Immerman [19] models this setting from a declarative perspective. It was mainly inspired by updates in relational databases. Within this framework, for a relational database subject to change, a *dynamic program* maintains auxiliary relations with the intention to help answering a query $\mathcal{Q}$. When a modification to the database, that is an insertion or deletion of a tuple, occurs, every auxiliary relation is updated through a first-order update formula (or, equivalently, through a core SQL query) that can refer to the database as well as to the auxiliary relations. The result of $\mathcal{Q}$ is, at every time, represented by some distinguished auxiliary relation. The class of all queries maintainable by dynamic programs with first-order update formulas is called DynFO and we refer to such programs as DynFO-programs. We note that shortly before the work of Patnaik and Immerman, the declarative approach was independently formalized in a similar way by Dong, Su and Topor [6].

* The first and third author acknowledge the financial support by DFG grant SCHW 678/6-1.

The main question studied in Dynamic Complexity has been which queries that are not statically expressible in first-order logic (and therefore not in Core SQL), can be maintained by DynFO-programs. Recently, it has been shown that the Reachability query, a very natural such query, can be maintained by DynFO programs [1]. Altogether, research in Dynamic Complexity succeeded in proving that many non-FO queries are maintainable in DynFO. These results and their underlying techniques yield many interesting insights into the the nature of Dynamic Complexity.

However, to complete the understanding of Dynamic Complexity, it would be desirable to complement these techniques by methods for proving that certain queries are *not* maintainable by DynFO programs. But the state of the art with respect to inexpressibility results is much less favorable: at this point, no general techniques for showing that a query is not expressible in DynFO are available. In order to get a better overall picture of Dynamic Complexity in general and to develop methods for inexpressibility proofs in particular, various restrictions of DynFO have been studied, based on, e.g., arity restrictions for the auxiliary relations [2, 5, 3], fragments of first-order logic [12, 10, 25, 23], or by other means [4, 11].

At the heart of our difficulties to prove inexpressibility results in Dynamic Complexity is our limited understanding of what dynamic programs with or without restrictions "can do" in general, and our limited ability to analyze what a particular dynamic program at hand "does". In this paper, we initiate a systematic study of the "analyzability" of dynamic programs. Static analysis of queries has a long tradition in Database Theory and we follow this tradition by first studying the emptiness problem for dynamic programs, that is the question, whether there exists an initial database and a modification sequence that is accepted by a given dynamic program.[1] Given the well-known undecidability of the finite satisfiability problem for first-order logic [21], it is not surprising that emptiness of DynFO programs is undecidable in general. However, we try to pinpoint the borderline of undecidability for fragments of DynFO based on restrictions of the arity of input relations, the arity of auxiliary relations and for the class DynProp of programs with quantifier-free update formulas.

In the fragments where undecidability of emptiness does not directly follow from undecidability of satisfiability in the corresponding fragment of first-order logic, our undecidability proofs make use of dynamic programs whose query answer might not only depend on the database yielded by a certain modification sequence, but also on the sequence itself, that is, on the order in which tuples are inserted or (even) deleted. From a useful dynamic program one would, of course, expect that it is *consistent* in the sense that its query answer always only depends on the current database, but not on the specific modification sequence by which it has been obtained. It turns out that the emptiness problem for consistent programs is easier than the general emptiness problem for dynamic programs. More precisely, there are fragments of DynFO, for which an algorithm can decide emptiness for dynamic programs that come with a "consistency guarantee", but for which the emptiness problem is undecidable, in general. However, it turns out that the combination of a consistency test with an emptiness test for consistent programs does not gain any advantage over "direct" emptiness tests, since the consistency problem turns out to be as difficult as the general emptiness problem.

Finally, we study a property that many dynamic programs in the literature share: they are *history independent* in the sense that all auxiliary relations always only depend on the

[1] The exact framework will be defined in Section 3, but we already mention that we will consider the setting in which databases are initially empty and the auxiliary relations are defined by first-order formulas.

Table 1 Summary of the results of this paper. DynFO(ℓ-in, m-aux) stands for DynFO-programs with (at most) ℓ-ary input relations and m-ary auxiliary relations. DynFO(m-aux) and DynFO(ℓ-in) represent programs with m-ary auxiliary relations (and arbitrary input relations) and programs with ℓ-ary input relations, respectively. Likewise for DynProp.

	Emptiness Consistency	Emptiness for consistent programs	History Independence
Undecidable	DynFO(1-in, 0-aux) DynProp(2-in, 0-aux) DynProp(1-in, 2-aux)	DynFO(1-in, 2-aux) DynFO(2-in, 0-aux)	DynFO(2-in, 0-aux)
Decidable	DynProp(1-in, 1-aux)	DynFO(1-in, 1-aux) DynProp(1-in) DynProp(1-aux)	DynFO(1-in) DynProp(1-aux)
Open		DynProp(2-in, 2-aux) and beyond	DynProp(2-in, 2-aux) and beyond

current (input) database. History independence can be seen as a strong form of consistency in that it not only requires the query relation, but *all* auxiliary relations to be determined by the input database. History independent dynamic programs (also called *memoryless* [19] or *deterministic* [4]) are still expressive enough to maintain interesting queries like undirected reachability [11]. But also some inexpressibility proofs have been found for such programs [4, 11, 25]. We study the *history independence problem*, that is, whether a given dynamic program is history independent. In a nutshell, the history independence problem is the "easiest" of the static analysis problems considered in this paper.

Our results, summarized in Table 1, shed light on the borderline between decidable and undecidable fragments of DynFO with respect to emptiness (and consistency), emptiness for consistent programs and history independence. While the picture is quite complete for the emptiness problem for general dynamic programs, for some fragments of DynProp there remain open questions regarding the emptiness problem for consistent dynamic programs and the history-independence problem. Some of the results shown in this paper have been already presented in the master's thesis of Nils Vortmeier [22].

Outline. We recall some basic definitions in Section 2 and introduce the formal setting in Section 3. The emptiness problem is defined and studied in Section 4, where we first consider general dynamic programs (Subsection 4.1) and then consistent dynamic programs (Subsection 4.2). In Subsection 4.3 we briefly discuss the impact of built-in orders to the results. The Consistency and History Independence problems are studied in Sections 5 and 6, respectively. We conclude in Section 7. Due to the space limit we only give proof sketches or even proof ideas in the body of this paper. Complete proofs can be found in the full version [20].

2 Preliminaries

We presume that the reader is familiar with basic notions from Finite Model Theory and refer to [8, 16] for a detailed introduction into this field. We review some basic definitions in order to fix notations.

In this paper, a *domain* is a non-empty finite set. For tuples $\vec{a} = (a_1, \ldots, a_k)$ and $\vec{b} = (b_1, \ldots, b_\ell)$ over some domain D, the $(k + \ell)$-tuple obtained by concatenating $\vec{a}$ and $\vec{b}$ is denoted by $(\vec{a}, \vec{b})$.

A (relational) *schema* is a collection τ of relation symbols[2] together with an arity function $\mathrm{Ar} : \tau \to \mathbb{N}$. A *database* $\mathcal{D}$ with schema τ and domain D is a mapping that assigns to every relation symbol $R \in \tau$ a relation of arity $\mathrm{Ar}(R)$ over D. The *size of a database*, usually denoted by n, is the size of its domain. We call a database *empty*, if all its relations are empty. We emphasize that empty databases have non-empty domains. A *τ-structure* $\mathcal{S}$ is a pair $(D, \mathcal{D})$ where $\mathcal{D}$ is a database with schema τ and domain D. Often we omit the schema when it is clear from the context.

We write $\mathcal{S} \models \varphi(\vec{a})$ if the first-order formula $\varphi(\vec{x})$ holds in $\mathcal{S}$ under the variable assignment that maps $\vec{x}$ to $\vec{a}$. The *quantifier depth* of a first-order formula is the maximal nesting depth of quantifiers. The *rank-q type* of a tuple $(a_1, \ldots, a_m)$ with respect to a τ-structure $\mathcal{S}$ is the set of all first-order formulas $\varphi(x_1, \ldots, x_m)$ (with equality) of quantifier depth at most q, for which $\mathcal{S} \models \varphi(\vec{a})$ holds. By $\mathcal{S} \equiv_q \mathcal{S}'$ we denote that two structures $\mathcal{S}$ and $\mathcal{S}'$ have the same rank-q type (of length 0 tuples).

For a subschema $\tau' \subseteq \tau$, the rank-q τ'-type of a tuple $\vec{a}$ in a τ-structure $\mathcal{S}$ is its rank-q type in the τ'-reduct of $\mathcal{S}$.

We refer to the rank-0 type of a tuple also as its *atomic type* and, since we mostly deal with rank-0 types, simply as its *type*. The *equality type* of a tuple is the atomic type with respect to the empty schema.

The *k-ary type* of a tuple $\vec{a}$ in a structure $\mathcal{S}$ is its $\tau_{\leq k}$-type, where $\tau_{\leq k}$ consists of all relation symbols of τ with arity at most k. The *τ'-color* of an element a in $\mathcal{S}$, for a subschema τ' of the schema of $\mathcal{S}$, is its τ'_1-type, where τ'_1 consists of all unary relation symbols of τ'. We often enumerate the possible τ'-colors as $c_0, \ldots, c_L$, for some L with c_0 being the color of elements that are in neither of the unary relations. We call these elements *τ'-uncolored*. If τ' is clear from the context we simply speak of colors and uncolored elements.

3 The dynamic complexity setting

For a database $\mathcal{D}$ over schema τ, a *modification* $\delta = (o, \vec{a})$ consists of an operation $o \in \{\mathrm{INS}_S, \mathrm{DEL}_S \mid S \in \tau\}$ and a tuple $\vec{a}$ of elements from the domain of $\mathcal{D}$. By $\delta(\mathcal{D})$ we denote the result of applying δ to $\mathcal{D}$ with the obvious semantics of inserting or deleting the tuple $\vec{a}$ to or from relation $S^{\mathcal{D}}$. For a sequence $\alpha = \delta_1 \cdots \delta_N$ of modifications to a database $\mathcal{D}$ we let $\alpha(\mathcal{D}) \stackrel{\text{def}}{=} \delta_N(\cdots(\delta_1(\mathcal{D}))\cdots)$.

A *dynamic instance*[3] of a query $\mathcal{Q}$ is a pair $(\mathcal{D}, \alpha)$, where $\mathcal{D}$ is a database over a domain D and α is a sequence of modifications to $\mathcal{D}$. The dynamic query $\mathrm{DYN}(\mathcal{Q})$ yields the result of evaluating the query $\mathcal{Q}$ on $\alpha(\mathcal{D})$.

Dynamic programs, to be defined next, consist of an initialization mechanism and an update program. The former yields, for every (initial) database $\mathcal{D}$, an initial state with initial auxiliary data. The latter defines the new state of the dynamic program for each possible modification δ.

A *dynamic schema* is a pair $(\tau_{\text{in}}, \tau_{\text{aux}})$, where τ_{in} and τ_{aux} are the schemas of the input database and the auxiliary database, respectively. We call relations over τ_{in} *input relations* and relations over τ_{aux} *auxiliary relations*. If the relations are 0-ary, we also speak of input or auxiliary *bits*. We always let $\tau \stackrel{\text{def}}{=} \tau_{\text{in}} \cup \tau_{\text{aux}}$.

[2] For simplicity we do not allow constants in this work but note that our results hold for relational schemas with constants as well.

[3] The following introduction to dynamic descriptive complexity is similar to previous work [25, 24].

▶ **Definition 1** (Update program). An *update program* P over a dynamic schema $(\tau_{\text{in}}, \tau_{\text{aux}})$ is a set of first-order formulas (called *update formulas* in the following) that contains, for every $R \in \tau_{\text{aux}}$ and every $o \in \{\text{INS}_S, \text{DEL}_S \mid S \in \tau_{\text{in}}\}$, an update formula $\phi_o^R(\vec{x}; \vec{y})$ over the schema τ where $\vec{x}$ and $\vec{y}$ have the same arity as S and R, respectively.

A *program state* $\mathcal{S}$ over dynamic schema $(\tau_{\text{in}}, \tau_{\text{aux}})$ is a structure $(D, \mathcal{I}, \mathcal{A})$ where[4] D is a finite domain, $\mathcal{I}$ is a database over the input schema (the *input database*) and $\mathcal{A}$ is a database over the auxiliary schema (the *auxiliary database*). The *semantics of update programs* is as follows. For a modification $\delta = (o, \vec{a})$, where $\vec{a}$ is a tuple over D, and 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 $\mathcal{A}'$ consists of relations $R^{\mathcal{A}'} \stackrel{\text{def}}{=} \{\vec{b} \mid \mathcal{S} \models \phi_o^R(\vec{a}; \vec{b})\}$. The effect $P_\alpha(\mathcal{S})$ of a modification sequence $\alpha = \delta_1 \dots \delta_N$ to a state $\mathcal{S}$ is the state $P_{\delta_N}(\dots(P_{\delta_1}(\mathcal{S}))\dots)$.

▶ **Definition 2** (Dynamic program). A *dynamic program* is a triple $(P, \text{Init}, R_{\mathcal{Q}})$, where
- P is an update program over some dynamic schema $(\tau_{\text{in}}, \tau_{\text{aux}})$,
- Init is a mapping that maps τ_{in}-databases to τ_{aux}-databases, and
- $R_{\mathcal{Q}} \in \tau_{\text{aux}}$ is a designated *query symbol*.

A dynamic program $\mathcal{P} = (P, \text{Init}, R_{\mathcal{Q}})$ *maintains* a dynamic query $\text{Dyn}(\mathcal{Q})$ if, for every dynamic instance $(\mathcal{D}, \alpha)$, the query result $\mathcal{Q}(\alpha(\mathcal{D}))$ coincides with the query relation $R_{\mathcal{Q}}^{\mathcal{S}}$ in the state $\mathcal{S} = P_\alpha(\mathcal{S}_{\text{Init}}(\mathcal{D}))$, where $\mathcal{S}_{\text{Init}}(\mathcal{D}) \stackrel{\text{def}}{=} (D, \mathcal{D}, \text{Init}(\mathcal{D}))$ is the initial state for $\mathcal{D}$. If the query relation $R_{\mathcal{Q}}$ is 0-ary, we often denote this relation as *query bit* Acc and say that $\mathcal{P}$ *accepts* α over D if Acc is true in $P_\alpha(\mathcal{S}_{\text{Init}}(\mathcal{D}))$.

In the following, we write $\mathcal{P}_\alpha(\mathcal{D})$ instead of $P_\alpha(\mathcal{S}_{\text{Init}}(\mathcal{D}))$ and $\mathcal{P}_\alpha(\mathcal{S})$ instead[5] of $P_\alpha(\mathcal{S})$ for a given dynamic program $\mathcal{P} = (P, \text{Init}, R_{\mathcal{Q}})$, a modification sequence α, an initial database $\mathcal{D}$ and a state $\mathcal{S}$.

▶ **Definition 3** (DynFO and DynProp). DynFO is the class of all dynamic queries that can be maintained by dynamic programs with first-order update formulas and first-order definable initialization mapping when starting from an initially empty input database. DynProp is the subclass of DynFO, where update formulas are quantifier-free[6].

A DynFO-program is a dynamic program with first-order update formulas, likewise a DynProp-program is a dynamic program with quantifier-free update formulas. A DynFO(ℓ-in, m-aux)-program is a DynFO-program over (at most) ℓ-ary input databases that uses auxiliary relations of arity at most m; likewise for DynProp(ℓ-in, m-aux)-programs.[7]

Due to the undecidability of finite satisfiability of first-order logic, the emptiness problem – the problem we study first – is undecidable even for DynFO-programs with only a single auxiliary relation (more precisely, with query bit only). Therefore, we restrict our investigations to fragments of DynFO. Also allowing arbitrary initialization mappings immediately yields an undecidable emptiness problem. This is already the case for first-order definable initialization mappings for arbitrary initial databases. In the literature classes with various restricted and unrestricted initialization mappings have been studied, see [24] for a discussion. In this work, in line with [19], we allow initialization mappings defined by arbitrary first-order formulas, but require that the initial database is empty. Of course, we could have studied

[4] We prefer the notation $(D, \mathcal{I}, \mathcal{A})$ over $(D, \mathcal{I} \cup \mathcal{A})$ to emphasize the two components of the overall database.
[5] The notational difference is tiny here: we refer to the dynamic program instead of the update program.
[6] We still allow the use of quantifiers for the initialization.
[7] We do not consider the case $\ell = 0$ where databases are pure sets with a fixed number of bits.

further restrictions on the power of the initialization formulas, but this would have yielded a setting with an additional parameter.

The following example illustrates a technique to maintain lists with quantifier-free dynamic programs, introduced in [10, Proposition 4.5], which is used in some of our proofs. The example itself is from [25].

▶ **Example 4.** We provide a DynProp-program $\mathcal{P}$ for the dynamic variant of the Boolean query NonEmptySet, where, for a unary relation U subject to insertions and deletions of elements, one asks whether U is empty. Of course, this query is trivially expressible in first-order logic, but not without quantifiers.

The program $\mathcal{P}$ is over auxiliary schema $\tau_{\text{aux}} = \{R_Q, \textsc{First}, \textsc{Last}, \textsc{List}\}$, where R_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 of $\mathcal{P}$ is to maintain a list of all elements currently in U. The list structure is stored in the binary relation $\textsc{List}^{\mathcal{S}}$. The first and last element of the list are stored in $\textsc{First}^{\mathcal{S}}$ and $\textsc{Last}^{\mathcal{S}}$, 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 U.

For a given instance of NonEmptySet the initialization mapping initializes the auxiliary relations accordingly.We only describe the (more complicated) case of deletions from U.

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.[8]

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

◀

In some parts of the paper we will use specific forms of modification sequences. An *insertion sequence* is a modification sequence $\alpha = \delta_1 \cdots \delta_m$ whose modifications are pairwise distinct insertions. An insertion sequence α over a unary input schema τ_{in} is in *normal form* if it fulfills the following two conditions.

(N1) For each element a, the insertions affecting a form a contiguous subsequence α_a of α. We say that α_a *colors* a.

(N2) For all elements a, b that get assigned the same τ_{in}-color by α, the projections of the subsequences α_a and α_b to their operations (i.e., their first parameters) are identical.

4 The Emptiness Problem

In this section we define and study the decidability of the emptiness problem for dynamic programs in general and for restricted classes of dynamic programs. The emptiness problem asks, whether the query relation R_Q of a given dynamic program $\mathcal{P}$ is always empty, more precisely, whether $R^{\mathcal{S}}_Q = \emptyset$ for every (empty) initial database $\mathcal{D}$ and every modification sequence α with $\mathcal{S} = \mathcal{P}_\alpha(\mathcal{D})$.

[8] We omit the (obvious) parts of formulas that deal with spurious deletions.

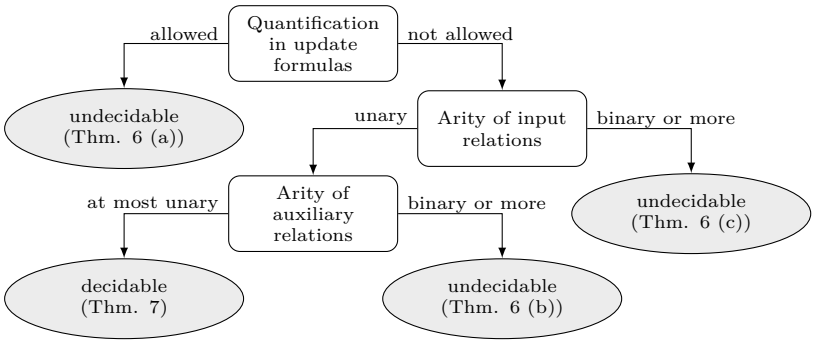

Figure 1 Decidability of EMPTINESS for various classes of dynamic programs.

To enable a fine-grained analysis, we parameterize the emptiness problem by a class $\mathcal{C}$ of dynamic programs.

Problem:	EMPTINESS($\mathcal{C}$)
Input:	A dynamic program $\mathcal{P} \in \mathcal{C}$ with FO initialization
Question:	Is $R_Q^{\mathcal{S}} = \emptyset$, for every initially empty database $\mathcal{D}$ and every modification sequence α, where $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}_\alpha(\mathcal{D})$?

As mentioned before, undecidability of the emptiness problem for unrestricted dynamic programs follows immediately from the undecidability of finite satisfiability of first-order logic.

▶ **Theorem 5.** EMPTINESS *is undecidable for* DYNFO(2-*in*, 0-*aux*)*-programs.*

In the remainder of this section, we will shed some light on the border line between decidable and undecidable fragments of DYNFO. In Subsection 4.1 we study fragments of DYNFO obtained by disallowing quantification and/or restricting the arity of input and auxiliary relations. In Subsection 4.2, we consider dynamic programs that come with a certain consistency guarantee.

4.1 Emptiness of general dynamic programs

In this subsection we study the emptiness problem for various restricted classes of dynamic programs. We will see that the problem is basically only decidable if all relations are at most unary and no quantification in update formulas is allowed. Figure 1 summarizes the results.

At first we strengthen the general result from Theorem 5. We show that undecidability of the emptiness problem for DYNFO-programs holds even for unary input relations and auxiliary bits. Furthermore, quantification is not needed to yield undecidability: for DYNPROP-programs, emptiness is undecidable for binary input or auxiliary relations.

▶ **Theorem 6.** *The emptiness problem is undecidable for*
(a) DYNFO(1-*in*, 0-*aux*)*-programs,*
(b) DYNPROP(1-*in*, 2-*aux*)*-programs,*
(c) DYNPROP(2-*in*, 0-*aux*)*-programs,*

Proof sketch. In all three cases, the proof is by a reduction from the emptiness problem for semi-deterministic 2-counter automata.

In a nutshell, a counter automaton (short: CA) is a finite automaton that is equipped with counters that range over the non-negative integer numbers. A counter c can be incremented

(inc(c)), decremented (dec(c)) and tested for zero (ifzero(c)). A CA does not read any input (i.e., its transitions can be considered to be ϵ-transitions) and in each step it can manipulate or test one counter and transit from one state to another state. More formally, a CA is tuple (Q, C, Δ, q_i, F), where Q is a set of states, $q_i \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and C is a finite set (the *counters*). The transition relation Δ is a subset of $Q \times \{\text{inc}(c), \text{dec}(c), \text{ifzero}(c) \mid c \in C\} \times Q$. A *configuration* of a CA is a pair $(p, \vec{n})$ where p is a state and $\vec{n} \in \mathbb{N}^C$ gives a value n_c for each counter c in C. A transition $(p, \text{inc}(c), q)$ can be applied in state p, transits to state q and increments n_c by one. A transition $(p, \text{dec}(c), q)$ can be applied in state p if $n_c > 0$, transits to state q and decrements n_c by one. A transition $(p, \text{ifzero}(c), q)$ can be applied in state p, if $n_c = 0$ and transits to state q.

A CA is *semi-deterministic* if from every state there is either at most one transition or there are two transitions, one decrementing and one testing the same counter for zero. The emptiness problem for (semi-deterministic) 2-counter automata (2CA) asks whether a given counter automaton with two counters has an accepting run and is undecidable [18, Theorem 14.1-1].

In all three reductions, the dynamic program $\mathcal{P}$ is constructed such that for every run ρ of a semi-deterministic 2CA $\mathcal{M}$ there is a modification sequence $\alpha = \alpha(\rho)$ that lets $\mathcal{P}$ simulate ρ, and an empty database $\mathcal{D}$, such that $\mathcal{P}$ accepts α over $\mathcal{D}$ if and only if ρ is accepting. More precisely, the states of $\mathcal{P}$ encode the states of $\mathcal{M}$ by auxiliary bits and the counters of $\mathcal{M}$ in some way that differs in the three cases. However, in all cases it holds that not every modification sequence for $\mathcal{P}$ corresponds to a run of $\mathcal{M}$. However, $\mathcal{P}$ can detect if α does *not* correspond to a run and assume a rejecting sink state as soon as this happens.

For (a), the two counters are simply represented by two unary relations, such that the number of elements in a relation is the current value of the counter. The test whether a counter has value zero thus boils down to testing emptiness of a set and can easily be expressed by a formula with quantifiers.

The lack of quantifiers makes the reductions for (b) and (c) a bit more complicated. In both cases, the counters are represented by linked lists, similar to Example 4, and the number of elements in the list corresponds to the counter value (in (c): plus 1). With such a list a counter value zero can be detected without quantification. Due to the allowed relation types, the lists are built with auxiliary relations in (b) and with input relations in (c). ◀

The next result shows that emptiness of DynProp(1-in, 1-aux)-programs is decidable, yielding a clean boundary between decidable and undecidable fragments.

▶ **Theorem 7.** Emptiness *is decidable for* DynProp(1-*in*, 1-*aux*)-*programs.*

Proof. The proof uses the following two simple observations about DynProp(1-in, 1-aux)-programs $\mathcal{P}$.

- The initialization formulas of $\mathcal{P}$ assign the same τ_{aux}-color to all elements. This color and the initial auxiliary bits only depend on the size of the domain. Furthermore there is a number $n(\mathcal{P})$, depending solely on the initialization formulas, such that the initial auxiliary bits and τ_{aux}-colors are the same for all empty databases with at least $n(\mathcal{P})$ elements. This observation actually also holds for DynFO(1-in, 1-aux)-programs.
- When $\mathcal{P}$ reacts to a modification $\delta = (o, a)$, the new (τ-)color of an element $b \neq a$ only depends on o, the old color of b, the old color of a, and the 0-ary relations. In particular, if two elements b_1, b_2 (different from a) have the same color before the update, they both have the same new color after the update. Thus, the overall update basically consists of assigning new colors to each color (for all elements except a), and the appropriate handling of the element a and the 0-ary relations.

We will show below that the behavior of DynProp(1-in, 1-aux)-programs can be simulated by an automaton model with a decidable emptiness problem, which we introduce next.

A *multicounter automaton* (short: MCA) is a counter automaton which is not allowed to test whether a counter is zero, i.e. the transition relation Δ is a subset of $Q \times \{\mathrm{inc}(c), \mathrm{dec}(c) \mid c \in C\} \times Q$. A *transfer multicounter automaton* (short: TMCA) is a multicounter counter automaton which has, in addition to the increment and the decrement operation, an operation that simultaneously transfers the content of each counter to another counter. More precisely the transition relation Δ is a subset of $Q \times (\{\mathrm{inc}(c), \mathrm{dec}(c) \mid c \in C\} \cup \{t \mid t : C \to C\}) \times Q$. Applying a transition (p, t, q) to a configuration $(p, \vec{n})$ yields a configuration $(q, \vec{n}')$ with $n'_c \stackrel{\text{def}}{=} \sum_{t(d)=c} n_d$ for every $c \in C$. A configuration $(q, \vec{n})$ of a TCMA is *accepting*, if $q \in F$. The emptiness problem for TCMAs[9] is decidable by reduction to the coverability problem for transfer petri nets[10] which is known to be decidable [7].

Let $\mathcal{P}$ be a DynProp-program over unary schema $\tau = \tau_{\text{in}} \cup \tau_{\text{aux}}$ with query symbol $R_{\mathcal{Q}}$ which may be 0-ary or unary. Let Γ_0 be the set of all 0-ary (atomic) types over τ and let Γ_1 be the set of τ-colors. We construct a transfer multicounter automaton $\mathcal{M}$ with counter set $Z_1 = \{z_\gamma \mid \gamma \in \Gamma_1\}$. The state set Q of $\mathcal{M}$ contains Γ_0, the only accepting state f and some further "intermediate" states to be specified below.

The intuition is that whenever $\mathcal{P}$ can reach a state $\mathcal{S}$ then $\mathcal{M}$ can reach a configuration $c = (p, \vec{n})$ such that p reflects the 0-ary relations in $\mathcal{S}$ and, for every $\gamma \in \Gamma_1$, n_γ is the number of elements of color γ in $\mathcal{S}$.

The automaton $\mathcal{M}$ works in two phases. First, $\mathcal{M}$ guesses the size n of the domain of the initial database. To this end, it increments the counter z_γ to n, where γ is the color assigned to all elements by the initialization formula for domains of size n, and it assumes the state corresponding to the initial 0-ary relations for a database of size n. Here the first of the above observations is used. Then $\mathcal{M}$ simulates an actual computation of $\mathcal{P}$ from the initial database of size n as follows. Every modification $\mathrm{INS}_S(a)$ (or $\mathrm{DEL}_S(a)$, respectively) in $\mathcal{P}$ is simulated by a sequence of three transitions in $\mathcal{M}$:

- First, the counter z_γ, where γ is the color of a before the modification, is decremented.
- Second, the counters for all colors are adapted according to the update formulas of $\mathcal{P}$.
- Third, the counter $z_{\gamma'}$, where γ' is the color of a after the modification, is incremented.

If a modification changes an input bit, the first and third step are omitted. The state of $\mathcal{M}$ is changed to reflect the changes of the 0-ary relations of $\mathcal{P}$. For this second phase the second of the above observations is used.

To detect when the simulation of $\mathcal{P}$ reaches a state with non-empty query relation $R_{\mathcal{Q}}$, states $p \in \Gamma_0$ may have a transition to the accepting state f. ◀

4.2 Emptiness of consistent dynamic programs

Some readers of the proof of Theorem 6 might have got the impression that we were cheating a bit, since the dynamic programs it constructs do not behave as one would expect: in all three cases each modification sequence α that yields a non-empty query relation $R_{\mathcal{Q}}$ can be changed, e.g., by switching two operations, into a sequence that does not correspond to a run of the CA and therefore does *not* yield a non-empty query relation. That is, the program $\mathcal{P}$ is *inconsistent* because it might yield different results when the same database is reached through two different modification sequences.

---

[9] We note that (the complement of) this emptiness problem is often called *control-state reachability* problem.

[10] The simulation of states by counters can be done as in [13, Lemma 2.1]

It seems, that this inconsistency made the proof of Theorem 6 much easier. Therefore, the question arises, whether the emptiness problem becomes easier if it can be taken for granted that the given dynamic program is actually consistent. We study this question in this subsection and will investigate the related decision problem whether a given dynamic program is consistent in the next section.

As Table 1 shows, the emptiness problem for consistent dynamic programs is indeed easier in the sense that it is decidable for a considerably larger class of dynamic programs. While emptiness for general DynFO programs is already undecidable for the tiny fragment with unary input relations and 0-ary auxiliary relations, it is decidable for consistent DynFO programs with unary input and unary auxiliary relations. Likewise, for DynProp there is a significant gap: for consistent programs it is decidable for arbitrary input arities (with unary auxiliary relations) or arbitrary auxiliary arities (with unary input relations), but for general programs emptiness becomes undecidable as soon as binary relations are available (in the input *or* in the auxiliary database).

We call a dynamic program $\mathcal{P}$ *consistent*, if it maintains a query with respect to an empty initial database, that is, if, for all modification sequences α to an empty initial database $\mathcal{D}_\emptyset$, the query relation in $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ depends only on the database $\alpha(\mathcal{D}_\emptyset)$. In the remainder of this subsection we show the undecidability and decidability results stated in Table 1.

▶ **Theorem 8.** *The emptiness problem is undecidable for*
(a) *consistent* DynFO(2-*in*, 0-*aux*)*-programs, and*
(b) *consistent* DynFO(1-*in*, 2-*aux*)*-programs.*

Proof idea. Statement (a) is a corollary of the proof of Theorem 5, as the reduction in that proof always yields a consistent program.

For (b), we present another reduction from the emptiness problem for semi-deterministic 2CAs (see also the proof of Theorem 6). From a semi-deterministic 2CA $\mathcal{M}$ we will construct a consistent Boolean dynamic program $\mathcal{P}$ with a single unary input relation U. The query maintained by $\mathcal{P}$ is "$\mathcal{M}$ halts after at most $|U|$ steps". Clearly, such a program has a non-empty query result for some database and some modification sequence if and only if $\mathcal{M}$ has an accepting run.

The general idea is that $\mathcal{P}$ simulates one step of the run of $\mathcal{M}$ whenever a new element is inserted to U. A slight complication arises from deletions from U, since it is not clear how one could simulate $\mathcal{M}$ one step "backwards". Therefore, when an element is deleted from U, $\mathcal{P}$ freezes the simulation and stores the size m of $|U|$ before the deletion. It continues the simulation as soon as the current size ℓ of U grows larger than m, for the first time. ◀

Contrary to the case of not necessarily consistent programs, the emptiness problem is decidable for consistent DynFO(1-in, 1-aux)-programs.

▶ **Theorem 9.** Emptiness *is decidable for consistent* DynFO(1-*in*, 1-*aux*)*-programs.*

Proof idea. The proof uses the fact that the truth of first-order formulas with quantifier depth k in a state of a DynFO(1-in, 1-aux)-program only depends on the number of elements of every color up to k. The states of a consistent DynFO(1-in, 1-aux)-program can therefore be abstracted by a bounded amount of information, namely the number of elements of every color up to $k+1$. This can be used to construct, from a consistent DynFO(1-in, 1-aux)-program $\mathcal{P}$, a nondeterministic finite automaton $\mathcal{A}$ that reads encoded modification sequences for $\mathcal{P}$ in normal form and represents the abstracted state of $\mathcal{P}$ in its own state. In this way the emptiness problem for consistent DynFO(1-in, 1-aux)-programs reduces to the emptiness problem for nondeterministic finite automata. ◀

The picture of decidability of emptiness for consistent programs for all classes of the form DynFO(ℓ-in, m-aux) is pretty clear and simple: it is decidable if and only if $\ell = 1$ *and* $m \leq 1$. Now we turn our focus to the corresponding classes of consistent DynProp-programs. Here we do not have a full picture. We show in the following that it is decidable if $\ell = 1$ *or* $m \leq 1$.

▶ **Theorem 10.** *The emptiness problem is decidable for*
(a) *consistent* DynProp(1-*in*)-*programs.*
(b) *consistent* DynProp(1-*aux*)-*programs.*

Proof idea (of Theorem 10 (a)). The statement follows almost immediately from the fact that every consistent DynProp(1-in)-program with 0-ary query relations maintains a regular language [10, Theorem 3.2]. ◀

To highlight the role of the Sunflower Lemma for the proof of Theorem 10 (b), we give a full exposition of this proof in the following. At first, we sketch the basic proof idea for consistent DynProp(1-aux)-programs over graphs, i.e., the input schema contains a single binary relation symbol E. For simplicity we also assume a 0-ary query relation. The general statement requires more machinery and is proved below.

Our goal is to show that if such a program $\mathcal{P}$ accepts some graph then it also accepts one with "few" edges, where "few" only depends on the schema of the program. To this end we show that if a graph G accepted by $\mathcal{P}$ contains many edges then one can find a large "well-behaved" edge set in G from which edges can be removed without changing the result of $\mathcal{P}$. Emptiness can then be tested in a brute-force manner by trying out insertion sequences for all graphs with few edges (over a canonical domain $\{1, \ldots, n\}$).

More concretely, we consider an edge set "well-behaved", if it consists only of self-loops, it is a set of disjoint non-self-loop-edges, or is is a *star*, that is, the edges share the same source node or the same target node. From the Sunflower Lemma [9] it follows that for every $p \in \mathbb{N}$ there is an $N_p \in \mathbb{N}$ such that every (directed) graph with N_p edges contains p self-loops, or p disjoint edges, or a star with p edges.

Let us now assume, towards a contradiction, that the minimal graph accepted by $\mathcal{P}$ has N edges with $N > N_{M^2+1}$, where M is the number of binary (atomic) types over the schema $\tau = \tau_{\text{in}} \cup \tau_{\text{aux}}$ of $\mathcal{P}$. Then G either contains $M^2 + 1$ self-loops, or $M^2 + 1$ disjoint edges, or a $(M^2 + 1)$-star.

Let us assume first that G has a set $D \subseteq E$ of $M^2 + 1$ disjoint edges. We consider the state $\mathcal{S}$ reached by $\mathcal{P}$ after inserting all edges from $E \setminus D$ into the initially empty graph. Since D contains $M^2 + 1$ edges, there is a subset $D' \subseteq D$ of size $M + 1$ such that all edges in D' have the same atomic type in state $\mathcal{S}$. Let $\mathcal{S}_0$ be the state reached by $\mathcal{P}$ after inserting all edges in $D \setminus D'$ in $\mathcal{S}$. All edges in D' still have the same type in $\mathcal{S}_0$ since $\mathcal{P}$ is a quantifier-free program (though this type can differ from the type in $\mathcal{S}$). Let $e_1, \ldots, e_{M+1}$ be the edges in D' and denote by $\mathcal{S}_i$ the state reached by $\mathcal{P}$ after inserting $e_1, \ldots, e_i$ in $\mathcal{S}_0$. For each i, all edges $e_{i+1}, \ldots, e_{M+1}$ have the same type γ_i in state $\mathcal{S}_i$, again. As the number of binary atomic types is M, there are $i < j$ such that $\gamma_i = \gamma_j$, thus e_{M+1} has the same type in $\mathcal{S}_i$ and $\mathcal{S}_j$. Therefore, inserting the edges $e_{j+1}, \ldots, e_{M+1}$ in $\mathcal{S}_i$ yields a state with the same query bit as inserting those edges in $\mathcal{S}_j$. As the query bit in the latter case is accepting, it is also accepting in the former case, yet in that case the underlying graph has fewer edges than G, the desired contradiction. The case where G contains $M^2 + 1$ self-loops is completely analogous.

Now assume that G contains a star with $M^2 + 1$ edges. The argument is very similar to the argument for disjoint edges. First insert all edges not involved in the star into an initially empty graph. Then there is a set D of many star edges of the same type, and they still have

the same type after inserting the other edges of the star. A graph with fewer edges that is accepted by $\mathcal{P}$ can then be obtained as above.

The idea generalizes to input schemata with larger arity by applying the Sunflower Lemma in order to obtain a "well-behaved" sub-relation within an input relation that contains many tuples. In order to prove this generalization we first recall the Sunflower Lemma, and observe that it has an analogon for tuples.

The Sunflower Lemma was introduced in [9], here we follow the presentation in [14]. A *sunflower* with p *petals* and a *core* Y is a collection of p sets $S_1, \ldots, S_p$ such that $S_i \cap S_j = Y$ for all $i \neq j$.

▶ **Lemma 11** (Sunflower Lemma, [9]). *Let $p \in \mathbb{N}$ and let $\mathcal{F}$ be a family of sets each of cardinality ℓ. If $\mathcal{F}$ consists of more than $N_{\ell,p} \stackrel{def}{=} \ell!(p-1)^\ell$ sets then $\mathcal{F}$ contains a sunflower with p petals.*

We call a set H of tuples of some arity ℓ a *sunflower (of tuples)* if it has the following three properties.

(i) All tuples in H have the same equality type.
(ii) There is a set $J \subset \{1, \ldots, \ell\}$ such that $t_j = t'_j$ for every $j \in J$ and all tuples $t, t' \in H$.
(iii) For all tuples $t \neq t'$ in H the sets $\{t_i \mid i \notin J\}$ and $\{t'_i \mid i \notin J\}$ are disjoint.

We say that H has $|H|$ petals.

The following Sunflower Lemma for tuples has been stated in various variants in the literature, e.g., in [17, 15].

▶ **Lemma 12** (Sunflower Lemma for tuples). *Let $\ell, p \in \mathbb{N}$ and let R be a set of ℓ-tuples. If R contains more than $\bar{N}_{\ell,p} \stackrel{def}{=} \ell^\ell p^\ell (\ell!)^2$ tuples then it contains a sunflower with p petals.*

Proof. Let R be an ℓ-ary relation that contains $\bar{N}_{\ell,p}$ tuples. As there are less than ℓ^ℓ equality types of ℓ-tuples there is a set $R' \subseteq R$ of size at least $p^\ell(\ell!)^2$, in which all tuples have the same equality type. Application of Lemma 2 in [15] yields[11] a sunflower with p petals. ◀

It is instructive to see how Lemma 12 shows that a graph with sufficiently many edges has many selfloops, disjoint edges or a large star: Selfloops correspond to the equality type of tuples (t_1, t_2) with $t_1 = t_2$, many disjoint edges to the case $J = \emptyset$ and the two possible kinds of stars to $J = \{1\}$ and $J = \{2\}$, respectively.

Proof (of Theorem 10 (b)). Now the proof for binary input schemas easily translates to general input schemas. For the sake of completeness we give a full proof.

Suppose that a consistent DynProp(1-aux)-program $\mathcal{P}$ over schema τ with 0-ary[12] query relation accepts an input database $\mathcal{D}$ that contains at least one relation R with many tuples.

Suppose that R is of arity ℓ and contains $\bar{N}_{\ell,M^2+1}$ diverse tuples where M is the number of ℓ-ary (atomic) types over the schema of $\mathcal{P}$. We show that $\mathcal{P}$ already accepts a database with less tuples than $\mathcal{D}$.

By Lemma 12, R contains a sunflower R' of size M^2+1. Consider the state $\mathcal{S}$ reached by $\mathcal{P}$ after inserting all tuples from $R \setminus R'$ into the initially empty database. Since R' contains M^2+1 tuples, there is a subset $R'' \subseteq R'$ of size $M+1$ such that all tuples in R'' have the same atomic type in state $\mathcal{S}$. Let $\mathcal{S}_0$ be the state reached by $\mathcal{P}$ after inserting all tuples in

[11] In [15], elements from the "outer part" of a petal can also occur in the "core". As in R' all tuples have the same equality type, this can not happen in our setting.
[12] At the end of the proof we discuss how to deal with unary query relations.

$R' \setminus R''$ in $\mathcal{S}$. All tuples in R'' still have the same type in $\mathcal{S}_0$ since $\mathcal{P}$ is a quantifier-free program (though this type can differ from the type in $\mathcal{S}$).

Let $\vec{a}_1, \ldots, \vec{a}_{M+1}$ be the tuples in R'' and denote by $\mathcal{S}_i$ the state reached by $\mathcal{P}$ after inserting $a_1, \ldots, a_i$ in $\mathcal{S}_0$. In state $\mathcal{S}_i$ all tuples $a_{i+1}, \ldots, a_{M+1}$ have the same type, again. As the number of ℓ-ary atomic types is k, there are $i < j$ such that a_{M+1} has the same type in $\mathcal{S}_i$ and $\mathcal{S}_j$. Therefore, inserting the edges $e_{j+1}, \ldots, e_{M+1}$ in $\mathcal{S}_i$ yields a state with the same query bit as inserting this sequence in $\mathcal{S}_j$. As the query bit in the latter case is accepting, it is also accepting in the former case, yet in that case the underlying database has fewer tuples than $\mathcal{D}$, the desired contradiction.

If $\mathcal{P}$ has a unary query relation, then the proof has to be adapted as follows. For an accepted database $\mathcal{D}$, the unary query relation contains some element a. Now M is chosen as the number of $(\ell+1)$-ary atomic types (instead of the number of ℓ-ary atomic types), and R'' is chosen as sub-sunflower where all tuples $(\vec{a}_1, a), \ldots, (\vec{a}_{M+1}, a)$ have the same atomic type. The rest of the proof is analogous. ◀

The final result of this subsection gives a characterization of the class of queries maintainable by consistent DynProp(0-aux)-programs. This characterization is not needed to obtain decidability of the emptiness problem for such queries, since this is included in Theorem 10. However, we consider it interesting in its own right.

▶ **Theorem 13.** *A Boolean query $\mathcal{Q}$ can be maintained by a consistent* DynProp(0-*aux*) *program if and only if it is a modulo query.*

Intuitively[13], a *modulo query* is a Boolean combination of constraints of the form: the number of tuples that have some atomic type γ is q modulo p, for some natural numbers $p \geq 2$ and $q < p$.

4.3 The impact of built-in orders

A closer inspection of the proof that the emptiness problem is undecidable for consistent DynFO(1-in, 2-aux)-programs (Theorem 8) reveals that the construction only requires one binary auxiliary relation: a linear order on the "active" elements. The proof would also work if a global linear order on all elements of the domain would be given. We say that a dynamic program has a *built-in linear order*, if there is one auxiliary relation $R_<$ that is always initialized by a linear order on the domain and never changed. Likewise, for a *built-in successor relation.*

That is, the border of undecidability for consistent DynFO-programs actually lies between consistent DynFO(1-in, 1-aux)-programs and consistent DynFO(1-in, 1-aux)-programs with a built-in linear order. Similarly, the border of undecidability for (not necessarily consistent) DynProp-programs actually lies between DynProp(1-in, 1-aux)-programs and DynProp(1-in, 1-aux)-programs with a built-in linear order.

▶ **Proposition 14.** *The emptiness problem is undecidable for*

(a) *consistent* DynFO(1-*in*, 1-*aux*)*-programs with a built-in linear order or a built-in successor relation,*

(b) DynProp(1-*in*, 1-*aux*)*-programs with a built-in successor relation.*

However, for dynamic programs that only have auxiliary bits, linear orders or successor relations do not affect decidability.

[13] The actual formalization uses sets of elements rather than tuples.

▶ **Proposition 15.** *The emptiness problem is decidable for*

(a) *consistent* DynFO(1-*in*, 0-*aux*)*-programs with a built-in linear order or a built-in successor relation,*

(b) DynProp(1-*in*, 0-*aux*)*-programs with a built-in linear order or a built-in successor relation.*

5 The Consistency Problem

In Section 4.2 we studied Emptiness for classes of consistent dynamic programs. It turned out that with this restriction the emptiness problem is easier than for general dynamic programs. One might thus consider the following approach for testing whether a given general dynamic program is empty: Test whether the program is consistent and if it is, use an algorithm for consistent programs. To understand whether this approach can be helpful, we study the following algorithmic problem, parameterized by a class $\mathcal{C}$ of dynamic programs.

Problem:	Consistency($\mathcal{C}$)
Input:	A dynamic program $\mathcal{P} \in \mathcal{C}$ with FO initialization
Question:	Is $\mathcal{P}$ a consistent program with respect to empty initial databases?

It is not very surprising that Consistency is not easier than Emptiness, since deciding Emptiness boils down to finding *one* modification sequence resulting in a state with particular properties and Consistency is about finding *two* modification sequences resulting in two states with particular properties. This high level comparison can actually be turned into rather easy reductions from Emptiness to Consistency.

On the other hand, Consistency can also be reduced to Emptiness. For this direction the key idea is to simulate two modification sequences simultaneously and to integrate their resulting states into one joint state. This is easy if quantification is available, and requires some work for DynProp-fragments.

▶ **Theorem 16.** *Let* $\ell \geq 1, m \geq 0$.

(a) *For every* $\mathcal{C} \in \{\text{DynFO}(\ell\text{-}in, m\text{-}aux), \text{DynFO}(\ell\text{-}in), \text{DynFO}(m\text{-}aux), \text{DynFO}\}$,

  (i) $\text{Emptiness}(\mathcal{C}) \leq \text{Consistency}(\mathcal{C})$, *and*

  (ii) $\text{Consistency}(\mathcal{C}) \leq \text{Emptiness}(\mathcal{C})$.

(b) *For every*
$\mathcal{C} \in \{\text{DynProp}(\ell\text{-}in, m\text{-}aux), \text{DynProp}(\ell\text{-}in), \text{DynProp}(m\text{-}aux), \text{DynProp}\}$,

  (i) $\text{Emptiness}(\mathcal{C}) \leq \text{Consistency}(\mathcal{C})$, *and*

  (ii) $\text{Consistency}(\mathcal{C}) \leq \text{Emptiness}(\mathcal{C})$.

6 The History Independence problem

As discussed in Section 4.2, it is natural to expect that a dynamic program is consistent, i.e., that the query relation only depends on the current database, but not on the modification sequence by which it has been reached. Many dynamic programs in the literature satisfy a stronger property: not only their query relation but *all* their auxiliary relations depend only on the current database. Formally, we call a dynamic program *history independent* if all auxiliary relations in $\mathcal{P}_\alpha(\mathcal{D})$ only depend on $\alpha(\mathcal{D})$, for all modification sequences α and initial empty databases $\mathcal{D}$. History independent dynamic programs (also called *memoryless* [19] or *deterministic* [4]) are still expressive enough to maintain interesting queries like undirected reachability [11], but also some lower bounds are known for such programs [4, 11, 25].

In this section, we study decidability of the question whether a given dynamic program is history independent.

Problem:	HistoryIndependence($\mathcal{C}$)
Input:	A dynamic program $\mathcal{P} \in \mathcal{C}$ with FO initialization
Question:	Is $\mathcal{P}$ history independent with respect to empty initial databases?

Not surprisingly, HistoryIndependence is undecidable in general. This can be shown basically in the same way as the general undecidability of Emptiness in Theorem 5.

▶ **Theorem 17.** HistoryIndependence *is undecidable for* DynFO(2-*in*, 0-*aux*)*-programs.*

However, the precise borders between decidable and undecidable fragments are different for HistoryIndependence than for Emptiness and Emptiness for consistent programs. More precisely, HistoryIndependence is decidable for DynFO- and DynProp-programs with unary input databases, and for DynProp-programs with unary auxiliary databases.

For showing that HistoryIndependence is decidable for DynFO-programs with unary input databases, we prove that if such a program is not history independent then this is witnessed by some reachable small "bad state". A decision algorithm can then simply test whether such a state exists. Bad states satisfy one of two properties: they either locally contradict history independence or they are "inhomogeneous". We define both notions in the following.

A state $\mathcal{S}$ over domain D is *locally history independent*[14] for a dynamic program $\mathcal{P}$ if the following three conditions hold.

(H1) $\mathcal{P}_{\delta_1\delta_2}(\mathcal{S}) = \mathcal{P}_{\delta_2\delta_1}(\mathcal{S})$ for all insertions δ_1 and δ_2.
(H2) $\mathcal{S} = \mathcal{P}_{\text{INS}_R(\vec{a})\text{DEL}_R(\vec{a})}(\mathcal{S})$ if $\vec{a} \notin R^{\mathcal{S}}$, for all $R \in \tau_{\text{in}}$ and $\vec{a}$ over D.
(H3) $\mathcal{S} = \mathcal{P}_{\text{INS}_R(\vec{a})}(\mathcal{S})$ if $\vec{a} \in R^{\mathcal{S}}$ and $\mathcal{S} = \mathcal{P}_{\text{DEL}_R(\vec{a})}(\mathcal{S})$ if $\vec{a} \notin R^{\mathcal{S}}$, for all $R \in \tau_{\text{in}}$ and $\vec{a}$ over D.

Locally history independence is well-suited to algorithmic analysis. The following lemma shows that for testing history independence it actually suffices to test locally history independence for all states reachable by very restricted modification sequences.

▶ **Lemma 18.** *Let $\mathcal{P}$ be a dynamic program.*

(a) *$\mathcal{P}$ is history independent if and only if every state reachable by $\mathcal{P}$ via insertion sequences is locally history independent.*
(b) *If $\mathcal{P}$ is a* DynFO(1-*in*)*-program, then $\mathcal{P}$ is history independent if and only if every state reachable by $\mathcal{P}$ via insertion sequences in normal form is locally history independent.*

A state $\mathcal{S}$ is *homogeneous* if all tuples $\vec{a}$ and $\vec{b}$ with the same (atomic) τ_{in}-type also have the same (atomic) τ_{aux}-type. The following lemma is an immediate consequence of [4, Lemma 16].

▶ **Lemma 19.** *For every history independent* DynFO(1-*in*)*-program, every reachable state is homogeneous.*

We call a state of a DynFO(1-in)-program that is not homogeneous or not locally history independent a *bad state*. The following lemma is the key ingredient for deciding history independence for DynFO(1-in)-programs. It restricts the size of the smallest bad state and therefore allows for testing history independence in a brute-force manner.

[14] We define this term for arbitrary input arity, since the first part of Lemma 18 holds in general.

▶ **Proposition 20.** *Let $\mathcal{P}$ be a* DynFO(1-*in*, *m*-*aux*)*-program. There is a number $N \in \mathbb{N}$, that can be computed from $\mathcal{P}$, such that if $\mathcal{P}$ is* not *history independent, then there exists an empty database $\mathcal{D}_\emptyset$ of size at most N and an insertion sequence α in normal form such that $\mathcal{P}_\alpha(\mathcal{D}_\emptyset)$ is bad.*

▶ **Theorem 21.** HistoryIndependence *is decidable for* DynFO(1-*in*)*-programs.*

Using the same technique as in the proof of Theorem 10 (b), history independence can be shown to be decidable for DynProp(1-aux)-programs.

▶ **Theorem 22.** HistoryIndependence *is decidable for* DynProp(1-*aux*)*-programs.*

7 Conclusion

In this work we studied the algorithmic properties of static analysis problems for (restrictions of) dynamic programs. Most of the results are summarized in Table 1. In general only very strong restrictions yield decidability.

The only cases left open are about DynProp-programs when both the arity of the input and the arity of the auxiliary relations is at least 2. For such programs the status of history independence and emptiness of consistent remains open. We conjecture that for history independence the decidable fragment of DynProp is larger than exhibited here.

Our results will hopefully contribute to a better understanding of the power of dynamic programs. On the one hand the undecidability proofs show that very restricted dynamic programs can already simulate powerful machine models. It is natural to ask whether this power can be used to maintain other, more common queries. On the other hand the decidability results utilize limitations of the state space and the transition between states for classes of restricted programs. Such limitations can be a good starting point for the development of techniques for proving lower bounds for the respective fragments.

References

1 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in DynFO. In *ICALP*, pages 159–170, 2015.

2 Guozhu Dong, Leonid Libkin, and Limsoon Wong. On impossibility of decremental recomputation of recursive queries in relational calculus and SQL. In *DBPL*, page 7, 1995.

3 Guozhu Dong, Leonid Libkin, and Limsoon Wong. Incremental recomputation in local languages. *Inf. Comput.*, 181(2):88–98, 2003.

4 Guozhu Dong and Jianwen Su. Deterministic FOIES are strictly weaker. *Ann. Math. Artif. Intell.*, 19(1-2):127–146, 1997.

5 Guozhu Dong and Jianwen Su. Arity bounds in first-order incremental evaluation and definition of polynomial time database queries. *J. Comput. Syst. Sci.*, 57(3):289–308, 1998.

6 Guozhu Dong, Jianwen Su, and Rodney Topor. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence*, 14, 1995.

7 Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *ICALP*, pages 103–115, 1998.

8 Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995.

9 Paul Erdős and Richard Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, s1-35(1):85–90, 1960.

10 Wouter Gelade, Marcel Marquardt, and Thomas Schwentick. The dynamic complexity of formal languages. *ACM Trans. Comput. Log.*, 13(3):19, 2012.

11 Erich Grädel and Sebastian Siebertz. Dynamic definability. In *ICDT*, pages 236–248, 2012.

12 William Hesse. *Dynamic Computational Complexity*. PhD thesis, University of Massachusetts Amherst, 2003.

13 John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theor. Comput. Sci.*, 8:135–159, 1979.

14 Stasys Jukna. *Extremal combinatorics*, volume 2. Springer, 2001.

15 Stefan Kratsch and Magnus Wahlström. Preprocessing of min ones problems: A dichotomy. In *ICALP*, pages 653–665, 2010.

16 Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

17 Dániel Marx. Parameterized complexity of constraint satisfaction problems. *Computational Complexity*, 14(2):153–183, 2005.

18 Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

19 Sushant Patnaik and Neil Immerman. Dyn-FO: A parallel, dynamic complexity class. In *PODS*, pages 210–221. ACM Press, 1994.

20 Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. Static analysis for logic-based dynamic programs. *CoRR*, abs/1507.04537, 2015.

21 Boris A. Trahtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *AMS Translations, Series 2*, 23:1–5, 1963.

22 Nils Vortmeier. Komplexitätstheorie verlaufsunabhängiger dynamischer Programme. Master's thesis, TU Dortmund University, 2013. In German.

23 Thomas Zeume. The dynamic descriptive complexity of k-clique. In *MFCS*, pages 547–558, 2014.

24 Thomas Zeume and Thomas Schwentick. Dynamic conjunctive queries. In *ICDT*, pages 38–49, 2014.

25 Thomas Zeume and Thomas Schwentick. On the quantifier-free dynamic complexity of reachability. *Inf. Comput.*, 240:108–129, 2015.