The Dynamic Complexity of Formal Languages

WOUTER GELADE
Hasselt University and Transnational University of Limburg
and
MARCEL MARQUARDT
Technische Universität Dortmund
and
THOMAS SCHWENTICK
Technische Universität Dortmund

The paper investigates the power of the dynamic complexity classes DynFO, DynQF and DynPROP over string languages. The latter two classes contain problems that can be maintained using quantifier-free first-order updates, with and without auxiliary functions, respectively. It is shown that the languages maintainable in DynPROP are exactly the regular languages, even when allowing arbitrary precomputation. This enables lower bounds for DynPROP and separates DynPROP from DynQF and DynFO. Further, it is shown that any context-free language can be maintained in DynFO and a number of specific context-free languages, for example all Dyck-languages, are maintainable in DynQF. Furthermore, the dynamic complexity of regular tree languages is investigated and some results concerning arbitrary structures are obtained: There exist first-order definable properties which are not maintainable in DynPROP. On the other hand, any existential first-order property can be maintained in DynQF when allowing precomputation.

Categories and Subject Descriptors: F.4.3 [**Theory of Computation**]: Mathematical Logic and Formal Languages—*Formal Languages*

General Terms: Languages, Theory

Additional Key Words and Phrases:

1. INTRODUCTION

Traditional complexity theory asks for the necessary effort to decide whether a given input has a certain property, more precisely, whether a given string is in a certain language. *Dynamic complexity*, in contrast, asks for the effort to maintain sufficient knowledge to be able to decide whether the input object has the property *after a series of small changes of the object*. The complexity theoretic investigation of the dynamic complexity of algorithmic problems was initiated by Patnaik and Immerman [**?**]. They defined the class DynFO of dynamic problems where small changes in the input can be mastered by formulas of (first-order) predicate logic (or, equivalently, poly-size circuits of bounded depth, see [**?**]). More precisely, the dynamic program makes use of an auxiliary data structure and after each update (say, insertion or deletion) the auxiliary data structure can be adapted by a first-order formula.

Among others, they showed that the dynamic complexity of the following problems is in DynFO: reachability in undirected graphs, minimum spanning forests, multiplication, regular languages, and the Dyck languages D_n. Subsequent work has yielded more problems in DynFO [**?**] some of which are LOGCFL-complete [**?**] and even PTIME-complete [**?**; **?**] (even though the latter are highly artificial). Other work also considered stronger classes (like Hesse's result that Reachability in

This research was carried out when Wouter Gelade was a Research Assistant of the Fund for Scientific Research - Flanders (Belgium).
An extended abstract of this paper appeared in the proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS'09).

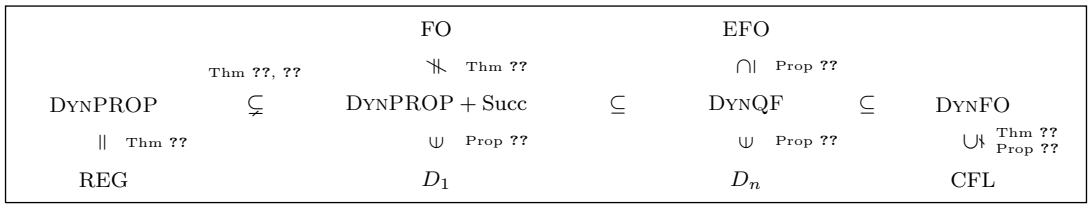

Fig. 1. An overview of the main results in this paper.[1]

arbitrary directed graphs is in DynTC^0 [**?**]), studied notions of completeness for dynamic problems [**?**], and elaborated on the handling of precomputations [**?**].

The choice of first-order logic as update language in [**?**] was presumably triggered by the hope that, in the light of lower bounds for AC^0, it would be possible to prove that certain problems do *not* have DynFO dynamic complexity. As it is easy to show that every DynFO problem is in PTIME, a non-trivial lower bound result would be that the dynamic complexity of some PTIME problem is not in DynFO. However, so far there are no results of this kind.

The inability to prove lower bounds has naturally led to the consideration of subclasses of DynFO. Hesse studied problems with quantifier-free update formulas, yielding DynPROP if the maintained data structure is purely relational and DynQF if functions are allowed as well [**?**; **?**]. As further refinements, the subclasses DynOR and DynProjections were studied. In [**?**] separation results for subclasses of DynPROP were shown and the separation between DynPROP and DynP was stated as an open problem.

The framework of [**?**] allows more general update operations and some of the results we mention depend on the actual choice of operations. Nevertheless, most research has concentrated on insertions and deletions as the only available operations. Furthermore, most work considered underlying structures of the following three kinds.

Graphs. Here, edges can be inserted or deleted. One of the main open questions is whether Reachability (aka Transitive Closure) can be maintained in DynFO for directed, possibly cyclic graphs.

Strings. Here, letters can be inserted or deleted. As mentioned above, [**?**] showed that regular languages and Dyck languages can be maintained in DynFO. Later, Hesse proved that the dynamic complexity of regular languages is actually in DynQF [**?**].

Databases. The dynamic complexity of database properties were studied in the slightly different framework of First-Order Incremental Evaluation Systems (FOIES) [**?**]. Many interesting results were shown, including a separation between deterministic and nondeterministic systems [**?**] and inexpressibility results for auxiliary relations of small arity [**?**; **?**]. However, general lower bounds have not been shown yet.

Continuing the above lines of research, this paper studies the dynamic complexity of formal languages with a particular focus on dynamic classes between DynPROP and DynQF. Our main contributions are as follows (see also Figure **??**):

—We give an exact characterization of the dynamic complexity of regular languages: A language can be maintained in DynPROP if and only if it is regular. This also holds in the presence of arbitrary precomputed (aka built-in) relations. (Section **??**)

—We provide new upper bounds for context-free languages: Every context-free language can be maintained in DynFO, Dyck languages even in DynQF; Dyck languages with one kind of brackets in a slight extension of DynPROP, where built-in successor and predecessor functions can be used. (Section **??**)

[1]In this figure the dynamic complexity classes are allowed to operate with precomputation. Some of the results also hold without precomputation, for example all results concerning formal languages.

—As an immediate consequence, we get a separation between DynPROP and DynQF, thereby also separating DynPROP from DynFO and DynP.

—We investigate a slightly different semantic for dynamic string languages, and we show that also regular tree languages can be maintained in DynPROP, when allowing precomputation and the use of built-in functions. (Section **??**).

—We also study general structures, and show that (bounded-depth) alternating reachability is not maintainable in DynPROP. It follows that not all first-order definable properties are maintainable in DynPROP. On the other hand, we prove that all existential first-order definable properties are maintainable in DynQF when allowing precomputation. (Section **??**)

Related work. We already discussed most of the related work above. A related research area is the study of incremental computation and the complexity of problems in the cell probe model. Here, the focus is not on structural (parallel) complexity of updates but rather on (sequential) update time [**?**; **?**]. In particular, [**?**; **?**] give efficient incremental algorithms and analyse the complexity of formal language classes based on completely different ideas.

Another area related to dynamic formal languages is the incremental maintenance of schema information (aka regular tree languages) [**?**; **?**] and XPath query evaluation [**?**] in XML documents. Again, the interest is mainly in fast algorithms, less in structural dynamic complexity. Nevertheless, techniques from dynamic algorithms for string languages also find applications in these settings.

2. DEFINITIONS

Let $\Sigma = \{\sigma_1, ..., \sigma_k\}$ be a fixed alphabet. We represent words over Σ encoded by *word structures*, that is, logical structures W with universe $\{1 \dots, n\}$, one unary relation R_σ for each symbol $\sigma \in \Sigma$, and the canonical linear order $<$ on $\{1 \dots, n\}$. We only consider structures in which, for each $i \leq n$, there is at most one $\sigma \in \Sigma$ such that $R_\sigma(i)$ holds, but there might be none such σ. We write $W(i) = \sigma$ if $R_\sigma(i)$ holds and $W(i) = \varepsilon$ if no such σ exists. The *size* of W is n.

The word $w = \text{word}(W)$ represented by a word structure W is simply the concatenation $W(1) \circ \cdots \circ W(n)$. Notice that, as certain elements in W might not carry a symbol, the actual length of the string can be less than n. In particular, every word w can be encoded by infinitely many different word structures. Let $[i, j]$ and $]i, j[$ denote the intervals from i to j, respectively, from $i+1$ to $j-1$. For a word structure W and positions $i \leq j$ in $[1, n]$, we write $w[i, j]$ for the (sub-)string $W(i) \circ \cdots \circ W(j)$. In particular, $w[i, i-1]$ denotes the empty substring between positions i and $i-1$.

By E_n we denote the structure with universe $\{1, .., n\}$ representing the empty string ε (thus in E_n all relations R_σ are empty).

2.1 Dynamic Languages and Complexity Classes

In this section, we first define dynamic counterparts of formal languages. Informally, a dynamic language consists of all sequences of insertions and deletions of symbols that transform the empty string into a string of a particular (static) language L. Here, the empty string is encoded by E_n, for some n, and inserting a symbol σ at position i amounts to adding i to R_σ. Then we define dynamic programs which keep track of whether the string resulting from a sequence of updates is in L. Finally, we define complexity classes of dynamic languages. Most of our definitions are inspired by [**?**] but, as we consider strings as opposed to arbitrary structures, we try to keep the formalism as simple as possible.

Dynamic Languages. We will associate with each string language L a dynamic language $\text{Dyn}(L)$. The idea is that words can be changed by a sequence of insertions and deletions of letters and $\text{Dyn}(L)$ is basically the set of update sequences α which turn the empty string into a string in L.

For an alphabet Σ we define the set $\Delta := \{\text{ins}_\sigma \mid \sigma \in \Sigma\} \cup \{\text{reset}\}$ of *abstract updates*. A *concrete update* is a term of the form $\text{ins}_\sigma(i)$ or $\text{reset}(i)$, where i is a positive integer. A concrete update is *applicable* in a word structure of size n if $i \leq n$. By Δ_n we denote the set of applicable

concrete updates for word structures of size n. If there is no danger of confusion we will simply write "update" for concrete or abstract updates.

The semantics of applicable updates is defined as expected: $\delta(W)$ is the structure resulting from W by

—setting $R_\sigma(i)$ to true and $R_{\sigma'}(i)$ to false, for $\sigma' \neq \sigma$, if $\delta = \text{ins}_\sigma(i)$; and
—setting all $R_\sigma(i)$ to false, if $\delta = \text{reset}(i)$.

For a sequence $\alpha = \delta_1 \dots \delta_k \in \Delta_n^+$ of updates we define $\alpha(W)$ as $\delta_k(\dots(\delta_1(W))\dots)$.

DEFINITION 2.1. *Let L be a language over alphabet Σ. The* dynamic language *Dyn(L) is the set of all (non-empty) sequences α of updates, for which there is an $n > 0$ such that $\alpha \in \Delta_n^+$ and word($\alpha(E_n)$) $\in L$. We call L the* underlying language *of Dyn(L).*[2]

Dynamic Programs. Informally, a dynamic program is a transition system which reads sequences of concrete updates and stores the current string and some auxiliary relations in its state. It also maintains the information whether the current string is in the (static) language under consideration.

A *program state* S is a word structure W extended by (auxiliary) relations over the universe of W. The *schema* of S is the set of names and arities of the auxiliary relations of S. We require that each program has a 0-ary relation ACC.

A *dynamic program* P over alphabet Σ and schema $\mathcal{R}$ consists of an *update function* $\phi^R_{\text{op}}(y; x_1, \dots, x_k)$, for every op $\in \Delta$ and $R \in \mathcal{R}$, where $k = \text{arity}(R)$. A dynamic program P operates as follows. Let S be a program state with word structure W. The application of an applicable update $\delta = \text{op}(i)$ on S yields the new state $S' = \delta(S)$ consisting of $W' = \delta(W)$ and new relations $R' = \{\vec{j} \mid S \models \phi^R_{\text{op}}(i, \vec{j})\}$, for each $R \in \mathcal{R}$. For each $n \in \mathbb{N}$ and update sequence $\alpha = \delta_1 \dots \delta_k \in \Delta_n^+$ we define $\alpha(S)$ as $\delta_k(\dots(\delta_1(S))\dots)$. A state S is *accepting* if $S \models$ ACC, that is, if the 0-ary ACC-relation contains the empty tuple.[3]

We say that a dynamic program P *recognizes* the dynamic language Dyn(L) if for all $n \in \mathbb{N}$ and all $\alpha \in \Delta_n^+$ it holds that $\alpha(S_n)$ is accepting if and only if word($\alpha(E_n)$) $\in L$, where S_n denotes the state with word structure E_n and otherwise empty relations.

Dynamic Complexity Classes. DynFO is the class of all dynamic languages that are recognized by dynamic programs whose update functions are definable by first-order formulas. DynPROP is the subclass of DynFO where all these formulas are quantifier free.

2.2 Extended Dynamic Programs

To gain more insight into the subtle mechanics of dynamic computations, we study two orthogonal extensions of dynamic programs: auxiliary functions and precomputations.

Dynamic programs with functions. A *dynamic program with auxiliary functions* P is a dynamic program over a schema $\mathcal{R}$, possibly containing function symbols, which has, for each $\sigma \in \Sigma$ and each function symbol $f \in \mathcal{R}$, an update function $\psi^f_\sigma(i; x_1, ..., x_k)$, where $k = \text{arity}(f)$.

As we are mainly interested in quantifier free update functions for updating auxiliary functions we restrict ourselves to update functions defined by *update terms*, such that

—every x_i is an update term;
—if $f \in \mathcal{R}$ is a function and $\vec{t}$ contains only update terms then $f(\vec{t})$ is an update term; and

[2]There is a danger of confusion as we deal with two kinds of languages: "normal languages" consisting of "normal strings" and dynamic languages consisting of sequences of updates. We use the terms "word" and "string" only for "normal strings" and call the elements of dynamic languages "sequences".
[3]0-ary relations can be viewed as propositional variables: either they contain the empty tuple (corresponding to TRUE) or not.

—if ϕ is a quantifier free formula (possibly using update terms) and t_1 and t_2 are update terms, then $\mathbf{ite}(\phi, t_1, t_2)$ is an update term.

The semantics of update terms is straightforward for the first two rules. A term $\mathbf{ite}(\phi, t_1, t_2)$ takes the value of t_1 if ϕ evaluates to true and the value of t_2 otherwise.

After an update δ, the auxiliary functions in the new state are defined by the update functions in the straightforward way. Unless stated otherwise, the functions in the initial state S_n map every tuple to its first element.

Dynamic programs with precomputations. Sometimes it can be useful for a dynamic algorithm to have a precomputation which prepares some sophisticated data structure. Such precomputations can easily be incorporated into the model of dynamic programs.

In [**?**], the class DynFO$^+$ allowed polynomial time precomputations on the auxiliary relations. The structural properties of dynamic algorithms with precomputation were further studied and refined in [**?**]. In this paper, we do not consider different complexities of precomputations but distinguish only the cases where precomputations are allowed or not.

A *dynamic program P with precomputations* uses an additional set of *initial auxiliary relations* (and possibly *initial auxiliary functions*). For each initial auxiliary relation symbol R and each n, P has a relation R_n^{init} over $\{1, \dots, n\}$. The semantics of dynamic programs with precomputations is adapted as follows: In the initial state S_n each initial auxiliary relation R is interpreted by R_n^{init}. Similarly, for initial auxiliary function symbol f and each n, there is a function f_n^{init} over $\{1, \dots, n\}$.

Initial auxiliary relations and functions are never updated, that is, P does not have update functions for them.

The extension of dynamic programs by functions and precomputations can be combined and gives rise to different complexity classes: For $I \in \{\bot, \text{Rel}, \text{Fun}\}$ and $A \in \{\text{Rel}, \text{Fun}\}$, we denote by DynC$(I, A)$ the class of dynamic languages recognized by dynamic programs

—without precomputations, if $I = \bot$;
—with initial auxiliary relations, if $I = \text{Rel}$;
—with initial auxiliary relations and functions, if $I = \text{Fun}$;
—with (updatable) auxiliary relations only, if $A = \text{Rel}$; and
—with (updatable) auxiliary relations and functions, if $A = \text{Fun}$.

Thus, we have DynFO = DynFO$(\bot, \text{Rel})$ and DynPROP = DynPROP$(\bot, \text{Rel})$. If the base class DynC is DynPROP or DynFO, DynC(I, A) is clearly monotonic with respect to the order $\bot < \text{Rel} < \text{Fun}$. In particular,

$$\text{DynPROP}(\text{Rel}, \text{Rel}) \subseteq \text{DynPROP}(\text{Fun}, \text{Rel}) \subseteq \text{DynPROP}(\text{Fun}, \text{Fun})$$

As we are particularly interested in the class DynPROP$(\bot, \text{Fun})$, we also denote it more consisely by DynQF.

As auxiliary functions can be simulated by auxiliary relations when the update functions are first-order formulas, we also have DynFO(Rel, Rel) = DynFO(Fun, Fun) and DynFO = DynFO$(\bot, \text{Fun})$. Thus, in our setting there are only two classes with base class DynFO: the one with and the one without precomputations.

We also examine the setting where we only allow a specific set of initial auxiliary (numerical) functions, namely built-in successor and predecessor functions. For each universe size n let succ be the function that maps every universe element to its successor (induced by the ordering) and the element n to itself, let pre be the function mapping each element to its predecessor and the element 1 to itself, and let min be the constant (that is, nullary function) mapping to the minimal element 1 in the universe. Then DynPROP$(\text{Succ}, \text{Rel})$ is the class of dynamic languages recognized by dynamic

programs using quantifier free formulas with initial (precomputed) auxiliary relations, the auxiliary functions succ, pre and min and updatable auxiliary relations.

Dynamic Programs with initialisation. In some cases, dynamic programs need some weak kind of precomputation. In these cases, it will be useful to be able to suitably initialize the auxiliary relations, in particular in settings where no precomputation is allowed. The following lemma shows that this is indeed possible, if the initialization functions can be defined in the same logic as the update functions. A *dynamic program with initialization* is a dynamic program with an additional quantifier free formula $\beta_R(\vec{x})$, for each auxiliary relation R. The value of each relation R in the initial state S_n is then determined by β_R.

LEMMA 2.2. *For each dynamic* DYNPROP- *or* DYNFO*-program P with initialization there is an equivalent program P' that does not use initialization.*

PROOF. The simulating program P' uses an additional 0-ary relation I_0 which contains the empty tuple if some update has already occurred. The update formulas of P' are obtained from those of P by replacing each atom of the form $R(\vec{x})$ by $(I_0 \wedge \beta_R(\vec{x})) \vee (\neg I_0 \wedge R(\vec{x}))$. The update formulas for I_0 are constantly true. ∎

Finally, to simplify presentation, we restrict the update sequences under consideration. An update sequence is *useful* if (1) whenever an update $\mathrm{reset}(z)$ occurs, the position z carried a symbol before the update, and (2) whenever an update $\mathrm{ins}_\sigma(z)$ occurs, the position z was empty (that is, did not carry a symbol). Restricting the update sequences to useful ones can be done without loss of generality, as shown in the following lemma. Therefore, when constructing dynamic programs in the remainder of the paper, we always assume that it only has to deal with useful update sequences.

LEMMA 2.3. *For every dynamic program P working only on useful update sequences, there exists an equivalent one P' working on all update sequences.*

PROOF. The program P' can simulate P. Indeed, for the reset operation, P' can test whether z was empty before the update, in which case it returns the original value of the updated relation or function; or, if z carried a symbol, it uses the update functions of P. In the case of an insertion at a position z for which z already carried a symbol, P' can *simulate* what would happen if in P consecutively the updates $\mathrm{reset}(z)$ and $\mathrm{ins}_\sigma(z)$ would occur. Technically, this can be achieved by replacing in all formulas $\phi^R_{\mathrm{ins}_\sigma}$ any occurrence of a relation name R' by $\phi^{R'}_{\mathrm{reset}}$. These modified update formulas then compute exactly the relations and functions P would compute after handling the updates $\mathrm{reset}(z)$ and $\mathrm{ins}_\sigma(z)$. ∎

2.3 Types

We finally introduce the concept of the type of a tuple of elements. Informally, the type of a tuple captures all information a quantifier free formula can express about a tuple. Let $\vec{i} = (i_1, \ldots, i_l)$ be an l-tuple of elements of a state S and let φ be a quantifier free formula using variables from $x_1, \ldots, x_l$. We write $\varphi[\vec{i}]$ for the formula resulting from φ by replacing each x_j with i_j. For instance, for $\vec{i} = (2, 5, 4)$ and the atom $\varphi = R(x_3, x_1)$, we have $\varphi[\vec{i}] = R(4, 2)$.

Let the *type* $\langle S, \vec{i}\rangle$ of an l-tuple $\vec{i} = (i_1, \ldots, i_l)$ in state S be the set of those atomic formulas φ over $x_1, \ldots, x_l$ for which $\varphi[\vec{i}]$ holds in S. We also sometimes call $\langle S, \vec{i}\rangle$ an *l-type*. A tuple $\vec{i} = (i_1, \ldots, i_l)$ is *ordered* if $i_1 < i_2 < \cdots < i_l$. An *ordered type* is the type of an ordered tuple.

3. DYNAMIC COMPLEXITY OF REGULAR LANGUAGES

As already mentioned in the introduction, every regular language can be recognized by a DYNFO program [**?**]; and the full power of DYNFO is actually not needed: Every regular language is recognized by some DYNQF program [**?**].

Our first result is a precise characterization of the dynamic languages $\mathrm{Dyn}(L)$ with an underlying regular language L: They exactly constitute the class DYNPROP. Before stating the result

formally and sketch its proof, we will give a small example to illustrate how regular languages can be maintained in DynPROP.

Example 3.1. *We consider the regular language $(a+b)^*a(a+b)^*$ over the alphabet $\{a,b\}$. The dynamic program maintains one binary relation A containing all pairs (i,j) for which $i<j$ and there exists $k \in]i,j[$ such that $w[k,k]=a$. It further uses a unary relations I containing all j for which there exists a $k<j$ such that $w[k,k]=a$ and, dually, a unary relation F containing all i for which there exists a $k>i$ such that $w[k,k]=a$.*

We will state here the update formulas for the three kinds of operations: ins_a, ins_b, and reset. The formulas for the insertion of the symbol b into the string or the deletion of a string symbol are the same, since the language only distinguishes whether there is an a in the string or not.

After the operation $ins_a(y)$, the relations can be updated as follows

$$\begin{aligned}
\phi^A_{ins_a}(y;x_1,x_2) &\equiv \big[(y \le x_1 \vee y \ge x_2) \wedge A(x_1,x_2)\big] \vee \big[x_1 < y < x_2\big]\\
\phi^I_{ins_a}(y;x) &\equiv \big[y \ge x \wedge I(y)\big] \vee \big[y < x\big]\\
\phi^F_{ins_a}(y;x) &\equiv \big[y \le x \wedge F(y)\big] \vee \big[y > x\big]\\
\phi^{ACC}_{ins_a}(y) &\equiv \mathit{true},
\end{aligned}$$

and after the operations $ins_b(y)$ and $reset(y)$, the relations can be updated as follows

$$\begin{aligned}
\phi^A_{reset/ins_b}(y;x_1,x_2) &\equiv \big[(y \le x_1 \vee y \ge x_2) \wedge A(x_1,x_2)\big] \vee \big[x_1 < y < x_2 \wedge A(x_1,y) \vee A(y,x_2)\big]\\
\phi^I_{reset/ins_b}(y;x) &\equiv \big[y \ge x \wedge I(y)\big] \vee \big[y < x \wedge I(y) \vee A(y,x)\big]\\
\phi^F_{reset/ins_b}(y;x) &\equiv \big[y \le x \wedge F(y)\big] \vee \big[y > x \wedge F(y) \vee A(y,x)\big]\\
\phi^{ACC}_{reset/ins_b}(y) &\equiv I(y) \vee F(y).
\end{aligned}$$

It is crucial here that $A(i,j)$ refers to the substring from $i+1$ up to position $j-1$ (as opposed to i and j). Otherwise it would not be possible to maintain these auxiliary relations. In the update formula $\phi^A_{ins_a}(y;x_1,x_2)$ for example, one can only use the three variables y, x_1 and x_2 to compute the new value of $A(x_1,x_2)$ but needs the knowledge about the string on the intervals $]x_1,y[$ and $]y,x_2[$. ∎

As said before, the dynamic languages $\mathrm{Dyn}(L)$ with an underlying regular language L exactly constitute the class DynPROP.

Theorem 3.2. *Let L be a language. Then, the following are equivalent:*

(a) L is regular.
(b) $Dyn(L) \in$ DynPROP.
(c) $Dyn(L) \in$ DynPROP(Rel, Rel).

The rest of this section is devoted to proving this theorem. First, in Proposition **??** we show that, for every regular language L, $\mathrm{Dyn}(L) \in$ DynPROP (a ⇒ b). Its converse is proven in Proposition **??** (b ⇒ a). As any dynamic language in DynPROP is, by definition, also in DynPROP(Rel, Rel) (b ⇒ c); it then suffices to extend Proposition **??** to Proposition **??**, which states that for every dynamic language $\mathrm{Dyn}(L)$ in DynPROP(Rel, Rel), L is regular (c ⇒ a). Together, these imply Theorem **??**.

Proposition 3.3. *For every regular language L, $Dyn(L) \in$ DynPROP.*

Proof. Let $A=(Q,\delta,s,F)$ be a DFA accepting L. Here, Q is the set of states, $\delta : Q \times \Sigma \to Q$ is the transition function, s is the initial state and F is the set of accepting states. As usual, we denote by $\delta^* : Q \times \Sigma^* \to Q$ the reflexive, transitive closure of δ. Then, $w \in L(A)$ if and only if $\delta^*(s,w) \in F$.

The program P recognizing $\mathrm{Dyn}(L)$ uses the following relations.

—For any pair of states $p, q \in Q$, a relation

$$R_{p,q} = \{(i,j) \mid i < j \wedge \delta^*(p, w[i+1, j-1]) = q\};$$

—For each state q, a relation $I_q = \{j \mid \delta^*(s, w[1, j-1]) = q\}$; and
—For each state p, a relation $F_p = \{i \mid \delta^*(p, w[i+1, n]) \in F\}$,
where n is the size of the word structure.

As already mentioned in example **??**, it is crucial here that $R_{p,q}(i,j)$ refers to the substring from position $i+1$ up to position $j-1$ (as opposed to j), as will become clear in the following.

Thanks to Lemma **??** we can assume that these relations are initialized as follows.

—$R_{p,p} = \{(i,j) \mid i < j\}$ and $R_{p,q} = \emptyset$, for $p \neq q$;
—$I_s = \{1, \ldots, n\}$ and $I_q = \emptyset$, for $q \neq s$;
—$F_p = \{1, \ldots, n\}$ if $p \in F$ and $F_p = \emptyset$, otherwise.

We now show how these relations can be maintained. First, for each $\sigma \in \Sigma$ and $p, q \in Q$, we have the following update formulas for relations $R_{p,q}$:

$$\begin{aligned}
\phi^{R_{p,q}}_{\mathrm{ins}_\sigma}(y; x_1, x_2) \equiv \quad & \big(y \notin]x_1, x_2[\wedge R_{p,q}(x_1, x_2)\big) \\
\vee \ & \big(y \in]x_1, x_2[\wedge \bigvee_{\substack{p', q' \in Q \\ \delta(p', \sigma) = q'}} R_{p,p'}(x_1, y) \wedge R_{q',q}(y, x_2)\big), \\
\phi^{R_{p,q}}_{\mathrm{reset}}(y; x_1, x_2) \equiv \quad & \big(y \notin]x_1, x_2[\wedge R_{p,q}(x_1, x_2)\big) \\
\vee \ & \big(y \in]x_1, x_2[\wedge \bigvee_{p' \in Q} R_{p,p'}(x_1, y) \wedge R_{p',q}(y, x_2)\big).
\end{aligned}$$

The formulas for the other relations are along the same lines, e.g., for each $\sigma \in \Sigma$ and $q \in Q$, and the relation I we have the following update formula:

$$\begin{aligned}
\phi^{I_q}_{\mathrm{ins}_\sigma}(y; x) \equiv \quad & \big(y \geq x \wedge I_q(x)\big) \\
\vee \ & \big(y < x \wedge \bigvee_{\substack{p', q' \in Q \\ \delta(p', \sigma) = q'}} I_{p'}(y) \wedge R_{q',q}(y, x)\big).
\end{aligned}$$

Finally, ACC can be updated by the formulas

$$\phi^{\mathrm{ACC}}_{\mathrm{ins}_\sigma}(y) \equiv \bigvee_{\substack{p', q' \in Q \\ \delta(p', \sigma) = q'}} I_{p'}(y) \wedge F_{q'}(y) \qquad \text{and} \qquad \phi^{\mathrm{ACC}}_{\mathrm{reset}}(y) \equiv \bigvee_{p' \in Q} I_{p'}(y) \wedge F_{p'}(y).$$

■

We next show that the converse of Proposition **??** is also true.

PROPOSITION 3.4. *Let Dyn(L) be a dynamic language in* DYNPROP. *Then L is regular.*

PROOF. The idea of the proof is as follows. We consider a dynamic program P for Dyn(L) and the situation where, starting from the empty word, the positions of a word are set in a left-to-right fashion. Since the acceptance of the word by P does not depend on the order of updates used to produce the word, it suffices to consider this particular sequence of updates. We then show that, for such update sequences, we can construct a finite automaton which *simulates* P and thus recognizes L. It then follows that L is regular.

The three following observations enable us to simulate P by an automaton.

(1) After each update, all tuples of positions that have not been set yet have the same type (cf. Subsection **??**).

(2) There is only a bounded number (depending only on the number and the maximal arity of the auxiliary relations of P) of different types of such tuples.
(3) The new type of the tuples after one update is uniquely determined by the inserted symbol and the previous type.

Together these observations will enable us to define a finite automaton for L.

To state these observations more formally, we introduce some notation. We call a set I of elements of a state S *l-indiscernible* if all ordered l-tuples over I have the same type. Notice that if $l' < l < |I|$ and I is l-indiscernible then I is also l'-indiscernible.

Let P be a DynPROP program recognizing a dynamic language $\mathrm{Dyn}(L)$ and let $k \geq 1$ be the highest arity of any auxiliary relation of P. We make the observations above concrete. Recall that S_n is the state over universe $\{1, \ldots, n\}$ containing both the auxiliary relations and the relations encoding a word, in which all relations are empty.

Observation 1. *Let S be a state that is reached from S_n by insertions and deletions at positions $\leq m$, for some m. Then, the set $\{m+1, \ldots, n\}$ is k-indiscernible.*

Proof. Consider two ordered k-tuples $\vec{j} = (j_1, \ldots, j_k)$ and $\vec{j'} = (j'_1, \ldots, j'_k)$ of elements from $\{m+1, \ldots, n\}$. Let $\vec{i}$ and $\vec{i'}$ be the tuples $(1, \ldots, m, j_1, \ldots, j_k)$ and $(1, \ldots, m, j'_1, \ldots, j'_k)$. We show by induction on the number of update steps that after every sequence of updates starting in state S_n and resulting in state S it holds that

$$\langle S, \vec{i} \rangle = \langle S, \vec{i'} \rangle.$$

As this holds for every pair of ordered k-tuples in $\{m+1, \ldots, n\}$ we can conclude that $\{m+1, \ldots, n\}$ is indeed k-indiscernible. Obviously, in the state S_n the equation holds. Assume now that in some state S the equation holds and consider one update operation on an element $m' \in \{1, \ldots, m\}$ resulting in state S'. Then, $\langle S', \vec{i} \rangle = \langle S', \vec{i'} \rangle$ if, for every atom φ over the set of variables $\{x_1, ..., x_{m+k}\}$, it holds that $S' \models \varphi[\vec{i}]$ if and only if $S' \models \varphi[\vec{i'}]$. However, the truth values of $\varphi[\vec{i}]$ and $\varphi[\vec{i'}]$ after the update are determined by the evaluation in S of a quantifier free formula ψ in which the free variables are replaced by elements of $\vec{i}$ and $\vec{i'}$, respectively. As $\langle S, \vec{i} \rangle = \langle S, \vec{i'} \rangle$, $S \models \psi[m'; \vec{i}]$ if and only if $S \models \psi[m'; \vec{i'}]$ and, hence, $S' \models \varphi[\vec{i}]$ if and only if $S' \models \varphi[\vec{i'}]$. ∎

Observation 2. *Let S be a state and let $l > k$. If a set I of at least l elements from S is k-indiscernible then it is also l-indiscernible. Furthermore, the type of any ordered l-tuple over I is uniquely determined by the type of its first k elements.*

Proof. Suppose I is k-indiscernible. Let $\vec{i} = (i_1, \ldots, i_l)$ and $\vec{i'} = (i'_1, \ldots, i'_l)$ be two ordered l-tuples over I. We show that $\langle S, \vec{i} \rangle = \langle S, \vec{i'} \rangle$, from which it then follows that I is l-indiscernible. To this end, let R be any relation from S and let k' be its arity, and let $j_1, \ldots, j_{k'} \in [1, l]$. Because $k' \leq k$ and I is k-indiscernible $S \models R(i_{j_1}, \ldots, i_{j'_k})$ if and only if $S \models R(i'_{j_1}, \ldots, i'_{j'_k})$. As this holds for every R and every sequence $j_1, \ldots, j_{k'}$ we conclude that indeed $\langle S, \vec{i} \rangle = \langle S, \vec{i'} \rangle$ and thus I is l-indiscernable.

We next show that the type of an ordered l-tuple $\vec{i} = (i_1, \ldots, i_l)$ is determined by the type of its first k elements i_1 to i_k. Indeed, the type of $\vec{i}$ only depends on whether $S \models R(i_{j_1}, \ldots, i_{j'_k})$, for every relation R, with $\mathrm{arity}(R) = k'$, and $j_1, \ldots, j_{k'} \in [1, l]$. However, as $k' \leq k$, the set $\{i_{j_1}, \ldots, i_{j'_k}\}$ contains at most k different elements and hence as I is k-indiscernable, there is a tuple $\vec{i'}$ containing only elements in $\{1, \ldots, k\}$ such that $S \models R(i_{j_1}, \ldots, i_{j'_k})$ if and only if $S \models R(\vec{i'})$. Hence, the type of $\vec{i}$ is determined by that of its first k elements. ∎

Observation 3. *Let S_1, S'_1 be states with universes of size n and n', respectively and $m, m', l \in \mathbb{N}$ such that $\langle S_1, m, \ldots, m+l \rangle = \langle S'_1, m', \ldots, m'+l \rangle$. Let S_2 and S'_2 be the states resulting from S_1 and S'_1 by inserting the same symbol σ at positions m and m', respectively. Then $\langle S_2, m+1, \ldots, m+l \rangle = \langle S'_2, m'+1, \ldots, m'+l \rangle$.*

This observation can be proved along the same lines as Observation 1.

The automaton for L is defined as follows. A type τ of ordered k-tuples is *allowed* if there is a (not necessarily reachable) state S with elements $1, \ldots, k+1$ for which every ordered k-tuple is of type τ. Let Q be the set of allowed types of ordered k-tuples. For each such type τ and each symbol σ let $\delta(\tau, \sigma)$ be determined as follows: Let S be a state[4] with elements $\vec{i} = 1, \ldots, k+1$ in which every ordered k-tuple is of type τ. Let S' be the state reached from S after the update $\mathrm{ins}_\sigma(1)$. Then $\delta(\tau, \sigma)$ is $\langle S', 2, \ldots, k+1\rangle$. This new type is also allowed, which can be seen as follows. Because τ is an allowed type, the set $\{1, \ldots, k+1\}$ was k-indiscernable before the update, and hence k'-indiscernable for any $k' \leq k$. Therefore also the set $\{2, \ldots, k+1\}$ has to be k'-indiscernable, for any $k' \leq k$ after the update operation. Now we can add one more element $k+2$ and define the auxiliary relations of all tuples containing $k+2$ just like any arbitrary other tuple (not containing $k+2$) with the same ordering on the elements. Let F be the set of types for which ACC holds. Then $A = (Q, \delta, \tau_0, F)$, where τ_0 is $\langle E'_k, 1, \ldots, k\rangle$. Notice that as the number of k-types is bounded, A is indeed a finite automaton.

We now argue that $L(A) = L$. To this end, consider any word $w = \sigma_1 \cdots \sigma_n$, and the associated update sequence $\alpha_w = \mathrm{ins}_{\sigma_1}(1) \cdots \mathrm{ins}_{\sigma_n}(n)$. Now, we consider an execution of P on this update sequence in a universe of size $n+k$. Then, $\mathrm{word}(\alpha_w(E'_{n+k})) = w$, and hence $\alpha_w(E'_{n+k}) \models \mathrm{ACC}$ iff $w \in L$. Using the observations above it can now be shown that, for any $i \in [0, n]$, it holds in state $\alpha_{w[1,i]}(E'_{n+k})$ that (1) the set $\{i+1, \ldots, n+k\}$ is l-indescernable, for any l; and (2) $\delta(w[1, i], \tau_0)$ is exactly the k-type of the set $\{i+1, \ldots, i+k\}$, determining the type of the entire set $\{i+1, \ldots, n+k\}$. As $\tau \in F$ iff ACC holds in τ, it follows that $w \in L(A)$ iff $\alpha_w(E'_{n+k}) \models \mathrm{ACC}$. ∎

REMARK 3.5. *Proposition* **??** *can be considered a lower bound result as, of course, for every non-regular language L, Dyn(L) $\notin$* DynPROP*.*

The proof of Proposition **??** intuitively relies on the fact that, if a string is constructed by successive insertions in a left-to-right fashion, all remaining string positions cannot be distinguished before they are updated. Using a Ramsey argument, this idea can be generalized to the setting with precomputations, thus showing that (relational) precomputations do not increase the expressive power of DynPROP-programs.

PROPOSITION 3.6. *Let Dyn(L) be a dynamic language in* DynPROP(*Rel, Rel*)*. Then L is regular.*

PROOF. The only thing left to prove is that for any language L such that Dyn(L) is recognized by a DynPROP(Rel, Rel) program, it holds that L is regular. Thus, we extend the technique of the proof of Proposition **??** to also handle DynPROP programs with precomputations. The proof is a generalization of that proof by a Ramsey argument.

To this end, let P be a DynPROP(Rel, Rel) program recognizing a dynamic language Dyn(L) and let $k \geq 1$ be the highest arity of any auxiliary or initial auxiliary relation of P. Again, our goal is to construct a finite automaton for L thus showing that L is regular.

The key to the proof is the following observation.

Observation 1′. *For each n there is some m such that for every state S over a universe of size m there is a k-indiscernible set I of size n.*

PROOF. of Observation 1′: The proof uses a version of Ramsey's theorem for hypergraphs [**?**]: *Given a number c of colors and a natural number n there exists a number $R_c(n)$ such that if the edges of a complete k-hypergraph (all edges are of size k) with $R_c(n)$ vertices are colored with c colors, then it must contain a complete sub-k-hypergraph with n vertices whose edges are all colored with the same color.*

[4]The states of P should not be confused with the states of A. We reserve the word "state" for the former and refer to the latter as A-states.

Let c be the number of different ordered k-types (which only depends on the number and arity of the initial auxiliary relations). Then m can be chosen as $R_c(n)$. Consider a state S over a universe of size m. Construct a hypergraph G as follows. As the vertex set use the set of universe elements and add for every set of elements of size k a k-hyperedge colored with its k-type. This leads to a complete k-hypergraph for which the vertex set of each complete monocolored sub-k-hypergraph corresponds to a k-indiscernable set. By Ramsey's theorem, G must contain a monocolored sub-k-hypergraph of size at least n and hence S contains a k-indiscernable set I of size n. ∎

To complete the proof of Proposition **??** we only need to consider computations of P which set the elements of some k-indiscernible set in a left-to-right fashion. The automaton A is constructed similarly as in the proof of Proposition **??**. Now for every string w of some length n there is, by Observation $1'$, an m such that every state over m elements has a k-indiscernible set $I = \{i_1, \ldots, i_{n+k}\}$ of size $n+k$. By considering the left-to-right update sequence $\delta_w = \mathrm{ins}_{\sigma_1}(i_1) \cdots \mathrm{ins}_{\sigma_n}(i_n)$ which sets the word $w = \sigma_1 \cdots \sigma_n$ on the elements of I, in a universe of size m, it is easy to show that $w \in L$ if and only if w is accepted by A. ∎

4. DYNAMIC COMPLEXITY OF CONTEXT-FREE LANGUAGES

In the previous section, we have seen that the regular languages are exactly those languages that can be recognized by a DynPROP program. In this section, we will study the dynamic complexity of context-free languages. We first show that any context-free language can be maintained in DynFO. Later, we exhibit languages that can be maintained in DynQF or a weak extension of DynPROP.

THEOREM 4.1. *Let L be a context-free language. Then, Dyn(L) is in* DynFO.

PROOF. Let L be a context-free language defined by grammar $G = (V, S, D)$ over an alphabet Σ. Here, V is the set of non-terminals, $S \in V$ is the initial non-terminal, and D is the set of derivation rules. W.l.o.g. we assume that G is in chomsky normal form, that is, every rule in D is either of the form $U \to XY$, with $X, Y \in V$; $U \to a$, with $a \in \Sigma$; or $U \to \varepsilon$. Further, w.l.o.g., we assume that there is a distinguished non-terminal $E \in V$ such that $E \to \varepsilon$ and for all $U \in V$, $U \to UE$ and $U \to EU$. For $U \in V$, and $w \in (V \cup \Sigma)^*$, we write $U \to^* w$ when w can be derived from U. Then, $L(G) = \{w \mid w \in \Sigma^* \wedge S \to^* w\}$.

Our dynamic program P recognizing L maintains, for all $X, Y \in V$, the following relation:

$$R_{X,Y} = \{(i_1, i_2, j_1, j_2) \mid [j_1, j_2] \subseteq [i_1, i_2] \wedge X \to^* w[i_1, j_1 - 1] Y w[j_2 + 1, i_2]\}$$

Intuitively, $(i_1, i_2, j_1, j_2) \in R_{X,Y}$ implies that, assuming $Y \to^* w[j_1, j_2]$, it follows that $X \to^* w[i_1, i_2]$. Notice also that, by our assumptions above, we have $X \to w[i_1, j_1 - 1] w[j_2 + 1, i_2]$ if and only if $R_{X,E}(i_1, i_2, j_1, j_2)$.

We now state the update formulae. For every $\sigma \in \Sigma$ and $X, Y \in V$, the update formula for $\phi^{R_{X,Y}}_{\mathrm{ins}_\sigma}(z; x_1, x_2, y_1, y_2)$ is

$$[y_1, y_2] \subseteq [x_1, x_2] \wedge \phi_1 \wedge \phi_2 \wedge \phi_3 \,,$$

where ϕ_1, ϕ_2, and ϕ_3 are defined according to the position of z with respect to the other variables.

—When z lies in the interval $[y_1, y_2]$ or outside $[x_1, x_2]$, nothing changes:

$$\phi_1 \equiv (z \notin [x_1, x_2] \vee z \in [y_1, y_2]) \wedge R_{X,Y}(x_1, x_2, y_1, y_2).$$

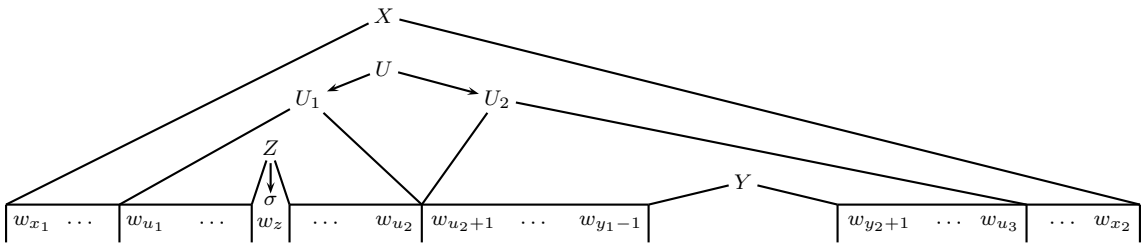

Fig. 2. Update of $R_{X,Y}$ after operation $\mathrm{ins}_\sigma(z)$

—When z lies in $[x_1, y_1[$, the truth value of $R_{X,Y}(x_1, x_2, y_1, y_2)$ can be modified. Figure **??** illustrates this situation.

$$\phi_2 \equiv z \in [x_1, y_1[\wedge \bigvee_{\substack{Z,U,U_1,U_2 \in V \\ Z\to\sigma, U\to U_1U_2 \in D}} \exists u_1, u_2, u_3 :\ u_1 \le u_2 < u_3 \wedge u_1, u_2 \in [x_1, y_1[\wedge u_3 \in [y_2, x_2] \wedge R_{X,U}(x_1, x_2, u_1, u_3) \wedge R_{U_1,Z}(u_1, u_2, z, z) \wedge R_{U_2,Y}(u_2 + 1, u_3, y_1, y_2).$$

—When z lies in the interval $]y_2, x_2[$, the situation is very similar to the previous one; and, hence, ϕ_3 is almost identical to ϕ_2.

For all $X, Y \in V$, the update formula $\phi^{R_{X,Y}}_{\mathrm{reset}}$ is defined very similar as the formula for $\phi^{R_{X,Y}}_{\mathrm{ins}_\sigma}$ above. The only difference is that Z (for which $Z \to \sigma \in D$) is replaced by E (for which $E \to \varepsilon \in D$).

We finally give the update formulae for the acceptance relation ACC:

$$\mathrm{ACC}_{\mathrm{ins}_\sigma}(z) \equiv \bigvee_{\substack{Z \in V \\ Z\to\sigma \in D}} R_{S,Z}(\min, \max, z, z)$$

and

$$\mathrm{ACC}_{\mathrm{reset}}(z) \equiv R_{S,E}(\min, \max, z, z).$$

Notice that we have used many abbreviations in the above formulae. However, these can all easily seen to be definable in first-order logic using the built-in order. In particular, the constants min and max and the successor function are definable and are hence not precomputed functions as in other settings considered in this paper. ∎

However, we cannot hope for an equivalence between DynFO and the context-free languages, as for DynPROP and the regular languages before. This is immediate as, opposed to the class of context-free languages, DynFO is closed under intersection and complement. Furthermore, non-contextfree languages can be maintained in DynQF and DynPROP(Succ, Rel). This is because, in some sense, dynamic programs in the latter classes can count. For any n, let EQUAL_n be the language over the alphabet $\Sigma = \{a_1, \ldots, a_n\}$ containing all strings with an equal number of occurrences of each symbol a_i. Note that EQUAL_n, for $n \ge 3$, is not context-free.

Proposition 4.2. *For any n,*

(1) $Dyn(EQUAL_n) \in$ DynPROP$(Succ, Rel)$; and
(2) $Dyn(EQUAL_n) \in$ DynQF.

PROOF. In both cases, we prove the proposition for the language EQUAL_2. The general case is an easy generalization of this proof.
(1) We maintain the language EQUAL_2 by implementing a unary counter, which can be done in DynPROP(Succ, Rel). This counter will count the difference of the number of occurences of the symbols a_1 and a_2 in the string. For $i \in [1,2]$, let $\sharp a_i$ denote the number of a_i's in the current string. We maintain the following relations:

—Nullary relations (flags) A_1 and A_2 such that A_1 is true if and only if $\sharp a_1 > \sharp a_2$ and A_2 is true if and only if $\sharp a_2 > \sharp a_1$.
—A unary relation C such that $C(i)$ is true if and only if $|\sharp a_1 - \sharp a_2| = i$. Hence, as the universe consists of the elements $\{1, \ldots, n\}$, at each time C is true for one value i if $\sharp a_1 \neq \sharp a_2$ and is false for all i if and only if $\sharp a_1 = \sharp a_2$.

We give the update functions for these relations only for the case of the insertion of a symbol a_1. The deletion and the insertion of a_2 work similarly.

For the update $\mathrm{ins}_{a_1}(x)$, the flags A_1 and A_2 can be updated as follows

$$\phi^{A_1}_{\mathrm{ins}_{a_1}} \equiv \neg A_2 \text{ and } \phi^{A_2}_{\mathrm{ins}_{a_1}} = A_2 \wedge \neg C(\min).$$

For the update of C we distinguish three cases:

$$\begin{aligned}\phi^{C}_{\mathrm{ins}_{a_1}}(x) \;\equiv\; & (\neg(A_1 \vee A_2) \wedge x = \min) \vee \\ & (A_1 \wedge C(\mathrm{pre}(x)) \wedge x \neq \min) \vee \\ & (A_2 \wedge C(\mathrm{succ}(x)))\,.\end{aligned}$$

The acceptance query tests whether both A_1 and A_2 are false after the update. That is,

$$\phi^{\mathrm{ACC}}_{\mathrm{ins}_{a_1}}(x) \equiv \neg\phi^{A_1}_{\mathrm{ins}_{a_1}}(x) \wedge \neg\phi^{A_2}_{\mathrm{ins}_{a_1}}(x)\,.$$

(2) To prove that $\mathrm{Dyn}(\mathrm{EQUAL}_2) \in$ DynQF, we will use the same algorithm as before. But, of course, the previous algorithm makes extensive use of the functions of Succ, which are not available in DynQF. Instead, we incrementally construct the min, succ, and pre functions, using the power of DynQF.

Here, we do not require that the constructed min, succ and pre functions are consistent with the order relation. Instead, min will be the first position where a symbol is inserted, its successor the second such position etc. At each point in time, succ and pre therefore define a successor function on those positions that carry a symbol or carried a symbol earlier, which we call the *active* elements. We will not give the precise update functions which are necessary to construct these auxiliary functions, but simply mention the ideas necessary to construct them.

We additionally maintain a unary relation Act, containing all active elements currently included in the successor function, and a constant (that is, nullary function) MAX denoting the last element of the successor ordering. Recall that succ(MAX) = MAX and pre(min) = min should hold by definition of our successor and predecessor functions.

Then, when an update on an element x occurs there are two possibilities. Either $\mathrm{Act}(x)$ already holds in which case nothing has to be changed, or $\mathrm{Act}(x)$ does not hold and hence x has to be added to the successor structure. This is done by setting $\mathrm{Act}(x)$, making x the maximal element and setting the predecessor and successor functions of x, min, and (the old) MAX corresponding to the new situation.

We finally argue that the program constructed above still works properly when using these on-the-fly constructed functions instead of the precomputed ones in Succ. To this end, notice that there are only two differences. First, the constructed successor functions are not consistent with the built-in order relation. As the original program does not make use of this order relation, this does

not make a difference. Second, at any time the constructed successor functions are only defined on k elements, where k is the number of active elements. However, observe that whenever only k elements are active, the current string cannot contain more than k symbols, and hence $C(i)$ does not hold for $i > k$. It should be noted, however, that $C(k)$ can hold. Therefore, for every update, we first compute the new successor functions and use these newly computed functions in the updates of the other relations. This can also done without any problems, and hence we can conclude that the original program still works correctly. ∎

From Proposition **??** and Theorem **??** one can conclude the following:

Corollary 4.3.

(1) DynPROP $\subsetneq$ DynPROP($Succ, Rel$)
(2) DynPROP $\subsetneq$ DynQF ∎

One can also get better upper bounds for the Dyck-languages, the languages of properly balanced parentheses. For a set of opening brackets $\{(_1, ..., (_n\}$ and the set of its closing brackets $\{)_1, ...,)_n\}$, the language D_n is the language produced by the context free grammar

$$S \to SS \mid (_1S)_1 \mid ... \mid (_nS)_n \mid \varepsilon\,.$$

Proposition 4.4. *For every $n > 0$, $D_n \in$ DynQF.*

Proof. The basic idea is similar to the proof of Theorem **??**. We maintain relations R_1 and R_2 corresponding to $R_{S,E}$ and $R_{S,S}$ in the terminology of Theorem **??**. More precisely, $R_1(i_1, i_2)$ holds if the current substring $w[i_1, i_2]$ is well-bracketed. Likewise, $R_2(i_1, i_2, j_1, j_2)$ holds if the string $w[i_1, i_2]$ without the symbols at positions $j_1, \ldots, j_2$ is well-bracketed. More formally, $R_2(i_1, i_2, j_1, j_2)$ holds if and only if $S \to^* w[i_1, j_1 - 1]\, w[j_2 + 1, i_2]$.

Nevertheless, the update formulas in the proof of Theorem **??** make extensive use of existential quantifiers which are not available in DynQF. In the current proof we therefore replace these existential quantifiers by functions. To this end, we maintain several additional functions described below.

As in the proof of Proposition **??**(2), we make use of on-the-fly constructed functions min, succ, and pre, defined at any time on the elements on which an update already occurred in the update sequence. Then, we associate numbers with elements in this successor function, and let min denote the number 0, its successor 1, and so on. We denote the number represented by an element v as $\langle v\rangle$. Conversely, the element representing a number l is denoted by $\rangle l\langle$.

Now, we define the four auxiliary functions needed to maintain R_1 and R_2. In the following, for two positions $i_1 < i_2$, we write $d(i_1, i_2)$ for the number of closing brackets in $[i_1, i_2]$ minus the number of opening brackets in $[i_1, i_2]$. We write $\mathrm{Cl}(v)$ if position v carries a closing bracket and $\mathrm{Op}(v)$ if it carries an opening bracket.

—$f^{\rightarrow}(u, v) =_{\mathrm{def}} \min\{u' \mid \langle v\rangle \geq 1 \wedge \mathrm{Cl}(u') \wedge u' > u \wedge d(u + 1, u') = \langle v\rangle\}$.
Intuitively, $f^{\rightarrow}(u, v)$ is the position to the right of u where, for the first time, $\langle v\rangle$ many brackets pending at u could be closed.
—Analogously, $f^{\leftarrow}(u, v) =_{\mathrm{def}} \max\{u' \mid \langle v\rangle \geq 1 \wedge \mathrm{Op}(u') \wedge u' < u \wedge d(u', u - 1) = -\langle v\rangle\}$.
—$g^{\rightarrow}(u, v) =_{\mathrm{def}} \rangle \max\{d(u + 1, u') \mid u < u' \leq v\}\langle$.
Thus, $g^{\rightarrow}(u, v)$ gives the maximum surplus of closing brackets in a prefix of $w[u+1, v]$. Intuitively, this is the maximum number of pending open brackets at u that can be "digested" by $w[u+1, v]$. Note that the value of $g^{\rightarrow}(u, v)$ might well be 0.
—$g^{\leftarrow}(u, v) =_{\mathrm{def}} \rangle \max\{-d(u', u - 1) \mid v \leq u' < u\}\langle$.

The attentive reader might have noticed that these functions are not always defined for all combinations of arguments u and v. To this end, for each of them there is an accompanying

relation, telling which function values are valid. For instance, $R_f^{\rightarrow}(u,v)$ holds if and only if $f^{\rightarrow}(u,v)$ is defined.

As some of the update terms in the dynamic program for D_n are slightly involved we present the formulas by means of *update programs* in a pseudocode. These update programs (which should not be confused with the overall dynamic program) get the parameters of the relation or function as input, can assign (position) values to local variables, use conditional branching and return a function value (or TRUE or FALSE for relations). We abstain from a formal definition of update programs but it is straightforward to transform them into update terms by successively replacing each local variable with its definition.

We now give the update formulas for the different relations and functions. In the update programs the following subroutine P_0 will appear three times in update programs for D_n. Its meaning will become clear when it is first used.

Subroutine P_0

1: {Parameters: $x_1, x_2, y_1, y_2, i_0, j_0, z$}
2: $m := g^{\rightarrow}(j_0, y_1 - 1)$
3: $j_1 := f^{\rightarrow}(j_0, m)$
4: $i_1 := f^{\leftarrow}(i_0, m)$
5: $m' := g^{\leftarrow}(y_1, j_1 + 1)$
6: $j_2 := f^{\rightarrow}(y_2, m')$
7: **if** $R_1(i_0 + 1, j_0 - 1)$ AND
$R_2(i_1, j_1, i_0, j_0)$ AND
$R_2(j_1 + 1, j_2, y_1, y_2)$ AND
$R_2(x_1, x_2, i_1, j_2)$ **then**
8: Return TRUE
9: **else**
10: Return FALSE

We first give the update program for $R_2(x_1, x_2, y_1, y_2)$ for insertions of a symbol $(_l$ at a position z. Only the case where z is in the left interval (that is, in $[x_1, y_1 - 1]$) is considered. The other case is symmetric to the insertion of $)_l$ into the left interval which will be handled below.

Intuitively, the string is split into four parts each of which has to be well-bracketed:

—The string between $i_0 = z$ and the corresponding bracket to the right at j_0 (assuming that this is before y_1);
—the maximally bracketed string (from i_1 to j_1) around z inside $[x_1, y_1 - 1]$ without $[i_0, j_0]$;
—the substring starting to the right of j_1 and ending at the corresponding (= matching) position j_2 in $[y_2 + 1, x_2]$; and
—the remaining string before i_1 and after j_2.

An illustration can be found in Figure **??**(a). If the matching bracket for z is not before y_1 the construction is slightly different (Figure **??**(b)):

—The string between z and its matching bracket at j_0 in $[y_2 + 1, x_2]$ has to be well-bracketed; and
—the remaining string consisting of $w[x_1, z - 1]$ and $w[j_0 + 1, x_2]$ has to be well-bracketed.

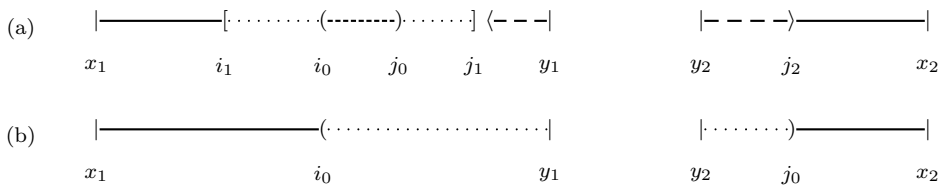

Fig. 3. Illustration of the update programs for (a) insertion of (if the matching bracket is in the left string, (b) if it is in the right string.

Update $R_2(x_1, x_2, y_1, y_2)$: insert $(_l$ at z

1: **if** $z \in [x_1, y_1 - 1]$ **then**
2: $\quad i_0 := z$
3: $\quad j_0 := f^{\rightarrow}(z, 1)$ {find the matching closing bracket}
4: $\quad$ **if** $j_0 < y_1$ **then**
5: $\quad\quad$ **if** $R_{)_l}(j_0)$ **then**
6: $\quad\quad\quad P_0$
7: $\quad\quad$ **else**
8: $\quad\quad\quad$ Return FALSE
9: $\quad$ **else**
10: $\quad\quad m := g^{\leftarrow}(y_1, z)$
11: $\quad\quad j_0 := f^{\rightarrow}(y_2, m + 1)$
12: $\quad\quad$ **if** $R_{)_l}(j_0)$ AND $R_2(z + 1, j_0 - 1, y_1, y_2)$ AND $R_2(x_1, x_2, z, j_0)$ **then**
13: $\quad\quad\quad$ Return TRUE
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ Return FALSE
16: **else**
17: $\quad$ {Symmetric case $z \in [y_2 + 1, x_2]$}

Note that the internal variable m is used for a position that is interpreted as a number (encoded as explained before). Thus, $m + 1$ is an abbreviation for succ(m). Likewise, 0 is an abbreviation for min.

It could be the case that in line 3 no matching bracket is found. In this case the update program fails and returns FALSE. In the actual function terms this can be handled by using the relation $R_f^{\rightarrow}$. We will stick to this convention in the following as well: Whenever a function value is not defined the value of the update program becomes FALSE (corresponding to undefined values for the function update programs below).

Next, we describe the update program for the insertion of $)_l$. This case is very similar to the insertion of $(_l$: The only difference is that j_0 is now the position z and i_0 is the matching position to the left. Furthermore, there is no case distinction as j_0 is always in the left string.

Update $R_2(x_1, x_2, y_1, y_2)$: insert $)_l$ at z

1: **if** $z \in [x_1, y_1 - 1]$ **then**
2: $\quad i_0 := f^{\leftarrow}(z, 1)$
3: $\quad j_0 := z$
4: $\quad P_0$
5: **else**
6: $\quad$ {Symmetric case $z \in [y_2 + 1, x_2]$}

Finally, the following update program handles reset operations. This can be handled just as an insertion with the difference that there is no string between i_0 and j_0.

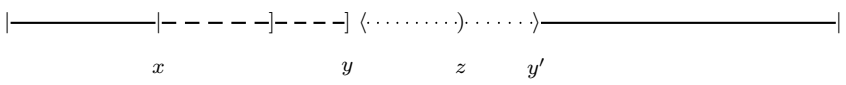

Fig. 4. Illustration of the update program for $f^{\rightarrow}$ under insertion of a closing bracket.

Update $R_2(x_1, x_2, y_1, y_2)$: reset z
1: **if** $z \in [x_1, y_1 - 1]$ **then**
2: $i_0 := z$
3: $j_0 := z$ {The empty string $w[z+1, z-1]$ is well-bracketed...}
4: P_0
5: **else**
6: {Symmetric case $z \in [y_2 + 1, x_2]$}

The update programs for R_1 are similar but easier. We now describe the update programs for the functions $f^{\leftarrow}$, $f^{\rightarrow}$, $g^{\leftarrow}$, and $g^{\rightarrow}$. We only give the update programs for $f^{\rightarrow}$ and $g^{\rightarrow}$ as $f^{\leftarrow}$ and $g^{\leftarrow}$ are again symmetric. We do not explicitly state the update programs for $R_f^{\rightarrow}$ and $R_g^{\rightarrow}$ as they are completely analogous to the programs for the functions.

For $f^{\rightarrow}(x, m)$ we only need to consider the case where m has a corresponding number and is different from min. The insertion of $(_l$ at position z only affects $f^{\rightarrow}(x, m)$ if $x < z < f^{\rightarrow}(x, m)$. In that case, the insertion of z increases $d(x, w)$ by one for all $w > z$ and therefore the previous value of $f^{\rightarrow}(x, m + 1)$ is the new value for $f^{\rightarrow}(x, m)$.

Update $f^{\rightarrow}(x, m)$: insert $(_l$ at z
1: **if** $z \leq x$ **then**
2: return $f^{\rightarrow}(x, m)$
3: **else**
4: $y := f^{\rightarrow}(x, m)$
5: **if** $y < z$ **then**
6: $y := f^{\rightarrow}(x, m)$
7: **else**
8: $y := f^{\rightarrow}(x, m + 1)$

Notice that in this program we are using the assumption that z was empty before the insertion (see Lemma **??**). The update of $f^{\rightarrow}(x, m)$ after the insertion of a closing bracket is slightly more involved. If $x < z < f^{\rightarrow}(x, m - 1)$ then the new value is just $f^{\rightarrow}(x, m - 1)$. Otherwise, we have to identify the maximal pair of matching brackets around z where the left bracket is to the right of $f^{\rightarrow}(x, m - 1)$ $(= y)$. Due to the additional closing bracket at z the right bracket of this pair (y') is then the new value for $f^{\rightarrow}(x, m)$. In case $m = 1$ we simply replace the role of $f^{\rightarrow}(x, m - 1)$ by x. The main case is illustrated by Figure **??**.

Update $f^{\rightarrow}(x,m)$: insert $)_l$ at z

1: **if** $z \leq x$ **then**
2: Return $f^{\rightarrow}(x,m)$
3: **else**
4: **if** $m = 1$ **then**
5: $y := x$
6: **else**
7: $y := f^{\rightarrow}(x, m-1)$
8: **if** $y > z$ **then**
9: Return y
10: **else**
11: $m' := g^{\leftarrow}(z, y+1)$
12: **if** $m' = 0$ **then**
13: Return z
14: **else**
15: Return $f^{\rightarrow}(z, m')$

The update program for a reset operation is similar to the insertion of $)_l$ in case z carries an opening bracket and simpler if z carries a closing bracket.

Update $f^{\rightarrow}(x,m)$: reset z

1: **if** z carries a closing bracket **then**
2: $y := f^{\rightarrow}(x,m)$
3: **if** $y < z$ **then**
4: Return y
5: **else**
6: Return $f^{\rightarrow}(x, m+1)$
7: **else**
8: **if** $z \leq x$ **then**
9: return $f^{\rightarrow}(x,m)$
10: **else**
11: **if** $m = 1$ **then**
12: $y := x$
13: **else**
14: $y := f^{\rightarrow}(x, m-1)$
15: **if** $y > z$ **then**
16: Return y
17: **else**
18: $m' := g^{\leftarrow}(z, y+1)$
19: Return $f^{\rightarrow}(z, m'+1)$

Next, we give the update programs for $g^{\rightarrow}(x,y)$. The first one handles the insertion of an opening bracket and also the reset for closing brackets.

Update $g^{\rightarrow}(x,y)$: insert $(_l$ at z

1: **if** $z \leq x$ OR $z > y$ **then**
2: Return $g^{\rightarrow}(x,y)$
3: $m := g^{\rightarrow}(x,y)$
4: $v := f^{\rightarrow}(x,m)$
5: **if** $v < z$ **then**
6: Return m
7: **else**
8: Return $m-1$

The next one handles insertion of closing brackets.

Update $g^{\rightarrow}(x,y)$: insert $)_l$ at z
1: **if** $z \leq x$ OR $z > y$ **then**
2: Return $g^{\rightarrow}(x,y)$
3: $m := g^{\rightarrow}(x,y)$
4: $v := f^{\rightarrow}(x,m)$
5: **if** $v > z$ **then**
6: Return $m+1$
7: $m' := g^{\leftarrow}(z,v)$
8: **if** $m' = 0$ **then**
9: Return $m+1$
10: **if** $f^{\rightarrow}(z,m') \leq y$ **then**
11: Return $m+1$
12: **else**
13: Return m

The last update program takes care of reset of opening brackets.

Update $g^{\rightarrow}(x,y)$: reset $(_l$ at z
1: **if** $z \leq x$ OR $z > y$ **then**
2: Return $g^{\rightarrow}(x,y)$
3: $m := g^{\rightarrow}(x,y)$
4: $v := f^{\rightarrow}(x,m)$
5: **if** $v > z$ **then**
6: Return $m+1$
7: $m' := g^{\leftarrow}(z,v)$
8: **if** $f^{\rightarrow}(z,m'+1) \leq y$ **then**
9: Return $m+1$
10: **else**
11: Return m

Finally, we give the update formulas for the acceptance relation ACC. To this end, we maintain two additional constants (0-ary functions) first and last. Here, first will denote the first element (first according to the given order, not the constructed successor functions) which has been updated, and, similarly, last denotes the last such element. Hence, at any time $w[1, \text{first}-1] = w[\text{last}+1, n] = \varepsilon$. These functions can easily be maintained. We give the update formulas for the acceptance relation again in our usual formalism:

$$\phi^{\text{ACC}}_{\text{ins}_\sigma}(z) \equiv \phi^{R_1}_{\text{ins}_\sigma}(z; \phi^{\text{first}}_{\text{ins}_\sigma}(z), \phi^{\text{last}}_{\text{ins}_\sigma}(z))$$

and

$$\phi^{\text{ACC}}_{\text{reset}}(z) \equiv \phi^{R_1}_{\text{reset}}(z; \phi^{\text{first}}_{\text{reset}}(z), \phi^{\text{last}}_{\text{reset}}(z)).$$

That is, the string is valid if and only if $R_1(\text{first}, \text{last})$ holds after the update has occurred. This completes the description of the update programs. The correctness proof is tedious but straightforward. ∎

We expect the result to hold for a broader class of context-free languages which has yet to be pinned down exactly. It is even conceivable that all deterministic or unambiguous context-free languages are in DynQF.

We next strengthen the above result for the Dyck language with only one kind of bracket, that is, D_1. Here, auxiliary functions are not needed if built-in successor and predecessor functions are given.

Proposition 4.5. $D_1 \in \text{DynPROP}(\mathit{Succ}, \mathit{Rel})$

Proof. In [**?**] it was shown that D_1 is maintainable in DynFO using the so-called "level trick". To each position i of the string, a number $L(i)$ (the level) is assigned such that $L(i)$ is equal to

the number of opening brackets minus the number of closing brackets in the substring $w[1..i]$. The string is in D_1 if and only if there every position has a positive level and the level of the last position in the string equals 0.

In the following program we maintain a data structure, called a ringlist, capable of storing a set of elements. Technically, a *ringlist* is the edge relation of a directed graph that is a circle. For instance, the set $\{a, b, c\}$ can be stored by the edge relation $\{(a, b), (b, c), (c, a)\}$.

The DynProp(Succ, Rel)-program for D_1 will maintain for all pairs (i, j) of positions in the string and for each number $l \in \{-n, \ldots, -2, -1, 0, 1, 2, \ldots, n\}$ a ringlist of all positions $k \in \{i, \ldots, j\}$ of level l. For this purpose, we use the following relations:

—$L_0(i, j, \cdot, \cdot)$ is a 4-ary relations containing the ringlist of all string positions of level 0. That is, $L_0(i, j, a, b)$ holds when (a, b) is an edge in the ringlist for positions i and j and level 0. We also denote this ringlist by $L_0(i, j)$.

—$L_+(i, j, l, \cdot, \cdot)$ and $L_-(i, j, l, \cdot, \cdot)$ are 5-ary relations containing ringlists for the positive and negative level l and $-l$. Similarly, we denote these ringlists by $L_+(i, j, l)$ and $L_-(i, j, l)$.

—$F_0(i, j)$ holds if $L_0(i, j)$ is not empty.

—$F_+(i, j, l)$ and $F_-(i, j, l)$ hold when $L_+(i, j, l)$ and $L_-(i, j, l)$ are not empty.

—$F\max_0(i)$ holds when $F_0(i, n)$ holds, where n is the universe size (remember that we do not have a constant for the maximal element).

—$F\max_-(i, l)$ and $F\max_+(i, l)$ hold when $F_-(i, n, l)$ and $F_+(i, n, l)$ hold.

—$\mathrm{Min}_0(i, j, k)$ (respectively, $\mathrm{Max}_0(i, j, k)$) holds if and only if k is the minimal (respectively, maximal) element of the ringlist $L_0(i, j)$.

—$\mathrm{Min}_+(i, j, l, k)$, $\mathrm{Min}_-(i, j, l, k)$, $\mathrm{Max}_+(i, j, l, k)$ and $\mathrm{Max}_-(i, j, l, k)$ are the corresponding relations for the ringlist of the other levels beside 0.

—Last_0 is a nullary relation which holds when the level of the last position is 0.

—$\mathrm{Last}_-(l)$ and $\mathrm{Last}_+(l)$ store the level of the last position.

Initially, for all i and j, $F_0(i, j)$, $F\max_0(i)$, Last_0, $\mathrm{Min}_0(i, j, i)$ and $\mathrm{Max}_0(i, j, j)$ are true and

$$L_0(i, j, a, b) = (a, b \in \{i, .., j\} \wedge b = \mathrm{succ}(a)) \vee (a = j \wedge b = i).$$

Thanks to Lemma **??**, we can assume these initializations to take place before the computation of the program.

We can maintain these relations because of the following observation: *After an update operation on some position x in the string, the level of all succeeding positions increases or decreases simultaniously by 1.*

Thus, when we for instance insert a (at position x, then a position is of level l after the update if it either comes before x and had level l before the update, or if it is x or comes after x and had level $l - 1$ before the update.

So, to construct the new ringlist for a level l after an update at a position x one has to merge the ringlist for the position between i and $\mathrm{pre}(x)$ of level l and the one for position between x and j of level $l+1$ or $l-1$. In order to do this, only the relations around the update position x, its two borders i and j and the minimal and maximal element (relative to the ordering) of the considered ringlists have to be changed. We will show that it is possible to express these updates using quantifier free formulas.

Let us first consider the update function for $L_0(i, j)$ and the operation $\mathrm{ins}_(x)$. Here, the levels of all positions from x to n have to increase by one. The update formulas for the relations L_- and L_+ are along the same line, and the ones for the update operations $\mathrm{ins}_)(x)$ and $\mathrm{reset}(x)$ are also very similar. For readability, we use case distinctions and state the formulae for each case separately. They can easily be put together in one (quantifier free) formula.

—If x does not lie in the interval $[i,j]$ then nothing happens: $L_0(i,j)$ remains the same.
—If $x = i$ then the whole list has to be increased by one, so

$$\phi^{L_0}_{\mathrm{ins}_(}(x;i,j,a,b) \equiv L_-(i,j,1,a,b).$$

Here, the constant 1 is not included as a nullary function but can be accessed via succ(min).
—Else, if $x \in [\mathrm{succ}(i),j]$, then he lists $L_0(i,\mathrm{pre}(x),\cdot,\cdot)$ and $L_-(x,j,1,\cdot,\cdot)$ have to be merged. Here, the emptiness-relations $F_0(i,\mathrm{pre}(x))$ and $F_-(x,j,1)$ come into play. Indeed, if one of the corresponding ringlists is empty, the other ringlist just has to be copied. If both are empty, then $L_0(i,j,\cdot,\cdot)$ has to be empty after the update. Only if both $F_0(i,\mathrm{pre}(x))$ and $F_-(x,j,1)$ are false, the following formula applies:

$$\begin{aligned}\phi^{L_0}_{\mathrm{ins}_(}(x;i,j,a,b) \;\equiv\; & a<b<x \wedge L_0(i,\mathrm{pre}(x),a,b) \vee \\ & a<x\le b \wedge (\mathrm{Max}_0(i,\mathrm{pre}(x),a) \wedge \mathrm{Min}_-(x,j,1,b)) \vee \\ & x\le a<b \wedge L_-(x,j,1,a,b) \vee \\ & b<a \wedge (\mathrm{Min}_0(i,\mathrm{pre}(x),b) \wedge \mathrm{Max}_-(x,j,1,a))\end{aligned}$$

Similar to the level-relations L_0, L_- and L_+ we only give the update formulae for $F_0(i,j)$ after the update operation $\mathrm{ins}_(x)$. The formulas for the other emptiness-relations F_- and F_+ and for the other kind of update operations are similar.

—If x does not belong to $[i,j]$, then $F_0(i,j)$ stays the same.
—If $x = i$ then

$$\phi^{F_0}_{\mathrm{ins}_(}(x;i,j) \equiv F_-(i,j,1)$$

because the whole ringlist $L_-(i,j,1,\cdot,\cdot)$ was shifted to L_0. Hence, if L_- was empty before the update operation then, after the update, L_0 is empty.
—In the third case, if $x \in [\mathrm{succ}(i),j]$ then $F_0(i,j)$, is non-empty if either $L_0(i,\mathrm{pre}(x),\cdot,\cdot)$ or $L_-(x,j,1,\cdot,\cdot)$ was non-empty before the update operation. So

$$\phi^{F_0}_{\mathrm{ins}_(}(x;i,j) \equiv F_0(i,\mathrm{pre}(x)) \vee F_-(x,j,1).$$

The relations $F\mathrm{max}_0$, $F\mathrm{max}_-$ and $F\mathrm{max}_+$ can be maintained in a similar way.

Next, we show how to maintain the relation $\mathrm{Min}_0(i,j,k)$ after the update operation $\mathrm{ins}_(x)$. Again, three cases have to be distinguished.

—If $x \notin [i,j]$ then nothing changes.
—If $x = i$, then

$$\phi^{\mathrm{Min}_0}_{\mathrm{ins}_(}(x;i,j,k) \equiv \mathrm{Min}_-(i,j,1,k)$$

—Else, we have to check whether the list $L_0(i,\mathrm{pre}(x),\cdot,\cdot)$ is empty. If it is empty, then the minimum has to be taken from the list $L_-(x,j,1)$. Otherwise, its minimum remains the same. So, we have the following formula for the third case:

$$\begin{aligned}\phi^{\mathrm{Min}_0}_{\mathrm{ins}_(}(x;i,j,k) \;\equiv\; & (\;\; F_0(i,\mathrm{pre}(x)) \wedge \mathrm{Min}_0(i,\mathrm{pre}(x),k)) \vee \\ & (\neg F_0(i,\mathrm{pre}(x)) \wedge \mathrm{Min}_-(x,j,1,k))\end{aligned}$$

Again, the relations Max_0, Min_-, Min_+, Max_+ and Max_- can be updated similarly. The last relations which have to be updated are Last_0, Last_- and Last_+. However, their change does not depend on the position of the actual update operation, but only on the type of the inserted or deleted symbol. In fact they only have to count the difference between the number of opening and closing brackets in the string. The maintenance of these relations is straightforward. For example after the insertion of an opening bracket we have

$$\phi^{\mathrm{Last}_0}_{\mathrm{ins}_(}(x) \equiv \mathrm{Last}_-(1)\,.$$

Now only the acceptance of a string has to be detected. The string is accepted if and only if, after the update, the level of the last position equals 0 and the ringlist of level -1 is empty. We only have to check the level -1, and not all negative levels. Indeed, if there is a position with level less then -1 there also has to be a position which has level -1. So, for instance for the update $\mathrm{ins}_($, the update formula for ACC is

$$\phi^{\mathrm{ACC}}_{\mathrm{ins}_(}(x) \equiv \neg\phi^{F\max_-}_{\mathrm{ins}_(}(x;\min,1) \wedge \phi^{\mathrm{Last}_0}_{\mathrm{ins}_(}(x)\,.$$

■

So, whereas built-in relations do not increase the expressive power of DynPROP, already the three simple functions succ, pre and min allow the maintenance of non-regular languages.

5. VARIATIONS

Alternative Semantics. Following [**?**], we have introduced in Section **??** dynamic languages in which it is both allowed to insert or change labels at positions in the string and to delete elements at positions. In a universe of size n, one can thus create all strings of length smaller or equal than n.

However, one can also consider the setting in which each position in the string must at any time be assigned a symbol. Although this setting is less "dynamic", it has the advantage that a word is always associated with its canonical logical structure. This can be achieved by starting with an initial structure in which each symbol is already assigned a symbol, and subsequently only allowing labels to be changed (and not deleted).

More formally, we assign to every language L a dynamic language Dyn-alt(L) as follows. For a distinguished *initial symbol* $a \in \Sigma$ and $n \in \mathbb{N}$, let E^a_n be the word structure in which $R_a(i)$ is true, for all i, and R_σ is empty, for all $\sigma \neq a$. Further, $\Delta_n = \{\mathrm{ins}_\sigma \mid \sigma \in \Sigma\}$. Then, Dyn-alt($L$) $= \{(n,\delta) \mid \delta \in \Delta^+_n \wedge \mathrm{word}(\delta(E^a_n)) \in L\}$[5].

Proposition **??** shows that the situation is less appealing than in the original semantics. In particular, there are regular languages which cannot be maintained without precomputation; and with precomputation all regular, but also non-regular, languages can be maintained. Here, MIDDLE $= \{wbw' \mid |w| = |w'|\}$ is the language over the alphabet $\Sigma = \{a,b\}$ which contains all strings whose middle element is b, which is clearly not regular.

Proposition 5.1.

(1) *Dyn-alt*($L((aa)^*)$) $\notin$ DynPROP
(2) *For any regular language L, Dyn-alt*(L) $\in$ DynPROP(*Rel*, *Rel*)
(3) *Dyn-alt*(*MIDDLE*) $\in$ DynPROP(*Rel*, *Rel*)

Proof. (1) Let $L = L((aa)^*)$. Let n be any positive even integer, and $\delta = \mathrm{ins}_a(1)$. Then, $\mathrm{word}(\delta(E^a_n)) \in L$, and $\mathrm{word}(\delta(E^a_{n+1})) \notin L$. Hence, $(n,\delta) \in$ Dyn-alt(L) and $(n+1,\delta) \notin$ Dyn-alt(L). We show that for any program $P \in$ DynPROP, $(n,\delta) \in L(P)$ if and only if $(n+1,\delta) \in L(P)$, which implies the proposition.

First, notice that $(n,\delta) \in L(P)$ if and only if $S^a_n \models \phi^{\mathrm{ACC}}_{\mathrm{ins}_a}(1)$, and, correspondingly, $(n+1,\delta) \in L(P)$ if and only if $S^a_{n+1} \models \phi^{\mathrm{ACC}}_{\mathrm{ins}_a}(1)$. However, these two questions can be decided in an identical manner: Take $\phi^{\mathrm{ACC}}_{\mathrm{ins}_a}$, replace any occurrence of R_a by true and any occurence of a relation symbol different from R_a by false, and evaluate the obtained boolean formula. Hence, $S^a_n \models \phi^{\mathrm{ACC}}_{\mathrm{ins}_a}(1)$ if and only if $S^a_{n+1} \models \phi^{\mathrm{ACC}}_{\mathrm{ins}_a}(1)$, which concludes the proof.

[5]Notice that Dyn(L) consists only of update sequences δ, whereas Dyn-alt(L) contains tuples (n,δ). This change is necessary as the membership of a word of a language under the current semantics can depend both on the size of the initial structure n, and the update sequence δ.

(2) As seen in the previous proof, DynPROP program without precomputation are not capable of maintaining all regular languages. The reason for this is that the initial string is a^n, for some n, whereas the initial string was empty in the original semantics. Then, when the computation starts, the DynPROP program did not have the chance to initialize its data structures according to a^n and is immediately lost.

However, when allowing precomputation, we can simply reuse the program P defined in the proof of Proposition **??**. Indeed, the only difference is in the initialization of the relations. Whereas they could be initialized by quantifier free formulas when the initial string was empty, we now have to use the power of precomputations to initialize them. In particular, for a language L accepted by automaton $A = (Q, \delta, s, F)$ they should be initialized as follows:

—$R_{p,q} = \{(i,j) \mid i < j \wedge (p, a^{j-i-1}, q) \in \delta\}$;
—$I_q = \{i \mid (s, a^{i-1}, q) \in \delta\}$; and
—$F_p = \{i \mid (p, a^{n-i}, q_f) \in \delta, \text{ for some } q_f \in F\}$.

From the correctness of the program of Proposition **??** and this precomputation, the correctness of this modified program immediately follows.

(3) The dynamic program P maintaining Dyn-alt(MIDDLE) will make use of the precomputed unary relation M containing the middle element of the structure, if the universe size is odd. Formally, for $n \in \mathbb{N}$, $M_n^{\text{init}} = \{\lceil n/2 \rceil \mid n \text{ is odd}\}$. The program P only needs to maintain the acceptance relation, which can be done as follows:

$$\phi^{\text{ACC}}_{\text{ins}_a}(x) \equiv \text{ACC} \wedge \neg M(x)$$

and

$$\phi^{\text{ACC}}_{\text{ins}_b}(x) \equiv \text{ACC} \vee M(x).$$

■

Notice that, contrary to Theorem **??**, Proposition **??** does not allow to infer lower bounds for DynPROP(Rel, Rel) under the current semantics. However, if we consider the class of languages with neutral elements, this becomes possible again. We say that a language L has a *neutral element* a if for all $w, w' \in \Sigma^*$ it holds that $ww' \in L$ if and only if $waw' \in L$. If a language has at least one neutral element, we assume that the initial symbol for its dynamic algorithm is one of these neutral elements.

A straigthforward generalization of Theorem **??** yields the following proposition which implies, for instance, that Dyn-alt(L) $\notin$ DynPROP(Rel, Rel) for all non-regular languages L which have a neutral element.

Proposition 5.2. *Let L be a language which has a neutral element. Then, the following are equivalent:*

(1) L is regular;
(2) Dyn-alt(L) $\in$ DynPROP; and
(3) Dyn-alt(L) $\in$ DynPROP(Rel, Rel).

Proof. As (2) $\Rightarrow$ (3) follows by definition, it suffices to show (1) $\Rightarrow$ (2) and (3) $\Rightarrow$ (1).

(1) $\Rightarrow$ (2): Let L be a regular language with neutral element and A be the minimal DFA accepting L. Then, the DynPROP program P, accepting Dyn(L), constructed in the proof of Proposition **??** accepts exactly Dyn-alt(L).

It should be clear that the correctness of the update functions of P carries over immediately to the current setting. To see that also the initialization of the different relations is correct, notice that, as A is minimal and a is a neutral element, $(q, a, p) \in \delta$ if and only if $q = p$, for all states p and q of A. Since word(E_n^a) $= a^n$ it follows that the different relations are properly initialized.

(3) ⇒ (1): Let L be a language such that Dyn-alt(L) is accepted by a DynPROP(Rel, Rel) program P. We show that L is regular by constructing a finite automaton accepting L. Again, this can be done almost identically as in the proof of implication (3) ⇒ (1) in Theorem **??**. The key point to notice is that a position which is labeled a in the current semantics can intuitively be seen as an empty, that is, not-labeled, position in the original semantics because a is a neutral element.

Therefore, we proceed in two steps. First, completely ignoring the symbol a, we create the automaton A exactly as in the proof of Theorem **??**. Denote $\Sigma \setminus \{a\}$ by Σ_a. Then, as before, it can be shown that $L(A) = L \cap \Sigma_a^*$, that is, A accepts all strings in L that do not contain an a. As a is a neutral element of L, it holds that $L = \bigcup_{w=\sigma_1 \cdots \sigma_n \in L(A)} L(a^*\sigma_1 a^* \cdots a^*\sigma_n a^*)$. Hence, the desired automaton A', with $L(A') = L$, can be obtained from A by adding the transition (q, a, q) to A, for all states q of A. ∎

Regular Tree Languages. We now investigate the dynamic complexity of the regular tree languages. To this end, we define dynamic tree languages. A tree t over an alphabet Σ is encoded by a logical structure T with as universe the first n elements of the list $(1, 11, 12, 111, 112, 121, 122, \ldots)$, for some $n \in \mathbb{N}$, and consisting of (1) one unary relation R_σ, for each symbol $\sigma \in \Sigma$; (2) a constant root, denoting the element 1; and (3) binary relations L-child and R-child, containing all tuples $(u, u1)$ and $(u, u2)$, respectively.

The updates are terms $\text{ins}_\sigma(u)$ and reset(u), setting and resetting the label of node u in T, exactly as in the string case. So, the logical structure T is a fixed balanced binary tree in which the labels can change. Then, the tree t encoded by T is the largest subtree of T whose root is the element 1 and in which all nodes are labelled with an alphabet symbol. Hence, a node of T is included in t if it, and all its ancestors, carry an alphabet symbol.

Exactly as for the word languages, for a tree language L, we let Dyn(L) be the set of update sequences leading to a tree $t \in L$. A dynamic program works on a dynamic tree language as it does on a dynamic language. We then obtain the following result.

Proposition 5.3. *Let L be a regular tree language. Then, Dyn(L) ∈ DynPROP(Fun, Rel).*

Proof. We first introduce some notation. For a node u of T, let subtree_T^u be the largest subtree of T whose root is u and in which all nodes are labelled with an alphabet symbol. Hence, T encodes the tree $\text{subtree}_T^{\text{root}}$. Further, for a tree t, we denote its set of nodes by nodes(t), and for $u \in \text{nodes}(t)$, $\text{lab}_t(u)$ denotes the *label* of u in t.

The program will make use of the following precomputed relations and functions on T:

—a binary relation Anc, such that Anc(x, y) holds if x is an ancestor of y;
—a binary funtion lca, such that lca$(x, y) = z$ if z is the least common ancestor of x and y;
—a unary function parent such that parent$(u) = v$ if L-child(v, u) or R-child(v, u); and parent$(u) = u$, if $u =$ root;
—unary functions l-child and r-child such that l-child$(u) = v$ if L-child(u, v) and l-child$(u) = u$, otherwise; and r-child$(u) = v$ if R-child(u, v) and r-child$(u) = u$, otherwise.

Let L be a regular (binary) tree language, and $A = (Q, \delta, (q_\sigma^I)_{\sigma\in\Sigma}, F)$ be a bottom-up deterministic tree automaton accepting L, with $\delta : Q \times Q \times \Sigma \to Q$ the (complete) transition function. A *run* of a A on a tree t is a mapping $\rho : \text{nodes}(t) \to Q$ such that (1) for all leaf nodes u of t, $\rho(u) = q_\sigma^I$, where $\text{lab}_t(u) = \sigma$; and (2) for all non-leaf nodes u, with children u_1, u_2, we have $\delta(\rho(u_1), \rho(u_2), \text{lab}(u)) = \rho(u)$. If $\rho(\text{root}) = q$, we say that ρ is a run of A on t *to* q. A tree t is accepted if there is a run of A on t to q_f, for some $q_f \in F$.

We denote by subtree_t^u the subtree of t rooted at u and by $\text{subtree}_t^{u,v}$ the subtree of t with root u which contains all descendants of u but no descendants of v. When t is clear from the context, we

omit it as a subscript. For such a tree subtree^{u,v}, we are also interested in runs which assign a state p to the new leaf node v, not necessarily consistent with the label of v, and are valid runs otherwise. Thus, a function $\rho : \text{nodes}(\text{subtree}_t^{u,v}) \to Q$ is a run of A on subtree$_t^{u,v}[v \to p]$ to q if and only if $\rho(u) = q$, $\rho(v) = p$, and ρ is a valid run of A on subtree$_t^{u,v}$, except that $p = q^I_{\text{lab}(v)}$ does not have to hold.

Before giving the relations we will maintain, we define a few subformulas which will be used several times in the subsequent definitions and formulas.

$$\text{Anc-self}(x,y) \equiv \text{Anc}(x,y) \vee x = y;$$

$$\text{Epsilon}(x) \equiv \bigwedge_{\sigma\in\Sigma} \neg R_\sigma(x); \text{ and}$$

$$\text{Leaf}(x) \equiv (\text{l-child}(x) = x \vee \text{r-child}(x) = x \vee (\text{Epsilon}(\text{l-child}(x)) \wedge \text{Epsilon}(\text{r-child}(x))).$$

Our dynamic program will maintain the following relations:

—$\text{Con} = \{(x,y) \mid \text{Anc-self}(x,y) \wedge \forall z$ with $\text{Anc-self}(x,z) \wedge \text{Anc}(z,y)$, $R_\sigma(z)$ is true, for some $\sigma \in \Sigma\}$;
—$R_q = \{x \mid$ there is a run of A on subtree^x to $q\}$; and
—$R_{q_1,q_2} = \{(x_1,x_2) \mid$ there is a run of A on subtree$^{x_1,x_2}[x_2 \to q_2]$ to $q_1\}$.

That is, the relation Con expresses that elements x and y are *connected* in T, that is, that all nodes on the path from x to y, except possibly y itself, carry an alphabet symbol. The relation R_q contains all nodes x for which there is a run on subtree^x to q; and $(x_1,x_2) \in R_{q_1,q_2}$ intuitively holds if, assuming there is a run on subtree^{x_2} to q_2, then there is a run on subtree^{x_1} to q_1.

First of all, by Lemma **??**, we can assume that these relations are initialized correctly as follows:

—$\text{Con} = \{(x,x)\}$;
—for all $q \in Q$, $R_q = \emptyset$; and
—for all $q_1, q_2 \in Q$, $R_{q_1,q_2} = \emptyset$ if $q_1 \neq q_2$, and $R_{q_1,q_2} = \{(x,x)\}$, otherwise.

We now give the update formulae for the different relations. First, the relation Con can easily be maintained. For all $\sigma \in \Sigma$,

$$\phi^{\text{Con}}_{\text{ins}_\sigma}(y;x_1,x_2) \equiv \big[\neg(\text{Anc-self}(x_1,y) \wedge \text{Anc}(y,x_2)) \wedge \text{Con}(x_1,x_2)\big] \vee \big[\text{Con}(x_1,y) \wedge (\text{Con}(\text{l-child}(y),x_2) \vee \text{Con}(\text{r-child}(y),x_2))\big]$$

$$\phi^{\text{Con}}_{\text{reset}}(y;x_1,x_2) \equiv \neg(\text{Anc-self}(x_1,y) \wedge \text{Anc}(y,x_2)) \wedge \text{Con}(x_1,x_2).$$

Before giving the update formulae for R_q and R_{q_1,q_2} we define a formula which will be used several times. For $p \in Q$ and $\sigma \in \Sigma$, the following formula intuitively says that "if node x is labeled σ, then there is a run on subtree^x to p":

$$\phi^p_\sigma(x) \equiv \big[(\text{Leaf}(x) \wedge q^I_\sigma = p\big] \vee \big[\neg\text{Leaf}(x) \wedge \bigvee_{\substack{p_1,p_2\in Q \\ \delta(p_1,p_2,\sigma)=p}} (R_{p_1}(\text{l-child}(x)) \wedge R_{p_2}(\text{r-child}(x)))\big].$$

We can now give the different update formulae for the insert operation. For each $\sigma \in \Sigma$ and $q \in Q$, the relation R_q can be updated as follows:

$$\phi^{R_q}_{\text{ins}_\sigma}(y;x) \equiv \big[\neg(\text{Anc-self}(x,y) \wedge \text{Con}(x,y)) \wedge R_q(x)\big] \vee \big[\text{Anc-self}(x,y) \wedge \text{Con}(x,y) \wedge \bigvee_{p\in Q} (\phi^p_\sigma(y) \wedge R_{q,p}(x,y)\big].$$

The update formula for R_{q_1,q_2} is similar but more involved. It is defined as follows:

$$\phi^{R_{q_1,q_2}}_{\mathrm{ins}_\sigma}(y;x_1,x_2) \equiv \text{Anc-self}(x_1,x_2) \wedge \phi^{\mathrm{Con}}_{\mathrm{ins}_\sigma}(y;x_1,x_2) \wedge (\phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 \wedge \phi_5),$$

where ϕ_1 to ϕ_5 are formulas defined according to the position of y with respect to x_1 and x_2:

—If y does not occur in subtree^{x_1,x_2} after $\mathrm{ins}_\sigma(y)$, or $y = x_2$, then the truth value of $R_{q_1,q_2}(x_1,x_2)$ is not changed:

$$\phi_1 \equiv (\neg\text{Con}(x_1,y) \vee \text{Anc-self}(x_2,y)) \wedge R_{q_1,q_2}(x_1,x_2).$$

—Let $\mathrm{lca}(x_2,y) = z$. If $y = z$ and x_2 is a left descendant of y, that is, Anc-self(l-child$(y), x_2$), we can determine the state p of z and use this information to decide whether $R_{q_1,q_2}(x_1,x_2)$:

$$\phi_2 \equiv y = z \wedge \text{Anc-self}(\text{l-child}(y),x_2) \wedge \bigvee_{\substack{p,p_1,p_2 \in Q \\ \delta(p_1,p_2,\sigma)=p}} (R_{p_1,q_2}(\text{l-child}(y),x_2) \wedge R_{p_2}(\text{r-child}(y)) \wedge R_{q_1,p}(x_1,y)).$$

—Else if $y = z$, and x_2 is a right descendant of y, then ϕ_3 is almost identical to ϕ_2.

—Else if $y \neq z$ and y is a left descendant of z, then:

$$\phi_4 \equiv y \neq z \wedge \text{Anc-self}(\text{l-child}(z),y) \wedge \bigvee_{p \in Q} \Big(\phi^p_\sigma(y) \wedge \bigvee_{\substack{r,r_1,r_2 \in Q, \sigma' \in \Sigma \\ \delta(r_1,r_2,\sigma')=r}} \big[R_{\sigma'}(r) \wedge R_{r_1,p}(\text{l-child}(z),y) \wedge R_{r_2,q_2}(\text{r-child}(z),x_2) \wedge R_{q_1,r}(x_1,z)\big]\Big).$$

—Else if $y \neq z$ and y is a right descendant of z, then ϕ_5 is almost identical to ϕ_4.

We now give the different formulae for the reset operation. Again, we first define a subformula which will be used several times. The following formula intuitively says that "if node y is reset, and y' is its parent, then there is a run on subtree$^{y'}$ to p":

$$\psi^p(y,y') \equiv \bigvee_{\substack{\sigma \in \Sigma \\ p = q^I_\sigma}} R_\sigma(y') \wedge \big[(\text{l-child}(y') = y \wedge \text{Epsilon}(\text{r-child}(y'))) \vee (\text{r-child}(y') = y \wedge \text{Epsilon}(\text{r-child}(y')))\big].$$

We can now define the different formulae for the reset operation. For all $q \in Q$,

$$\phi^{R_q}_{\mathrm{reset}}(y;x) \equiv \big[\neg(\text{Anc-self}(x,y) \wedge \text{Con}(x,y)) \wedge R_q(x)\big] \vee \big[\text{Anc}(x,y) \wedge \bigvee_{p \in Q} \psi^p(y,\text{parent}(y)) \wedge R_{q,p}(x,\text{parent}(y))\big].$$

Again, the formula $\phi^{R_{q_1,q_2}}_{\mathrm{reset}}$ is similar but more involved:

$$\phi^{R_{q_1,q_2}}_{\mathrm{reset}}(y;x_1,x_2) \equiv \text{Anc-self}(x_1,x_2) \wedge \neg(\text{Anc-self}(x_1,y) \wedge \text{Anc}(y,x_2)) \wedge \text{Con}(x_1,x_2) \wedge (\phi_1 \vee \phi_2 \vee \phi_3).$$

Notice that if any of these conditions is not satisfied then $R_{q_1,q_2}(x_1,x_2)$ cannot hold after reset(y). The formulas ϕ_1, ϕ_2 and ϕ_3 depend on the possible remaining positions of y w.r.t. x_1 and x_2. We only have to distinguish three cases here, as opposed to five before, because we do not have to consider the case $\mathrm{lca}(y,x_2) = y$. Indeed, if $\mathrm{lca}(y,x_2) = y$, resetting y disconnects x_1 from x_2.

—If y does not occur in subtree^{x_1,x_2}, then the truth value of $R_{q_1,q_2}(x_1,x_2)$ is not changed:

$$\phi_1 \equiv (\neg\text{Con}(x_1,y) \vee \text{Anc-self}(x_2,y)) \wedge R_{q_1,q_2}(x_1,x_2).$$

—Let $\mathrm{lca}(x_2, y) = z$ and $\mathrm{parent}(y) = y'$. If $y \neq z$ and y is a left descendant of z, then

$$\phi_4 \equiv y \neq z \wedge \text{Anc-self}(\text{l-child}(z), y) \wedge \bigvee_{p \in Q} \Big(\psi^p(y, y') \wedge \bigvee_{\substack{r, r_1, r_2 \in Q, \sigma' \in \Sigma \\ \delta(r_1, r_2, \sigma') = r}} \big[R_{\sigma'}(z) \wedge R_{r_1,p}(\text{l-child}(z), y') \wedge R_{r_2,q_2}(\text{r-child}(z), x_2) \wedge R_{q_1,r}(x_1, z) \big] \Big).$$

—If $y \neq z$ and y is a right descendant of z, the formula ϕ_3 is almost identical to ϕ_2.

Finally, the update formulae for the acceptence relation depend only on the new value of the relations R_q, for $q \in Q$. That is, for all $\sigma \in \Sigma$,

$$\phi^{\mathrm{ACC}}_{\mathrm{ins}_\sigma}(x) = \bigvee_{q \in F} \phi^{R_q}_{\mathrm{ins}_\sigma}(x, \mathrm{root})$$

and

$$\phi^{\mathrm{ACC}}_{\mathrm{reset}}(x) = \bigvee_{q \in F} \phi^{R_q}_{\mathrm{reset}}(x, \mathrm{root}).$$

■

6. BEYOND FORMAL LANGUAGES

The definitions given in Section **??** only concern dynamic problems for word structures. Following [**?**], we now extend these definitions to arbitrary structures. To this end, let γ be a vocabulary containing relation symbols of arbitrary arities. We assume that a structure over γ of size n has as universe $\{1, \ldots, n\}$. The empty structure over vocabulary γ of size n and only empty relations is denoted $E_n(\gamma)$.

The set of *abstract updates* $\Delta(\gamma)$ is defined as $\{\mathrm{ins}_R, \mathrm{del}_R \mid R \in \gamma\}$. A *concrete update* is a term of the form $\mathrm{ins}_R(i_1, \ldots, i_k)$ or $\mathrm{del}_R(i_1, \ldots, i_k)$, where $k = \mathrm{arity}(R)$. A concrete update is *applicable* in a structure of size n if $i_j \leq n$, for all $j \in [1, k]$. By $\Delta_n(\gamma)$ we denote the set of applicable concrete updates for structures over γ of size n. For a sequence $\alpha = \delta_1 \ldots \delta_k \in (\Delta_n(\gamma))^+$ of updates we define $\alpha(A)$ as $\delta_k(\ldots(\delta_1(A))\ldots)$, where $\delta(A)$ is the structure obtained from A by setting $R(i_1, \ldots, i_k)$ to true if $\delta = \mathrm{ins}_R(i_1, \ldots, i_k)$, and setting $R(i_1, \ldots, i_k)$ to false if $\delta = \mathrm{del}_R(i_1, \ldots, i_k)$.

Definition 6.1. *Let γ be a vocabulary and F be a set of γ-structures. The* dynamic problem *Dyn(F) is the set of all pairs (n, α), with $n > 0$ and $\alpha \in (\Delta_n(\gamma))^+$, such that $\alpha(E_n(\gamma)) \in F$. We call F the* underlying static problem *of Dyn(F).*

We now explain how a dynamic program operates on a dynamic problem. For a program P, there again is a program state S containing the current structure and auxiliary relations, one of which is ACC, which are updated according to the updates which occur and the update functions of P. The state S is *accepting* if $S \models \mathrm{ACC}$. Then, let $L(P) = \{(n, \alpha) \mid \alpha \in (\Delta_n(\gamma))^*$ and $\alpha(S_n(\gamma))$ is accepting$\}$, where $S_n(\gamma)$ denotes the structure $E_n(\gamma)$ extended with empty auxiliary relations.

A program P *accepts* a problem F if $L(P) = \mathrm{Dyn}(F)$. If $P \in \mathcal{C}$, for some dynamic complexity class $\mathcal{C}$, we also write $\mathrm{Dyn}(F) \in \mathcal{C}$.

Incomparability of FO and DynPROP. As we have seen in the previous sections, when restricted to monadic input schemas, DynPROP in a sense has the power of MSO. However, we show that if one binary relation is added, DynPROP cannot even capture first-order logic. This is also true if we allow the program to use precomputed functions from the set Succ.

Thus, we consider *alternating graphs*, encoded by the binary edge relation E and two unary relations A and B that form a decomposition of the set of nodes V into the set of *existential* and *universal* nodes, respectively. Given a node $s \in V$, the set of all *reachable* nodes $\mathrm{Reach}(s)$ is defined as the smallest set satisfying

—$s \in \text{Reach}(s)$;
—if $u \in A$ and there is a $v \in \text{Reach}(s)$ such that $(u, v) \in E$, then $u \in \text{Reach}(s)$; and
—if $u \in B$ and for all $v \in V$ with $(u, v) \in E$, we have $v \in \text{Reach}(s)$, then $u \in \text{Reach}(s)$.

We define ALT-REACH as the problem, given an alternating graph $G = (A, B, E)$ and two nodes s and t, is $t \in \text{Reach}(s)$. Note that ALT-REACH is P-complete (see, for example, [**?**]).

PROPOSITION 6.2. *Dyn(ALT-REACH)* $\notin$ DYNPROP(*Rel*, *Rel*)

Before proving the proposition, we state a lemma that describes an important property of DYNPROP programs. An *update sequence working on k-tuples* α is a sequence of updates over the (abstract) universe $\{1, ..., k\}$. Given a k-tuple $\vec{i} = (i_1, ..., i_k)$, $\alpha(\vec{i})$ denotes the sequence of updates one obtains when applying the updates on the elements of the k-tuple $\vec{i}$. So, instead of using the (abstract) universe element x the element i_x should be used. For example, the update $\text{ins}_R(1, 4, 2)$ results in an update $\text{ins}_R(i_1, i_4, i_2)$.

LEMMA 1. *Let α be a sequence of updates working on k-tuples. Let P be a* DYNPROP(*Rel*, *Rel*) *program, S a state of P and consider two tuples of elements $\vec{i} = (i_1, ..., i_k)$ and $\vec{j} = (j_1, ..., j_k)$ of S such that $\langle S, \vec{i}\rangle = \langle S, \vec{j}\rangle$. Then, $\langle \alpha(\vec{i})(S), \vec{i}\rangle = \langle \alpha(\vec{j})(S), \vec{j}\rangle$, that is, the type of $\vec{i}$ after applying $\alpha(\vec{i})$ and the type of $\vec{j}$ after applying $\alpha(\vec{j})$ are still the same. In particular, the value of the ACC-relation is the same in $\alpha(\vec{i})(S)$ and $\alpha(\vec{j})(S)$.*

PROOF. It suffices to consider one update operation δ working on k-tuples. Then the lemma follows by induction on the length of the update sequence α. Let ι be the tuple of elements in $\{1, \ldots, k\}$ which form the parameters of δ, and consider any (auxiliary) relation R updated by the program P on a tuple κ of elements also from $\{1, \ldots, k\}$. Let $\iota(\vec{i})$, $\iota(\vec{j})$, $\kappa(\vec{i})$ and $\kappa(\vec{j})$ denote the corresponding tuples in state S. Then, the evaluation of the update formula for R on $\kappa(\vec{i})$ after the operation δ with parameters $\iota(\vec{i})$ depends only on the type of the elements in $\kappa(\vec{i}) \cup \iota(\vec{i})$. The same holds for the tuples corresponding to $\vec{j}$. Since the types $\langle S, \vec{i}\rangle$ and $\langle S, \vec{j}\rangle$ are equal, the update formula evaluates to the same value. ∎

PROOF OF PROPOSITION **??**. We first define a family of alternating graphs $\mathcal{G} = \{G_m \mid m \in \mathbb{N}\}$. Every graph G_m consists of the following set V_m of nodes:

—two nodes s and t;
—a set of $2m$ nodes $P = \{p_1, ..., p_{2m}\}$;
—for each subset I of P of size m a node q_I, forming the set Q (of size $\binom{2m}{m}$); and
—for each subset J of Q a node r_J, forming the set R (of size $2^{|Q|}$).

All nodes are existential nodes except the nodes in set Q, which are universal. Thus, $A_m = V_m \setminus Q$ and $B_m = Q$. Further, the following set of edges E_m is already present in the graph G_m:

—for each subset I of P of size m, the set of edges $\{(q_I, p) \mid p \in I\}$ and
—for each subset J of Q the set of edges $\{(r_J, q) \mid q \in J\}$.

As updates we will only consider insertions of edges from s to nodes in the set R and from nodes in the set P to t.

We show that no dynamic program can maintain auxiliary relations such that it can incrementally answer, for every $n \in \mathbb{N}$, the question whether t is reachable from s in the alternating graph G_m, starting from G_m and arbitrary precomputation on the auxiliary relations. The proposition then follows.

We make use of the following two lemmas:

LEMMA 2. *For every m and every pair of distinct nodes $r, r' \in R$ of G_m, there exists a set $I \subset P$ of size m such that in the graph $G'_m := (A_m, B_m, E_m \cup \bigcup_{p \in I}(p, t))$, $t \in$ Reach(r) and $t \notin$ Reach(r').*

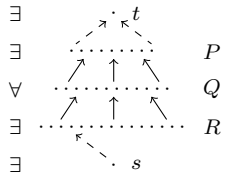

Fig. 5. An abstract illustration of the graph G_m. Full edges represent edges already present in G_m; dashed ones will be inserted later.

PROOF. Each of the nodes in R corresponds to a (unique) subset of Q. Hence, by definition of G_m, there is a set I and a node $q_I \in Q$ which in G_m is reachable from r but not from r'. We show that the set $I \subset P$ is the desired set, that is, for $G'_m := (A_m, B_m, E_m \cup \bigcup_{p\in I}(p,t))$, it holds that $t \in \text{Reach}(r)$ and $t \notin \text{Reach}(r')$. We note that in G'_m, q_I is the only node in the set Q such that $t \in \text{Reach}(q_I)$ because all nodes in Q are universal nodes. But now, as r and r' are existential nodes, and r is connected to q_I but r' is not, $t \in \text{Reach}(r)$ and $t \notin \text{Reach}(r')$. This concludes the proof. ∎

LEMMA 3. *The number of possible k-types of a structure with x auxiliary relations of maximal arity y is bounded by $2^{x\cdot k^y}$.*

PROOF. A k-type is constructed from a set of atoms $R(\vec{j})$ (where each element in $\vec{j}$ is in $[1,k]$) by adding either $R(\vec{j})$ or $\neg R(\vec{j})$ to the k-type. Hence, there exist at most $2^{|\text{atoms}|}$ different k-types where $|\text{atoms}|$ denotes the number of different atoms. For one y-ary relation R all atoms of R can be seen as the set of all y-tuples of elements in $\vec{i}$. So, one relation of arity y produces k^y different atoms. As there are x different relations, there are at most $x \cdot k^y$ atoms, and thus at most $2^{x\cdot k^y}$ different k-types. ∎

Now, assume, towards a contradiction, that there exists a dynamic program P for Dyn(ALT-REACH) in DYNPROP(Rel, Rel) that makes use of a auxiliary relations of maximal arity b. For a graph G_m and node $r \in R$, we consider the tuple $V_r := (s, t, r, p_1, ..., p_{2m})$.

Since

$$|Q| = \binom{2m}{m} = \prod_{i=0}^{m-1} \frac{2m-i}{m-i} \geq \prod_{i=0}^{m-1} \frac{2m}{m} = 2^m \text{ and so } |R| \geq 2^{2^m},$$

there exists a number m such that $|R|$ is bigger than the number of $(2m+3)$-types in any state S of P. Indeed, by Lemma **??** and since the program can use $a+6$ relations (the auxiliary relations, the input relations E, A, and B, and the equality, order and ACC relations) of maximal arity b, the number of $(2m+3)$-types in S is bounded by $2^{(a+6)\cdot(2m+3)^b}$. For a large enough value of m, this is clearly dominated by 2^{2^m}. Fix such an m and corresponding graph G_m, and let S be a state P is in when the current graph is G_m. Then, by the above reasoning, in the set R (of G_m) there must exist two distinct elements r and r' such that $\langle S, V_r\rangle = \langle S, V_{r'}\rangle$.

According to Lemma **??**, there is a set I of m elements in P such that after the insertion of all edges $\{(p,t) \mid p \in I\}$ in G_m, $t \in \text{Reach}(r)$ and $t \notin \text{Reach}(r')$. Let $I = \{p_{i_1}, ..., p_{i_m}\}$ and consider the two sequences of update operations

$$\begin{aligned} \alpha &= (\text{ins}_E(p_{i_1}, t), ..., \text{ins}_E(p_{i_m}, t), \text{ins}_E(s, r)) \\ \alpha' &= (\text{ins}_E(p_{i_1}, t), ..., \text{ins}_E(p_{i_m}, t), \text{ins}_E(s, r')). \end{aligned}$$

Notice that $\alpha(G_m)$ yields a graph in which $t \in \text{Reach}(s)$, whereas $t \notin \text{Reach}(s)$ in $\alpha'(G_m)$. However, as $\langle S, V_r\rangle = \langle S, V_{r'}\rangle$, it follows from Lemma **??** that also $\langle \alpha(S), V_r\rangle = \langle \alpha'(S), V_{r'}\rangle$. Hence, P will either in both cases claim that $t \in \text{Reach}(s)$ (if ACC holds in $\langle \alpha(S), V_r\rangle$) or claim in both cases that $t \notin \text{Reach}(s)$. We can conclude that there does not exist a DYNPROP(Rel, Rel) program for ALT-REACH. ∎

This proof can be adapted to show that even with a precomputed successor-function, one cannot maintain the reachability problem for alternating graphs.

PROPOSITION 6.3. *Dyn(ALT-REACH)* $\notin$ DYNPROP(*Succ, Rel*)

In order to prove this we need an observation similar to Lemma **??** for DYNPROP(Succ, Rel). For an element i and a number l, let the *l-neighborhood* of i, denoted $\mathcal{N}_l(i)$, be the following tuple of elements:

$$\left(\mathrm{pre}^l(i), \mathrm{pre}^{l-1}(i), ..., \mathrm{pre}(i), i, \mathrm{succ}(i), ..., \mathrm{succ}^{l-1}i, \mathrm{succ}^l(i)\right).$$

For a tuple of elements $\vec{i}$, we denote by $\mathcal{N}_l(\vec{i})$ the tuple $(\mathcal{N}_l(\min), \mathcal{N}_l(i_1), ..., \mathcal{N}_l(i_k))$.

LEMMA 4. *Let α be a sequence of updates working on k-tuples which consists of l updates. For each* DYNPROP(*Succ, Rel*) *program P, there exists a number c, depending only on P, such that the following holds: Let S be a state of P, and $\vec{i} = (i_1, ..., i_k)$ and $\vec{j} = (j_1, ..., j_k)$ tuples of elements of S such that $\langle S, \mathcal{N}_{c\cdot l}(\vec{i})\rangle = \langle S, \mathcal{N}_{c\cdot l}(\vec{j})\rangle$. Then, $\langle \alpha(\vec{i})(S), \vec{i}\rangle = \langle \alpha(\vec{j})(S), \vec{j}\rangle$.*

PROOF. The c of the lemma is the maximal nesting dephth of the functions succ and pre used in P. For example the term succ(succ(pre(x))) has nesting depth 3. Let α_n be the prefix of length n of the update sequence α. We will here prove the slightly stronger statement that, assuming the conditions of the lemma, $\langle \alpha_n(\vec{i})(S), \mathcal{N}_{c\cdot(l-n)}(\vec{i})\rangle = \langle \alpha_n(\vec{j})(S), \mathcal{N}_{c\cdot(l-n)}(\vec{j})\rangle$. Then the lemma follows because $\mathcal{N}_0(\vec{i}) = (\min, \vec{i})$. The proof works by induction on n (assuming $n < l$). For $n = 0$ the statement is contained in the condition of the lemma. So, assume the statement holds for $n < l$, we show that it still holds for $n + 1$. Let δ be the update such that $\alpha_{n+1} = \alpha_n\delta$. Just as in the proof of Lemma **??** consider any (auxiliary) relation R updated by the program P on elements in $\mathcal{N}_{c\cdot(l-(n+1))}(\vec{i})$. The evaluation of the update formula for R depends only on the type of $\mathcal{N}_{c\cdot(l-n)}(\vec{i})$. This is true because one can reach other elements in the universe only by using the functions. Since these are nested at most c times, from any element in $\mathcal{N}_{c\cdot(l-(n+1))}(\vec{i})$ only elements in $\mathcal{N}_{c\cdot(l-n)}(\vec{i})$ can be reached. The same holds for the tuples corresponding to $\vec{j}$. As $\langle \alpha_n(\vec{i})(S), \mathcal{N}_{c\cdot(l-n)}(\vec{i})\rangle = \langle \alpha_n(\vec{j})(S), \mathcal{N}_{c\cdot(l-n)}(\vec{j})\rangle$, and $\alpha_{n+1} = \alpha_n\delta$, it follows that $\langle \alpha_{n+1}(\vec{i})(S), \mathcal{N}_{c\cdot(l-(n+1))}(\vec{i})\rangle = \langle \alpha_{n+1}(\vec{j})(S), \mathcal{N}_{c\cdot(l-(n+1))}(\vec{j})\rangle$. ∎

PROOF OF PROPOSITION **??**. The proof is along the same lines as that of Theorem **??**. Assume that there exists a DYNPROP(Succ, Rel) program P for Dyn(ALT-REACH) making use of a auxiliary relations of maximal arity b. We will again consider (in a graph G_m) the tuples V_r and $V_{r'}$, but now their corresponding $(m+1)c$-neighborhoods $\mathcal{N}_{(m+1)c}(V_r)$ and $\mathcal{N}_{(m+1)c}(V_{r'})$, where c is the constant of Lemma **??** which only depends on P. Using Lemma **??**, we know that in any state S of P the number of types of these neighborhoods is bounded by $2^{(a+6)\cdot((2(m+1)c+1)(2m+3+1))^b}$. Hence we can again find a number m big enough such that there are distinct $r, r' \in R$ in G_m such that $\langle S, \mathcal{N}_{(m+1)c}(V_r)\rangle$ and $\langle S, \mathcal{N}_{(m+1)c}(V_{r'})\rangle$. Using the same argument as above and Lemma **??**, we get the desired contradiction. ∎

REMARK 6.4. *The proofs of the foregoing lemma and proposition depend heavily on the fact that, with each update operation, the neighborhood of a tuple increases only by a constant term. This is because the functions pre and succ are complementary in the sense that pre(succ) = succ(pre). So, the order of their usage is not important. If one allows two independent functions (for example, two different successor-functions on the universe) the size of the neighborhood possibly doubles after each operation so the proof of the proposition (based on a counting argument) does not work.*

From the proof of the above proposition one can conclude an even stronger statement. The graphs used in the proof are very restricted in the sense that the length of the longest path is bounded by a constant. Let ALT-REACH$_{\mathrm{depth}\leq d}$ be the alternating reachability problem on graphs of depth at most d. It is easily seen that ALT-REACH$_{\mathrm{depth}\leq d}$ is expressible by a FO-formula, so we get the following

THEOREM 6.5. *There exists a problem $F \in FO$ such that Dyn(F)* $\notin$ DYNPROP(*Succ, Rel*). ∎

On the other hand, the reachability problem on acyclic deterministic directed graphs can be maintained in DynPROP [**?**] but cannot be expressed in FO. So these classes are incomparable.

Using functions to maintain EFO. Next, we exhibit a class of properties which can be maintained in DynQF with precomputation. An *existential first-order (EFO)* sentence is a first-order sentence of the form $\exists x_1 \cdots \exists x_k \phi(\vec{x})$, where $\phi(\vec{x})$ is a quantifier free formula.

THEOREM 6.6. *For any EFO-definable problem F, $Dyn(F) \in$ DynPROP(Fun, Fun)*

PROOF. Let $\psi = \exists x_1 \cdots \exists x_k \phi(\vec{x})$ be an EFO-sentence over vocabulary γ. We show that there exists a DynPROP(Fun, Fun) program P which maintains whether $A \models \psi$, for any γ-structure A.

We first introduce some notation. A tuple $\vec{i} = (i_1, \ldots, i_l)$ is *disjoint* if $i_j \neq i_k$, for all $j, k \in [1, \ell]$, with $j \neq k$. A *disjoint type* is the type of a disjoint tuple. For a type τ, let ϕ_τ be an EFO sentence which is satisfied in a structure A if and only if A contains a tuple $\vec{x}$ such that $\langle A, \vec{x} \rangle = \tau$.

It is well known and easy to see that for any EFO sentence $\phi = \exists x_1 \cdots \exists x_k \psi(\vec{x})$, there exists a set θ_ψ of disjoint ℓ-types, with ℓ ranging from 1 to k, such that ψ is equivalent to $\bigvee_{\tau \in \theta} \phi_\tau$. Note that if we would not require the types to be disjoint, we would only need to consider k-types, and not ℓ-types, for all $\ell \leq k$. However, the latter restriction, and corresponding extension, will prove technically more convenient.

Using the information that $A \models \phi$ is completely determined by the set of types θ_ψ realized in A, we now present our dynamic algorithm. It will maintain the following functions. For every disjoint ℓ-type τ, with $\ell \leq k$, and set $I = \{i_1, \ldots, i_{|I|}\} \subseteq \{1, \ldots, \ell\}$, let

$$f_\tau^I(x_1, \ldots, x_{|I|}) = |\{(a_1, \ldots, a_\ell) \mid \langle A, \vec{a} \rangle = \tau \wedge \forall j \in [1, |I|] : a_{i_j} = x_j\}|$$

Here, we write $I = \{i_1, \ldots, i_{|I|}\}$ such that $i_j < i_{j+1}$, for all $j \in [1, |I| - 1]$. Then, for $I = \emptyset$, $f_\tau^\emptyset$ defines the number of disjoint tuples in A which have type τ. When $I = \{i_1, \ldots, i_{|I|}\} \neq \emptyset$, and given $\vec{x} = (x_1, \ldots, x_{|I|})$, $f_\tau^I(\vec{x})$ defines the number of tuples in A which (1) have type τ and (2) have at position i_j exactly element x_j, for all $j \in [1, |I|]$.

Notice that the numbers defined by the above functions can become bigger than n, the number of universe elements, but are always smaller than n^k. Hence, every such number can be encoded as a number with k digits in base n, which is exactly how our functions will encode these numbers. Thus, for every function f_τ^I mentioned above, there are actually k functions $f_\tau^{I,1}, \ldots, f_\tau^{I,k}$, each defining one digit of the desired number defined by f_τ^I. For clarity, we use the functions f_τ^I instead of the actual ones encoding their digits.

As we are in the setting where precomputation is allowed, we can assume that the functions are properly initialized. For any $l \in [1, k]$, let $\tau_\neg$ be the unique l-type containing only negated atoms, that is, atoms of the form $\neg R(\vec{i})$. Then, for all l-types $\tau \neq \tau_\neg$, set I, and tuple $\vec{x}$, initially

$$f_\tau^I(\vec{x}) = 0\,,$$

and for $\vec{x} = (x_1, \ldots, x_{|I|})$ it holds that

$$f_{\tau_\neg}^I(\vec{x}) = 0 \text{ if } x_i = x_j, \text{ for some } i \neq j,$$

and

$$f_{\tau_\neg}^I(\vec{x}) = (n - |I|) \cdot (n - (|I| + 1)) \cdot \cdots \cdot (n - l), \text{ otherwise.}$$

We now show how to incrementally maintain these functions. To this end, we give the precomputed functions and relations which will be used for the updates. For simplicity, we assume the universe of size n consists of the elements $\{0, \ldots, n-1\}$. Then, there is a constant (0-ary function) min denoting 0, functions plus and minus such that $\text{plus}(x, y) = x + y \pmod n$ and $\text{minus}(x, y) = x - y \pmod n$, and accompanying relations R_{plus} and R_{minus} such that $R_{\text{plus}}(x, y)$ holds if and only if $x + y \geq n$, and $R_{\text{minus}}(x, y)$ holds if and only if $x - y < 0$. That is, the functions plus and minus are defined on all parameters and count modulo n. The accompanying relations R_{plus} and R_{minus} contain the additional information saying whether the addition or subtraction indeed

went above $n-1$ or below 0. These functions allow to define addition and subtraction on the k-digit base-n numbers used in the functions. Therefore, we simply perform addition and subtraction on these numbers in the sequel.

Second, we introduce some additional notation. As before, we write $\vec{x}$ for a tuple of elements, but abuse notation and also denote the set of elements in $\vec{x}$ by $\vec{x}$, and, correspondingly, apply set-theoretic operations on them, for instance, $\vec{x} \cup \vec{y}$.

Further, for an integer ℓ, set $I = \{i_1, \ldots, i_{|I|}\} \subseteq \{1, \ldots, \ell\}$, and tuples $\vec{x} = (x_1, \ldots, x_{|I|})$ and $\vec{y}$, we let an *indexing* for $\ell, I, \vec{x}, \vec{y}$ be a function $\text{ind} : \vec{x} \cup \vec{y} \to \{1, \ldots, \ell\}$ such that for all $j \in [1, |I|]$, $\text{ind}(x_j) = i_j$. The indexing ind is *proper* if for all $z, z' \in \vec{x} \cup \vec{y}$, $\text{ind}(z) = \text{ind}(z')$ if and only if $z = z'$. Hence, a proper indexing ind associates elements of $\vec{x}$ to their corresponding elements in I, and associates elements of $\vec{y}$ to elements of $\{1, \ldots, \ell\}$ such that elements have the same index if and only if they are equal. Notice that while the fact whether a function ind is an indexing only depends on I and ℓ, whether it is proper depends on the actual values of $\vec{x}$ and $\vec{y}$. However, this can easily be tested by the following formula:

$$\phi_{\text{ind}}(\vec{y}, \vec{x}) = \bigwedge_{\substack{z,z' \in \vec{x} \cup \vec{y} \\ \text{ind}(z) = \text{ind}(z')}} z = z' \wedge \bigwedge_{\substack{z,z' \in \vec{x} \cup \vec{y} \\ \text{ind}(z) \neq \text{ind}(z')}} z \neq z'$$

Given $\vec{x}$ and $\vec{y}$ and a proper indexing ind, we write $(\vec{x}, \vec{y})_{\text{ind}}$ for the sequence $(u_1, \ldots, u_m)$, for some m, such that (1) $\vec{u}$ contains every element in $\vec{x} \cup \vec{y}$ exactly once and (2) $\text{ind}(u_i) < \text{ind}(u_{i+1})$, for all $i \in [1, m-1]$. Hence, $\vec{u}$ is obtained from $\vec{x} \cup \vec{y}$ by eliminating elements which are equal (and thus have the same index), and ordering the elements by their index. Further, we write $\text{ind}(\vec{y})$ to denote the tuple $(\text{ind}(y_1), \ldots, \text{ind}(y_m))$. Finally, for a type τ, and $R(\vec{i}) \notin \tau$, let $\tau + R(\vec{i})$ denote the type obtained from τ by removing $\neg R(\vec{i})$ and adding $R(\vec{i})$. When $\neg R(\vec{i}) \notin \tau$, $\tau + \neg R(\vec{i})$ is defined similarly by removing $R(\vec{i})$ and adding $\neg R(\vec{i})$.

We are now ready to give the update functions. For clarity, we write the $\textbf{ite}(\phi, t_1, t_2)$ construct as "if ϕ then t_1 else t_2". Then, for relation symbol R, ℓ-type τ, with $\ell \leq k$, and $I \subseteq \{1, \ldots, \ell\}$, let

$$\begin{aligned}\phi^{f^I_\tau}_{\text{ins}_R}(\vec{y}; \vec{x}) \equiv f^I_\tau(\vec{x}) \\ + \sum_{\substack{\text{ind for } \ell, I, \vec{x}, \vec{y} \\ R(ind(\vec{y})) \in \tau}} \text{if } \phi_{\text{ind}}(\vec{y}; \vec{x}) \text{ then } f^{I \cup \text{ind}(\vec{y})}_{\tau + \neg R(ind(\vec{y}))}(\vec{x}, \vec{y})_{\text{ind}} \text{ else } 0 \\ - \sum_{\substack{\text{ind for } \ell, I, \vec{x}, \vec{y} \\ \neg R(ind(\vec{y})) \in \tau}} \text{if } \phi_{\text{ind}}(\vec{y}; \vec{x}) \text{ then } f^{I \cup \text{ind}(\vec{y})}_{\tau}(\vec{x}, \vec{y})_{\text{ind}} \text{ else } 0\end{aligned}$$

and, similarly,

$$\begin{aligned}\phi^{f^I_\tau}_{\text{del}_R}(\vec{y}, \vec{x}) \equiv f^I_\tau(\vec{x}) \\ + \sum_{\substack{\text{ind for } \ell, I, \vec{x}, \vec{y} \\ \neg R(ind(\vec{y})) \in \tau}} \text{if } \phi_{\text{ind}}(\vec{y}; \vec{x}) \text{ then } f^{I \cup \text{ind}(\vec{y})}_{\tau + R(ind(\vec{y}))}(\vec{x}, \vec{y})_{\text{ind}} \text{ else } 0 \\ - \sum_{\substack{\text{ind for } \ell, I, \vec{x}, \vec{y} \\ R(ind(\vec{y})) \in \tau}} \text{if } \phi_{\text{ind}}(\vec{y}; \vec{x}) \text{ then } f^{I \cup \text{ind}(\vec{y})}_{\tau}(\vec{x}, \vec{y})_{\text{ind}} \text{ else } 0.\end{aligned}$$

Intuitively, both formulas compute the number of tuples with the given type τ in the same manner: Take the number of tuples which used to have type τ, add those which obtained type τ by the update, and remove the ones which had type τ, but do not anymore.

We briefly explain the correctness of these formulas by arguing that after an update $\text{ins}_R(\vec{y})$ for a tuple $\vec{x}$ the number of tuples which did not have type τ but do after the update is indeed equal to the number computed on the second line of the update formula $\phi^{f^I_\tau}_{\text{ins}_R}(\vec{y};\vec{x})$.

Let $\vec{a} = (a_1, \ldots, a_l)$ be a disjoint tuple consistent with $\vec{x}$ and I, that is, for all $j \in [1, |I|]$, $x_j = a_{i_j}$. We denote the structure obtained from A after the update $\text{ins}_R(\vec{y})$ by A'. Now, suppose $\langle A, \vec{a}\rangle \neq \tau$, but $\langle A', \vec{a}\rangle = \tau$. This can only hold if $\vec{y} \subseteq \vec{a}$ and thereby the insertion of $R(\vec{y})$ has changed the type of $\vec{a}$ in A. More precisely, if we define $\vec{k} = k_1, \ldots, k_m$ such that for all $j \in [1, m]$, $y_j = a_{k_j}$, then $\langle A, \vec{a}\rangle = \tau + \neg R(\vec{k})$ must hold. Notice also that $\vec{k}$ is uniquely defined because $\vec{a}$ is disjoint. Now $\vec{k}$, in turn, defines a proper indexing ind on $\vec{x}$ and $\vec{y}$ as follows: for all $j \in [1, |I|]$, $\text{ind}(x_j) = i_j$ (by definition) and for all $j \in [1, m]$, $\text{ind}(y_j) = k_j$. In this manner we can thus associate a unique proper indexing to all tuples $\vec{a}$ which did not have type τ, but do now. Then, for any indexing ind, the expression $f^{I \cup \text{ind}(\vec{y})}_{\tau + \neg R(ind(\vec{y}))}(\vec{x}, \vec{y})_{\text{ind}}$ defines exactly all such tuples with which ind is associated. By iterating over all proper indexings we hence count exactly all desired tuples.

Finally, for the acceptance relation we have to check whether there is a tuple in the new structure which has a type contained in θ_ψ:

$$\phi^{\text{ACC}}_{\text{ins}_R}(\vec{y}) \equiv \bigvee_{\tau \in \theta_\phi} \phi^{f^\emptyset_\tau}_{\text{ins}_R}(\vec{y}) \neq 0 \qquad \text{and} \qquad \phi^{\text{ACC}}_{\text{del}_R}(\vec{y}) \equiv \bigvee_{\tau \in \theta_\phi} \phi^{f^\emptyset_\tau}_{\text{del}_R}(\vec{y}) \neq 0.$$

■

7. CONCLUSION

We have studied the dynamic complexity of formal languages and, by characterizing the languages maintainable in DynPROP as exactly the regular languages, obtained the first lower bounds for DynPROP. This yields a separation of DynPROP from DynQF and DynFO. We proved that every context-free language can be maintained in DynFO and investigated the power of functions for dynamic programs in maintaining specific context-free and non context-free languages.

As a modest extension we also proved a lower bound for DynPROP with built-in successor functions. Hence, we are now one step closer to proving lower bounds for DynFO, but, of course, a number of questions arise:

—Can the results on the Dyck languages be extended to show that an entire subclass of the context-free languages, such as the deterministic or unambiguous context-free languages, can be maintained in DynQF?

—We showed that $D_1 \in \text{DynPROP}(\text{Succ}, \text{Rel})$. Can it be shown that $D_2 \notin \text{DynPROP}(\text{Succ}, \text{Rel})$?

—Can some of the lower bound techniques for DynPROP be extended to DynQF, in order to separate DynQF from DynFO, or at least from DynP? Is there a context-free language that is not maintainable in DynQF?

REFERENCES

Balmin, A., Papakonstantinou, Y., and Vianu, V. 2004. Incremental validation of XML documents. *ACM Trans. Database Syst. 29,* 4, 710–751.

Barbosa, D., Mendelzon, A. O., Libkin, L., Mignet, L., and Arenas, M. 2004. Efficient incremental validation of XML documents. In *ICDE*. 671–682.

Björklund, H., Gelade, W., Marquardt, M., and Martens, W. 2009. Incremental xpath evaluation. In *ICDT*. 162–173.

Dong, G., Libkin, L., and Wong, L. 2003. Incremental recomputation in local languages. *Inf. Comput. 181,* 2, 88–98.

Dong, G. and Su, J. 1997. Deterministic FOIES are strictly weaker. *Annals of Mathematics and Artificial Intelligence 19,* 1-2, 127–146.

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

Dong, G., Su, J., and Topor, R. W. 1995. Nonrecursive incremental evaluation of datalog queries. *Annals of Mathematics and Artificial Intelligence 14,* 2-4, 187–223.

Etessami, K. 1998. Dynamic tree isomorphism via first-order updates to a relational database. In *Proceedings of PODS '98*. 235–243.

Frandsen, G. S., Husfeldt, T., Miltersen, P. B., Rauhe, T., and Skyum, S. 1995. Dynamic algorithms for the Dyck languages. In *WADS*. 98–108.

Frandsen, G. S., Miltersen, P. B., and Skyum, S. 1997. Dynamic word problems. *J. ACM 44,* 2, 257–271.

Graham, R. L. and Rothschild, B. L. 1990. *Ramsey theory (2nd ed.)*. Wiley-Interscience, New York, NY, USA.

Hesse, W. 2003a. Conditional and unconditional separations of dynamic complexity classes. Unpublished manuscript, available from http://people.clarkson.edu/ whesse/ (seen Dec, 9, 2008).

Hesse, W. 2003b. The dynamic complexity of transitive closure is in DynTC^0. *Theor. Comput. Sci. 3,* 296, 473–485.

Hesse, W. 2003c. Dynamic computational complexity. Ph.D. thesis, University of Massachusetts Amherst.

Hesse, W. and Immerman, N. 2002. Complete problems for dynamic complexity classes. In *LICS*. 313–324.

Miltersen, P. B. 1999. Cell probe complexity - a survey. In *Advances in Data Structures. Satellite Workshop of the 19th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*.

Miltersen, P. B., Subramanian, S., Vitter, J. S., and Tamassia, R. 1994. Complexity models for incremental computation. *Theor. Comput. Sci. 130,* 1, 203–236.

Patnaik, S. and Immerman, N. 1997. Dyn-FO: A parallel, dynamic complexity class. *J. Comput. Syst. Sci. 55,* 2, 199–209.

Vollmer, H. 1999. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Weber, V. and Schwentick, T. 2007. Dynamic complexity theory revisited. *Theory Comput. Syst. 40,* 4, 355–377.