Symposium on Theoretical Aspects of Computer Science 2009 (Freiburg), pp. 481–492
www.stacs-conf.org

THE DYNAMIC COMPLEXITY OF FORMAL LANGUAGES

WOUTER GELADE [1] AND MARCEL MARQUARDT [2] AND THOMAS SCHWENTICK [2]

[1] Hasselt University and Transnational University of Limburg, School for Information Technology

[2] Technische Universität Dortmund

Abstract. 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 exactly are 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.

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. In contrast, *dynamic complexity* 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 [18]. 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 [8]). 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, the Dyck languages D_n. Subsequent work has yielded more problems in DynFO [8] some of which are LOGCFL-complete [20] and even PTIME-complete [17, 18] (even though the latter are highly artificial). Other work also considered stronger classes (like Hesse's result that Reachability in arbitrary directed graphs is in DynTC^0 [13]), studied

Key words and phrases: Dynamic complexity, Regular languages, Context-free languages, DynFO.
Wouter Gelade is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) .

$$\begin{tikzcd}
 & \text{FO} & \text{EFO} & \\
\text{DynPROP} \arrow[r, phantom, "\subsetneq", "\text{Thm 3.5, 4.5}"'] \arrow[d, phantom, "\|", "\text{Thm 3.5}"'] & \text{DynPROP} + \mathcal{SUCC} \arrow[r, phantom, "\subseteq"] \arrow[u, phantom, "\rotatebox{90}{$\nparallel$}", "\text{Thm 6.3}"'] \arrow[d, phantom, "\rotatebox{90}{$\Supset$}", "\text{Prop 4.5}"'] & \text{DynQF} \arrow[r, phantom, "\subseteq"] \arrow[u, phantom, "\rotatebox{90}{$\subsetneq$}", "\text{Prop 6.4}"'] \arrow[d, phantom, "\rotatebox{90}{$\Supset$}", "\text{Prop 4.4}"'] & \text{DynFO} \arrow[d, phantom, "\rotatebox{90}{$\not\supseteq$}", "\text{Thm 4.1, Prop 4.2}"'] \\
\text{REG} & D_1 & D_n & \text{CFL}
\end{tikzcd}$$

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

notions of completeness for dynamic problems [15], and elaborated on the handling of precomputations [20].

The choice of first-order logic as update language in [18] 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 have to show 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 [12, 14]. As further refinements the subclasses DynOR and DynProjections were studied. In [12] separation results for subclasses of DynPROP were shown and the separation between DynPROP and DynP was stated as an open problem.

The framework of [18] 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, [18] showed that regular languages and Dyck languages can be maintained in DynFO. Later on, Hesse proved that the dynamic complexity of regular languages is actually in DynQF [14].

Databases: The dynamic complexity of database properties were studied in the slightly different framework of First-Order Incremental Evaluation Systems (FOIES) [7]. Many interesting results were shown including a separation between deterministic and nondeterministic systems [5] and inexpressibility results for auxiliary relations of small arity [4, 6]. Nevertheless, 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 1):

- 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 3)

[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.

- We provide (presumably) better 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 4)
- 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 semantics for dynamic string languages, and we show that regular tree languages can be maintained in DynPROP, when allowing precomputation and the use of built-in functions. (Section 5).
- Further, we study general structures, and show that (bounded-depth) alternating reachability is not maintainable in DynPROP. From this we can conclude 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 6)

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 [16, 17]. In particular, [9, 10] 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) [1, 2] and XPath query evaluation [3] in XML documents. There, the interest is mainly in fast algorithms, less in structural dynamic complexity. Nevertheless techniques of dynamic algorithms on string languages also find applications in these settings.

Due to lack of space all proofs are omitted, except for some proof sketches. They are available in the full version of the paper [11].

2. Definitions

Let $\Sigma = \{\sigma_1, ..., \sigma_k\}$ be a fixed alphabet. We represent words over Σ encoded by *word structures*, i.e., logical structures W with universe $\{1 \ldots, n\}$, one unary relation R_σ for each symbol $\sigma \in \Sigma$, and the canonical linear order $<$ on $\{1 \ldots, n\}$. 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. We call n the *size* of W.

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, due to the fact that 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, resp. 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. Then we define dynamic programs which are intended to 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 [18] 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 $\mathrm{Dyn}(L)$. The idea is that words can be changed by insertions and deletions of letters and $\mathrm{Dyn}(L)$ 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 $\Delta := \{\mathrm{ins}_\sigma \mid \sigma \in \Sigma\} \cup \{\mathrm{reset}\}$ of *abstract updates.* A *concrete update* is a term of the form $\mathrm{ins}_\sigma(i)$ or $\mathrm{reset}(i)$, 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 $\delta = \mathrm{ins}_\sigma(i)$, and
- setting all $R_\sigma(i)$ to false, if $\delta = \mathrm{reset}(i)$.

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* $\mathrm{Dyn}(L)$ is the set of all (non-empty) sequences α of updates, for which there is $n > 0$ such that $\alpha \in \Delta_n^+$ and $\mathrm{word}(\alpha(E_n)) \in L$. We call L the *underlying language* of $\mathrm{Dyn}(L)$.[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* $\phi^R_{\mathrm{op}}(y; x_1, \dots, x_k)$, for every $\mathrm{op} \in \Delta$ and $R \in \mathcal{R}$, where $k = \mathrm{arity}(R)$. A dynamic program P operates as follows. Let S be a program state with word structure W. The application of an applicable update $\delta = \mathrm{op}(i)$ on S yields the new state $S' = \delta(S)$ consisting of $W' = \delta(W)$ and new relations $R' = \{\bar{j} \mid S \models \phi^R_{\mathrm{op}}(i, \bar{j})\}$, 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 $\delta_k(\dots(\delta_1(S))\dots)$. We say that a state S is *accepting* iff $S \models \mathrm{ACC}$, i.e., if the 0-ary ACC-relation contains the empty tuple.[3]

[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.

We say that a dynamic program P *recognizes* the dynamic language $\mathrm{Dyn}(L)$ if for all $n \in \mathbb{N}$ and all $\alpha \in \Delta_n^+$ it holds that $\alpha(E_n{}')$ is accepting iff $\mathrm{word}(\alpha(E_n)) \in L$, where $E_n{}'$ 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.

Extending dynamic programs with functions. A dynamic program P *with auxiliary functions* 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 $\psi_\sigma^F(i; x_1, ..., x_k)$ where $k = \mathrm{arity}(F)$.

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

- Every x_i is an update term.
- If $F \in \mathcal{R}$ is a function and $\bar{t}$ contains only update terms then $F(\bar{t})$ is an update term.
- 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 otherwise stated, the functions in the initial state $E_n{}'$ map every tuple to its first element.

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

In [18] 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 [20]. 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 $E_n{}'$ 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, i.e., 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 $I \in \{\bot, \mathrm{Rel}, \mathrm{Fun}\}$ and $A \in \{\mathrm{Rel}, \mathrm{Fun}\}$ we denote by $\mathrm{DynC}(I, A)$ the class of dynamic languages recognized by dynamic programs

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

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

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

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

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

We will 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 to predecessors and the element 1 to itself, and let MIN be the constant (i.e. nullary function) mapping to the minimal element 1 in the universe. Then $\mathrm{DynPROP}(\mathcal{SUCC}, \mathrm{Rel})$ 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.

3. Dynamic Complexity of Regular Languages

As already mentioned in the introduction, it was shown by Patnaik and Immerman [18] that every regular language can be recognized by a DynFO program. Hesse [14] showed that 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. Consider the regular language $(a + b)^*a(a + b)^*$ over the alphabet $\{a, b\}$. One has to maintain one binary relation $A(i, j)$ that is true iff $i < j$ and there exists $k \in]i, j[$ such that $w[k, k] = a$ and two unary relations $I(j) \equiv \exists k < j : w[k, k] = a$ and $F(i) \equiv \exists k > i : w[k, k] = a$. We note that it is important here that $A(i, j)$ refers to the interval $]i, j[$, and not $[i, j]$, in order to maintain these relations in DynPROP.

For the operation ins_a the update formulas are straightforward, therefore we will give here just the update formulas for A and ACC after the operation ins_b. Exactly the same formulas can be used for the reset operation.

$$\begin{array}{lcl} \phi^{A}_{\text{ins}_b}(y; x_1, x_2) & \equiv & \big(y \notin]x_1, x_2[\wedge A(x_1, x_2)\big) \\ & & \vee \big(y \in]x_1, x_2[\wedge (A(x_1, y) \vee A(y, x_2))\big) \\ \phi^{\text{ACC}}_{\text{ins}_b}(y) & \equiv & I(y) \vee F(y) \end{array}$$

■

Proposition 3.2. *For every regular language L, Dyn(L) $\in$ DynPROP.*

Proof. Let $A = (Q, \delta, s, F)$ be a DFA accepting L with transition function $\delta : Q \times \Sigma \to Q$. Let $\delta^* : Q \times \Sigma^* \to Q$ denote the reflexive-transitive closure of δ.

Like in the example above one has to maintain information about the open intervals $]i, j[$, namely whether the substring $w[i + 1, j - 1]$ brings the automaton from a state p to state q. Here, we only give the different auxiliary relations. For all states p and q we maintain

- $R_{p,q} = \{(i, j) \mid i < j \wedge \delta^*(p, w[i + 1, j - 1]) = q\}$
- $I_q = \{j \mid \delta^*(s, w[1, j - 1]) = q\}$ and $F_p = \{i \mid \delta^*(p, w[i + 1, n]) \in F\}$ ■

As a matter of fact, the converse of Proposition 3.2 is also true, thus DynPROP is the exact dynamic counterpart of the regular languages.

Proposition 3.3. *Let L = 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 L and see what happens if, 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 sequence of updates used to produce the word, it suffices to care only about this one update sequence.

We make the following observations.

(1) After each update all tuples of positions that have not been set yet behave the same with respect to the auxiliary relations.
(2) There is only a bounded number (depending only on P, namely on the number and the maximal arity of the auxiliary relations) of possible ways these tuples behave.
(3) The change in behavior of the tuples by one update is uniquely determined by the inserted symbol.

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

Remark 3.4. Proposition 3.3 is a powerful tool for proving lower bounds as it, of course, shows that, for every non-regular language L, Dyn(L) $\notin$ DynPROP.

The proof of Proposition 3.3 intuitively relies on the fact that all remaining string positions cannot be distinguished before they are set. 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. This fact and the above two propositions can then be combined into the following theorem.

Theorem 3.5. *Let L be a language. Then, the following are equivalent:*

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

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 on, 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. Consider a grammar $G = (V, \Sigma, S, D)$ for L and assume w.l.o.g. that is in Chomsky normal form. For $U \in V$, and $w \in (V \cup \Sigma)^*$, we denote by $U \rightarrow^* w$ that w can be derived from U. Then, $L = \{w \mid w \in \Sigma^* \wedge S \rightarrow^* w\}$.

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

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

■

However, we cannot hope for an equivalence between DynFO and the context-free languages, as for DynPROP and the regular languages before. This follows easily as opposed to the class of context-free languages, DynFO is closed under intersection and complement. Furthermore, one can show that non-contextfree languages can be maintained in DynQF and DynPROP($\mathcal{SUCC}$, Rel). This is because unary counters can be implemented easily by dynamic programs in these classes. Let EQUAL$_r$ be the language over the alphabet $\Sigma = \{a_1, \ldots, a_r\}$ containing all strings with an equal number of occurrences of each symbol a_i. Note that already EQUAL$_3$ is not context-free. Using the counters one can prove the following

Proposition 4.2.

(1) Dyn(EQUAL$_r$) $\in$ DynPROP($\mathcal{SUCC}$, Rel)
(2) Dyn(EQUAL$_r$) $\in$ DynQF ■

From Proposition 4.2 and Theorem 3.5 one can conclude the following

Corollary 4.3.

(1) DynPROP $\subsetneq$ DynPROP($\mathcal{SUCC}$, Rel)
(2) DynPROP $\subsetneq$ DynQF ■

One can also get better upper bounds for 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:

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

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

The proof of Proposition 4.4 mainly relies on the fact that each terminal symbol occurs in only one rule of the grammar and therefore corresponding positions (i.e. matching brackets) can be maintained using auxiliary functions. Together with relations similar to the proof of Theorem 4.1 these functions enable us to maintain D_n.

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.

It turns out that for Dyck languages with only one kind of brackets, i.e., D_1, auxiliary functions are not needed, if built-in successor and predecessor functions are given.

Proposition 4.5. *$D_1 \in$ DynPROP($\mathcal{SUCC}$, Rel)*

Proof. The idea of the proof uses the well known *level-trick* for the Dyck-languages (cf.[18]). All string positions of the same level are stored in a list, represented by the edge relation of a directed graph forming a cycle. This representations allows to infer whether there exists a string position of a given level and is maintainable in DynPROP($\mathcal{SUCC}$, Rel) ∎

Thus, whereas built-in relations did not increase the power of DynPROP, already the three simple functions SUCC, PRE and MIN allow the maintenance of non-regular languages.

5. Variations

Alternative Semantics. Following [18], we have introduced in Section 2 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_n^a 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, $\Delta_n = \{\text{ins}_\sigma \mid \sigma \in \Sigma\}$. Then, Dyn-alt$(L) = \{(n, \delta) \mid \delta \in \Delta_n^+ \wedge \text{word}(\delta(E_n^a)) \in L\}$[4].

Proposition 5.1 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) ∎

Notice that, contrary to Theorem 3.5, this proposition does not allow us 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 $ww' \in L$ iff $waw' \in L$. Here, if a language has at least one neutral element we will assume that the initial symbol for its dynamic algorithm is one of these neutral elements.

[4]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.

Then, a straigthforward generalization of Theorem 3.5 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). ■

Regular Tree Languages. We now investigate the dynamic complexity of the regular tree languages. Thereto, we first define dynamic tree language. 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 $\text{reset}(u)$, 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. Notice that 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 exactly 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$(L) \in$ DynPROP(Fun, Rel).* ■

6. Beyond Formal Languages

The definitions given in Section 2 only concerned dynamic problems for word structures. Following [20], we now extend these definitions to arbitrary structures. Thereto, 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 $\{\text{ins}_R, \text{del}_R \mid R \in \gamma\}$. A *concrete update* is a term of the form $\text{ins}_R(i_1, \ldots, i_k)$ or $\text{del}_R(i_1, \ldots, i_k)$, where $k = \text{arity}(R)$. 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 $\delta = \text{ins}_R(i_1, \ldots, i_k)$; and setting $R(i_1, \ldots, i_k)$ to false if $\delta = \text{del}_R(i_1, \ldots, i_k)$.

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).

Dynamic programs operate on dynamic problems just as they do on dynamic languages.

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, if we add one binary relation DynPROP cannot even capture first-order logic. This is also true if we allow the program to use precomputed functions from the set $\mathcal{SUCC}$.

Thereto we will consider *alternating graphs*, coded via the binary edge relation E and two unary relations A and B that form a decomposition of the universe V into the set of *existential* and *universal* nodes. Given a node $s \in V$, the set of all *reachable* nodes Reach(s) is defined as the smallest set satisfying (1) $s \in$ Reach(s), (2) if $u \in A$ and there is an $v \in$ Reach(s) such that $(u, v) \in E$, then $u \in$ Reach(s) and (3) if $u \in B$ and for all nodes v such that $(u, v) \in E$ we have $v \in$ Reach(s), then $v \in$ Reach(s). Now we define ALT-REACH as the problem, given an alternating graph $G = (A \dot{\cup} B, E)$ and two nodes s and t, is $t \in$ Reach(s). We note that ALT-REACH is P-complete (see for example [19]).

Proposition 6.2. *Dyn(ALT-REACH) $\notin$ DynPROP($\mathcal{SUCC}$, Rel)* ∎

In fact 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$_{\text{depth}\leq d}$ be the alternating reachability problem on graphs of depth at most d. It is easily seen that ALT-REACH$_{\text{depth}\leq d}$ is expressible by a FO-formula, so we get the following

Theorem 6.3. *There exists a problem $F \in$ FO such that Dyn(F) $\notin$ DynPROP($\mathcal{SUCC}$, Rel).* ∎

On the other hand the reachability problem on acyclic deterministic directed graphs can be maintained in DynPROP (Hesse [14]) but cannot be expressed via an FO-formula (as can be easily seen by standard EF-games arguments). 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 $\exists x_1, \ldots x_k \phi(\bar{x})$, where $\phi(\bar{x})$ is a quantifier free formula.

Theorem 6.4. *For any EFO-definable problem F, Dyn(F) $\in$ DynPROP(Fun, Fun)* ∎

The proof of this theorem relies on the fact that an EFO sentence can only assert whether a tuple of elements in the structure has certain properties, i.e. has a certain *type*. Then, using precomputed addition and subtraction functions, it is possible to count the number of tuples in a structure which have a certain type, and thus decide whether an EFO sentence is satisfied in the structure.

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 have seen that $D_1 \in \text{DynPROP}(\mathcal{SUCC}, \text{Rel})$. Can it be shown that $D_2 \notin \text{DynPROP}(\mathcal{SUCC}, \text{Rel})$?
- 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

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

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

[3] H. Björklund, W. Gelade, M. Marquardt, and W. Martens. Incremental XPath evaluation. In *ICDT*, 2009.

[4] G. Dong, L. Libkin, and L. Wong. Incremental recomputation in local languages. *Inf. Comput.*, 181(2):88–98, 2003.

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

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

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

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

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

[10] G. S. Frandsen, P. B. Miltersen, and S. Skyum. Dynamic word problems. *J. ACM*, 44(2):257–271, 1997.

[11] W. Gelade, M. Marquardt, and T. Schwentick. The dynamic complexity of formal languages. Available from *arXiv:0812.1915 [cs.CC]*.

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

[13] W. Hesse. The dynamic complexity of transitive closure is in DynTC^0. *Theor. Comput. Sci.*, 3(296):473–485, 2003.

[14] W. Hesse. *Dynamic Computational Complexity.* PhD thesis, University of Massachusetts Amherst, 2003.

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

[16] P. B. Miltersen. Cell probe complexity - a survey. In *FSTTCS*, 1999.

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

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

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

[20] V. Weber and T. Schwentick. Dynamic complexity theory revisited. *Theory Comput. Syst.*, 40(4):355–377, 2007.