Complete problems for Dynamic Complexity Classes

William Hesse* and Neil Immerman*
Computer Science Department
UMass, Amherst
Amherst, MA 01003 USA
{whesse,immerman}@cs.umass.edu

January 11, 2002

Abstract

Dynamic complexity classes such as DynFO, described by Immerman and Patnaik, have problems which are complete under bounded expansion reductions. These are reductions in which a change of a single bit in the original structure changes a bounded number of bits in the resulting structure.

Once the paper is written rewrite the abstract with much more detailed statement of its content???

1 Dynamic problems

Needs a better introduction and explanation! ???

We define a *dynamic problem*, $A = \{A_n \subseteq \Sigma_n^\star \mid n = 1, 2, \ldots\}$, to be a family of regular languages. For each value of n, A_n is a regular language over Σ_n, a polynomial-size alphabet of operations. If string $w \in \Sigma_n^\star$ is in A_n, we say that the sequence of operations w is *accepted* by the dynamic problem A[1].

*Work partially supported by NSF grant CCR-9877078.

[1]To avoid complications in our definition of reductions between problems, we require as part of the definition that the empty string is rejected by all languages A_n in all dynamic problems.

An example of a dynamic problem is the dynamic graph reachability problem DynREACH. This is the dynamic problem induced by the static decision problem REACH. An instance of REACH is a directed graph on n nodes, numbered 0 through $n-1$, with two distinguished nodes s and t. This instance is accepted if there is a directed path from node s to node t. The dynamic problem DynREACH contains, for each n, a regular language DynREACH$_n$ defined over the set of operations

$$\begin{aligned}\Sigma_n &= \{\text{Insert}(i,j) \mid 0 \le i,j < n\} \cup \{\text{Delete}(i,j) \mid 0 \le i,j < n\} \\ &\quad \cup \{\text{SetS}(i) \mid 0 \le i < n\} \cup \{\text{SetT}(i) \mid 0 \le i < n\} \ .\end{aligned}$$

The operation Insert(i,j) inserts a directed edge from node i to node j in the graph, and the operation Delete(i,j) deletes such an edge. The operations SetS(i) and SetT(i) set or change the start node and the final node of the reachability query. A sequence of operations $w \in \Sigma_n^\star$ is accepted by DynREACH if REACH contains the graph created by sequentially applying the operations in w to the empty graph on n nodes, with s and t initially set to node 0. Every static decision problem on an input of n bits has a corresponding dynamic problem defined in this way. The standard operations on the input are Set(i) and Reset(i), for $0 \le i < n$, which set and clear bit i of the input. A sequence of set and reset operations is in the dynamic version $DynA$ of a static problem A iff the n-bit input resulting from that sequence of operations in in A. The resulting languages $DynA_n$ are regular because the set of operations $\bigcup_{i=0}^{n-1}\{\text{Set}_i, \text{Reset}_i\}$ generates a finite monoid under sequential composition, and the decision problem A induces a boolean function over that monoid.

There are also dynamic problems that are not defined as the dynamic equivalent of a natural static problem. We will show later, however, that every dynamic problem is the equivalent of some static problem, with a less naturally defined set of operations. The requirement that the languages A_n contained in a dynamic problem A be regular implies the existence of a static problem and set of operations which induce A as their dynamic version, cf. Theorem **??**.

2 Dynamic Algorithms

We now define a model of dynamic computation, which will divide the set of dynamic problems into dynamic complexity classes. This model is based on finite model theory, and uses logical (relational) structures and logical formulas. This

model is suitable for describing any dynamic computation that maintains an at most polynomial-size auxiliary data structure.

Definition 2.1 (Dynamic Program) We define a *dynamic program*, $M = \{M_n = (Q_n, \Sigma_n, \delta_n, s_n, F_n) \mid n = 1, 2, \ldots\}$, to be a uniform family of deterministic finite automata. The state space Q_n will be the set of all relational data structures with a fixed vocabulary, $\tau = \langle R_1^{a_1}, \ldots, R_t^{a_t} \rangle$, over the finite universe $\{0, \ldots, n-1\}$. Each state is thus a particular instance of a polynomial-size data structure.

The alphabet, Σ_n, is the set of operations of some dynamic problem of size n. Such an alphabet will always be derived from a fixed vocabulary of operator names, $\Sigma = \langle O_1^{r_1}, \ldots, O_s^{r_s} \rangle$, by plugging in values from $\{0, \ldots, n-1\}$. That is,

$$\Sigma_n \quad = \quad \{O_j(b_1, \ldots, b_{r_j}) \mid b_1, \ldots, b_{r_j} < n; 1 \leq j \leq s\} \; .$$

Recall that in the example of DynREACH above, we had the operator names Insert, Delete of arity two, and SetS, SetT, of arity one. Thus the size of Σ_n is $O(n^r)$ where $r = \max_j r_j$.

Since the states are structures, $Q \in \text{STRUC}[\tau]$, the transition function δ_n is given as a set of logical formulas $\{\varphi_{i,j} \mid 1 \leq i \leq t, 1 \leq j \leq s\}$, defining the new state in terms of the old state and the operation performed. For each relation name, $\mathbf{R}_i \in \tau$, and operator name, $O_j \in \Sigma$, the formula $\varphi_{i,j}$ expresses the new value of R_i after the transition specified by the operator O_j. For example, the new value of R_i after operation $O_j(b_1, \ldots, b_{r_j})$ is given by,

$$R_i'(x_1, \ldots, x_{a_i}) \quad \equiv \quad \varphi_{i,j}(R_1, \ldots, R_k, b_1, \ldots, b_{r_j}, x_1, \ldots, x_{a_i}) \; .$$

If all the formulas $\varphi_{i,j}$ are first order, then the new state, Q' is a first-order translation of the previous state, Q.

The start states, s_n, are given as a set of logical formulas, $\{\sigma_i \mid 1 \leq i \leq t\}$. giving the initial values of the relations R_i, $1 \leq i \leq t$. The language used to express the start states may be different than the language used to express the transition function, meaning that the initialization of our dynamic program may have a different computational complexity than the implementation of each operation.

Finally, the final states are given as a single formula $\alpha \in \mathcal{L}$, where $\mathcal{L}$ could be $\mathcal{L}(\tau)$, the first-order language of τ, or some related language. A state $Q \in Q_n$ is in F_n iff $Q \models \alpha$.

We say that the dynamic program, M, *accepts* the dynamic problem A iff $A_n = \mathcal{L}(M_n), n = 1, 2, \ldots$. □

Given as an example the dynamic algorithm for REACH(acyclic) plus a reference to [Hes01]. ???

3 Dynamic complexity

Dynamic complexity classes arise naturally from the above definition of a dynamic program as a family of DFAs. The complexity of a dynamic program is the computational complexity of its transition function δ and its final set F. Also of interest is the precomputation of the initial state. We will assume by default that the complexity of computing the initial state is no greater than the maximum if the complexity of computing δ and F. If the initial complexity is greater, e.g., polynomial time, then we will describe this as dynamic complexity such and such *with polynomial-time precomputation*.

For example, if we require that the initial state, transition function, and final set are all described by first-order formulas, we have the dynamic complexity class Dyn-FO. This class is a slight generalization of the class Dyn-FO described by Immerman and Patnaik in [PI94]. The difference is that in [PI94], only the dynamic versions of static problems were considered; here we consider any dynamic problem whatsoever. This treatment provides a more general notion of dynamic computation which helped us discover the complete problems that we describe in the sequel.

4 Reductions

Let $A = \{A_n \subseteq \Sigma_n^\star \mid n = 1, 2, \ldots\}$ and $B = \{B_n \subseteq \Gamma_n^\star \mid n = 1, 2, \ldots\}$ be dynamic problems. A simple kind of reduction from A to B maps each operation $\sigma \in \Sigma_n$ to a uniformly bounded sequence of such requests $\gamma_1 \cdots \gamma_k \in \Gamma_{p(n)}^\star$. Note that such a reduction corresponds to a uniform, bounded sequence of homomorphisms from Σ_n to $\Gamma_{p(n)}$. Such reductions correspond to the bounded reductions investigated in [MSV94] and [PI94].

Definition 4.1 (Bounded Homomorphism)

Let $p(n)$ be a polynomial, let $k \in \mathbf{N}$ be a constant, and let $h = \{h_n \mid n = 1, 2, \ldots\}$ be a sequence of homomorphisms,

$$h_n : \Sigma_n^\star \to \Gamma_{p(n)}^\star \ ,$$

such that for all n, $\sigma \in \Sigma_n$, and $w \in \Sigma_n^\star$,

- $|h_n(\sigma)| \leq k$, and,
- $w \in A \quad \Leftrightarrow \quad h_n(w) \in B$.

Then we say that h is a *bounded homomorphism* from A to B ($h : A \leq_{\text{bh}} B$).
□

We say that bounded homomorphism h is *uniform* if the map that takes n to h_n has low complexity. For example, if h is uniformly definable by a sequence of s first-order formulas — one for each operation $O_j \in \Sigma$ — then f is a bounded first-order (bfo) homomorphism. Not surprisingly, as we will see, bfo homomorphisms and similar uniform homomorphisms preserve dynamic complexity classes.

Note that homomorphisms are *memoryless* in that they do not consider previous history when deciding how to map the current operation. It is sometimes useful to consider reductions that maintain some state as they transform one set of operations to another. However, homomorphisms are simpler and we will stick with them when they suffice.

Example 4.2 (Bounded Homomorphism from MEMBER to MEMBER′) Here is an example of a bounded homomorphism. Given a regular language $A \subseteq \Gamma^\star$, define the dynamic problem MEMBER$(A) = \{\text{M}(A)_n \mid n = 1, 2, \ldots\}$ such that $w \in \text{M}(A)_n$ iff,

- The operations are from the set $\{\text{Set}(i, \gamma) \mid 0 \leq i < n,\ \gamma \in \Gamma\}$. Set$(i, \gamma)$ is interpreted as setting character i of a string to be the symbol γ.
- If we begin with the string $(\gamma_0)^n$ and perform the operations w, then the resulting string is in A.

Define the dynamic problem MEMBER′(A) as the related problem with operations Insert(i, γ) and Delete(i), which insert character γ into the string at position i, increasing the length of the string, and delete the character at position i. Again, an operation sequence w is in MEMBER′(A) iff the sequence, w, applied to the string $(\gamma_0)^n$, yields a string in A. The size parameter n restricts the problem by ignoring "Insert" operations which would make the string longer than n characters.

Given these two dynamic problems, it is easy to see that the operation Set(i, σ) from the first problem can be implemented as the sequence of operations Delete(i), Insert(i, σ) in the second problem. Thus the bounded homomorphism from MEMBER(A) to MEMBER′(A) is given as follows: $p(n) = n$; $k = 2$; and, $h_n(\text{Set}(i, \sigma) = \text{Delete}(i)\ \text{Insert}(i, \sigma)$ □

5 Building a Complete Problem for DynFO

In this section we begin the process of constructing a complete problem for Dyn-FO. Recall that we defined Dyn-FO to be the dynamic complexity class in which the initial state, transition function, and final set are all described by first-order formulas.

Let $A = \{A_n \subseteq \Sigma_n^\star \mid n = 1, 2, \ldots\}$ be an arbitrary Dyn-FO problem. Let the operator alphabet be $\Sigma = \langle O_1^{r_1}, \ldots, O_s^{r_s} \rangle$.

Since $A \in$ Dyn-FO, it has a dynamic program, $M = \{M_n = (Q_n, \Sigma_n, \delta_n, s_n, F_n) \mid n = 1, 2, \ldots\}$, in which δ, s, and F are all first-order definable.

Recall that $Q_n \subseteq \text{STRUC}[(]\tau)$ consists of all structures with universe $\{0, 1, \ldots, n-1\}$ for a fixed vocabulary, $\tau = \langle R_1^{a_1}, \ldots, R_t^{a_t} \rangle$.

The fact that A is in Dyn-FO means that we are given first-order formulas as follows:

- σ_i, $1 \leq i \leq t$, defining the t initial relations;
- $\varphi_{i,j}$, $1 \leq i \leq t, 1 \leq j \leq s$, defining the new values of each of the t relations in response to each operator, O_j; and
- α, the acceptance condition.

We next observe that any Dyn-FO program can be easily transformed to one in which the initial relations and the accept relation are trivial:

Observation 5.1 *Let M be* Dyn-FO *dynamic program. Then there is an equivalent* Dyn-FO *program M' such that all the initial relations are empty, i.e, $sigma_i =$* **false**, *$i = 1, \ldots, t$, and the acceptance condition is also trivial, $\alpha = F$ where F is a new relation of arity 0, i.e., a boolean constant.*

Proof: Let $\tau' = \tau \cup \{S^0, F^0\}$, i.e., we augment our data structure with two bits: S (start) and F (final). In the initial states s'_n, all relations are empty, i.e., false.

The transition relations $\alpha_{i,j}$ are modified as follows, on input an operation $O_j(b_1, \ldots, b_{r_j})$, do the following,

1. If $\neg S$, then start by running the initialization definitions, $R_i := \sigma_i$ and $S' \equiv$ **true**.
2. Apply the transitions as in M, $R'_i := \varphi_{i,j}(b_1, \ldots, b_{r_j})$; and finally,

3. Record whether we are in a final state: $F' := \alpha$

Thus M' is exactly equivalent to M and performs the same work that M does, except that it explicitly records in the bit F whether or not we are currently in a final state. □

5.1 Single Step Circuit Value

We now describe the dynamic problem, single step circuit value (SSCV). We will show in Section **??** that a variant of this problem is complete for Dyn-FO.

SSCV is a not-necessarily-acyclic dynamic circuit value problem. Initially, the problem of size n consists of n "input"-gates with current value "0" and no wires connecting them. SSCV allows the following operations:

- Insert(i, j): add a wire from gate i to gate j;
- Delete(i, j): delete any wire from gate i to gate j;
- And(i), Or(i), Not(i), Input(i): make gate i an "and", "or", "not", "input" gate, respectively;
- Set(i), Reset(i): set the current value of gate i to "1", "0", respectively;
- Step: synchronously propagate values one step through the whole circuit. (An input gate retains its current value, a "not" gate with more than one input is treated as a "nand" gate.)

The acceptance condition for SSCV is that gate 0 has value "1".
It is easy to see that SSCV is in Dyn-FO,

Proposition 5.2 *SSCV is in* Dyn-FO.

Proof: We maintain the current value of the circuit as a colored graph, i.e., a logical structure with one binary edge relation and three unary relations indicating whether each gate is "and", "or", "not", or "input", and its current binary value.

The operation "Step" is easy to define in a first-order way. All the other operations simply set one value of the above relations. □

==========
Here is where I am now!!!??? ==========

6 A Complete Problem in DynFO: Complete SSCV (CSSCV)

We can create a variant of SSCV that is complete for DynFO under bounded reductions by modifying the initial state. The initial state of the SSCV variant will be an arrangement of gates that can emulate any FO update computation. This initial circuit will include, as one of its components, an FO-uniform AC^0 circuit that is equivalent to a universal first-order formula. The universal FO formula can emulate any FO formula, as described below. The initial circuit will also contain (cyclic) circuits that will store the auxiliary data of a DynFO algorithm, and circuits to present this auxiliary data as input to the universal formula, and copy the result back to the auxiliary data storage.

The complete initial circuit will allow us to emulate any FO update computation by making a bounded number of modifications to the circuit and propagating values through the circuit a bounded number of steps. This allows any FO update to be emulated by a bounded finite sequence of SSCV operations. Therefore, any DynFO algorithm can be emulated by our SSCV variant via a bounded homomorphism. The construction of the universal formula and the rest of the initial circuit are described in the remainder of this section.

6.1 A universal first-order formula

To form a complete problem for DynFO, we start with something like a complete problem for FO, which is equal to the circuit class uniform AC^0. It is known that no complete problem for FO exists, (true?)(cite?) since FO is stratified by quantifier alternation depth, in a way which cannot be obviated by the polynomial expansion allowed by reductions (we must use quantifier-free projections, as our reductions, since any reductions which can compute FO functions make any non-trivial problem complete for FO). Since in our dynamic setting, we are allowed to iterate an FO update formula a constant number of times, depending on the problem we are reducing from, we can overcome this stratification.

We show there is a universal first-order formula $\mathcal{U}$ which acts as an interpreter for FO logic. Given as input a first-order formula f, and the structure S it is to be evaluated over, the universal formula computes the truth value of the formula: $S \models f$. If f has free variables, then the universal formula computes the relation defined by f. The relation defined by f is the map from tuples of elements of the universe of S to truth values that is computed by substituting the elements of the

tuple for the free variables of f when evaluating f.

The universal formula is implemented as a map from strings to strings, which may need to be iterated. $\mathcal{U}$ has one free variable and its vocabulary has one unary relation symbol. Its input string, considered as a unary relation, R, is used as the interpretation of the unary relation symbol. The truth value of $\mathcal{U}$, for each value of its free variable, gives us the output string as a unary relation. Thus an input string is mapped to an output string of the same length by the formula $\mathcal{U}$.

The universal formula emulates any FO formula by iterating this map. The number of iterations depends only on the size of the emulated formula.

Lemma 6.1 *There is a first-order formula $\mathcal{U}$ over the vocabulary containing the fixed numeric predicates $<$ and BIT, and the unary predicate R, so that:*

For any FO formula f over vocabulary V, where V contains only relation symbols, no functions or constants:

There is an encoding w_f of f as a binary string, a number d, the depth of formula f, and a padding function $p(n)$ that is a polynomial in n, so that:

For any structure S over the vocabulary V, encoded as a binary string w_S that begins with the specification of the universe size, n, as 0^n1:

The result of applying $\mathcal{U}$ to the string $w_f\ w_S\ 0^{p(n)}$ d times is a string containing the relation f^S:

$$\mathcal{U}^{(d)}(w_f\ w_S\ 0^{p(n)}) = w_f\ w_S\ w_{\text{final}},$$

where w_{final} begins with the binary encoding of the relation

$$f^S = \{(a_1, \ldots, a_r) \mid (S, [a_1/x_1] \ldots [a_r/x_r]) \models f\},$$

where $x_1, \ldots, x_r$ are the free variables of f.

Proof: This formula is constructed in Appendix **??**. □

Because any function described by an FO formula can be computed by an FO-uniform AC^0 circuit [cite?], we can use the circuit corresponding to the universal formula $\mathcal{U}$ as part of the initial circuit of the SSCV variant.

6.2 The initial circuit

The inital state of our complete SSCV variant is a (cyclic) circuit containing the following components:

- A *latch* containing gate 0 of the SSCV problem. The value of this latch will determine whether the current state of the SSCV problem is accepting or not. A latch is a gadget made from gates that can remember a circuit value: a memory cell.

- Two gates with values 0 and 1. These can be constructed as an OR gate with no inputs and an AND gate with no inputs. These are used wherever we need a constant input in our other components.

- An array of n latches, denoted $A[]$. Individual elements of the array are denoted by $A[0], \ldots, A[n-1]$. These will store the auxiliary data of the dynamic computation, and additional intermediate data needed by our emulation of the FO updates.

- The FO-uniform AC^0 circuit computing the universal FO formula $\mathcal{U}$. The input to this circuit will be the array $A[]$, and the output, an array of bits of the same length, will be the inputs to array $A[]$, so that they can be written into $A[]$ by triggering the control signal of the latches in $A[]$.

- A depth two AC^0 circuit which can shift the contents of array $A[]$ to the left or right any number of positions. The amount of shift is controlled by setting one of $2n-1$ control signals to 1, and letting the others remain 0. The output of this shift circuit is fed back into the latches $A[]$, controlled by a second latch control signal.

The construction of latches can be accomplished by any of the familiar constructions of a memory cell or flip-flop from basic logical gates. We show an example of such a latch in appendix B. Any latch may be set to a constant value, by temporarily redirecting its input to a constant 1 or 0, and temporarily redirecting its control line to 1.

The size of the final circuit is bounded by a polynomial in n, the number of latches in the memory array $A[]$. This size is at least quadratic in n, if we use a simple circuit implementing an arbitrary shift of the array $A[]$. Let this polynomial be $q(n)$.

We now define our complete variant of SSCV, called CSSCV. CSSCV is the dynamic problem accepted by the following dynamic computation:

- The state space and transition function of CSSCV are the same as for SSCV. We have relations indicating the structure of a circuit with n gates, and the current values of those gates.

- The final set is the same as for SSCV. Those states of the circuit in which gate 0 has value 1 are accepting states.
- The start state of CSSCV is the initial circuit described above. If the value of the size parameter m of the CSSCV problem is equal to $q(n)$ for some n, then the initial state of the dynamic computation is the circuit above with a memory array of size n. If m is not of this form, then the inital state of the computation is the same as for SSCV.

6.3 emulating a DynFO update

To reduce any problem A in DynFO to CSSCV, we first simplify the DynFO problem as stated in section ???, eliminating the complexity of the initialization and the complexity of querying the auxiliary data structure. We also change the auxiliary data structure to be a unary data structure over a polynomially expanded universe. This simplified problem, A', is then characterized by a set of FO formulas, one for each operation type. These formulas give the new value of the unary relation given the previous value of that relation and the parameters of the operation.

The polynomial expansion required by our reduction from problem is a combination of the expansion required by the simplification from A to A', the maximum expansion required by our universal FO formula $\mathcal{U}$ when simulating any of the update formulas φ on the auxiliary data, and the expansion required by our circuit, which has a size polynomial in the size of its memory register $A[]$. [Make sure the polynomial expansion required by $\mathcal{U}$ on emulating a fixed formula φ has been discussed].

To emulate an update operation $O(i_1, \ldots, i_k)$, we evaluate the formula φ_O corresponding to that operation upon the auxiliary data kept in $A[]$. We will be copying the formula φ_O, properly encoded, into the beginning of the register $A[]$. Therefore, the first step is to copy the auxiliary data to the correct position in the middle of $A[]$. The correct position can be computed in FO from the size of the universe of A', n. (The formula φ_O is fixed). The first operation is then to set the correct control line of the shift circuit to 1, by connecting it to the constant gate with value 1. We then step the circuit a constant number of times to effect the shift, and latch the shifted values into the register $A[]$ by triggering the latch control line. We then perform the constant number of modifications necessary to put the encoded version of φ_O into the beginning of $A[]$, and set the correct bit in $A[]$ so that 0^n1, the encoding of n in unary, appears in the correct place. [Make sure that the encoding is 0^n1 in lemma and appendix].

At this point, the memory register $A[]$ contains the binary string $w_\varphi 0^n 1 w_R 0^r$, where w_R is the unary relation that is the auxiliary data for the dynamic computation A' and 0^r is some string of 0s longer than the necessary padding in the definition of the universal formula $\mathcal{U}$. Thus, the memory register is in the correct format for an input to $\mathcal{U}$. Therefore, we propogate the circuit values forward a sufficient number of times to compute $\mathcal{U}$. The depth of the circuit computing $\mathcal{U}$ is a constant independent of the particular formula φ_O that $\mathcal{U}$ is emulating. After we have propagated the circuit values to at least that depth, the circuit computing $\mathcal{U}$ has stabilized, and we toggle the control signal that latches the result into the memory register $A[]$. This is just one iteration of $\mathcal{U}$; we apply $\mathcal{U}$ a number of times depending on the depth of φ_O in order to compute φ_O on the input data w_R. Thus we repeat the process of stepping the circuit computing $\mathcal{U}$ until it stabilizes, then copying the result back to $A[]$, a constant number of times depending on the formula φ_O.

At this point, the memory register $A[]$ contains the predicate computed by φ_O in some portion of the string w_{result}. This predicate is not a unary predicate containing the new value of the unary relation R though. Since φ_O has free variables which must be replaced by the parameters to the operation O, the predicate computed by φ_O has arguments corresponding to these parameters, and so contains all the new values for R for all possible parameters to the operation O. If the free variable of φ not corresponding to an input parameter, but to the bit position in the resulting bit string (unary relation), is the last free variable of φ in its encoding, then the bit strings for different values of the parameters are simply concatenated to make the encoding of the higher-arity predicate computed by φ_O. Therefore, to copy the correct new value for R back to the beginning of $A[]$, we must apply a shift operator to A which depends on the input parameters to operation O. The size of the shift is FO computable from the parameters and the formula φ_O, so we can set the appropriate control signal and perform the shift by applying a constant number of operations to CVSSCV, with parameters that can be computed in FO from the parameters to O.

Finally, we copy the appropriate bit of $A[]$ to the latch which contains gate 0 of the circuit, so that the problem CVSSCV accepts if and only if that bit of $A[]$ is set, which is true if and only if the DynFO computation being simulated accepts at this point.

References

[Hes01] William Hesse, "The Dynamic Complexity of Transitive Closure is in TC^0, *Intl. Conf. on Database Theory*(2001), 234 - 247.

[MSV94] Peter Bro Miltersen, Sairam Subramanian, Jeffrey Scott Vitter, and Roberto Tamassia, "Complexity Models for Incremental Computation," *Theoret. Comp. Sci.* (130:1) (1994), 203-236.

[PI94] S. Patnaik and N. Immerman, "Dyn-FO: A Parallel, Dynamic Complexity Class," *J. Comput. Sys. Sci.* 55(2) (1997), 199-209. A preliminary version appeared in *ACM Symp. Principles Database Systems* (1994), 210-221.

A Universal formula $\mathcal{U}$ for FO

First, give basic idea.

The universal formula $\mathcal{U}$ will operate on an input string that encodes a first-order formula φ and the data that formula operates on. The first part of the string is the encoding of the first-order formula φ. This encoding contains a data structure for each subformula of φ, and thus encodes the abstract syntax of the formula φ. The second part of the string contains blocks of data representing the predicates computed by the subformulas of φ. Any subformula represents a predicate over its free variables, and the truth tables of these subformulas are stored as Boolean arrays in the data blocks of the string. . Using the inductive definition of truth, the predicate computed by a formula depends in one of a few simple ways on the predicates computed by its maximal strict subformulas. Our universal formula $\mathcal{U}$ will be able to compute a single step of this inductive definition of truth. By applying $\mathcal{U}$ to the string repeatedly, we can compute the predicate corresponding to the formula φ in a number of rounds equal to the depth of the parse tree of φ. The details of the encoding and the computation are presented below.

We define the encoding of a formula φ based on the inductive construction of φ. We allow formulas to be constructed by the following rules, in which α and β are formulas, R is a relation from the vocabulary of the formula, and $x, x_1, x_2, \ldots, x_k$ are variables.

The following are formulas, for any formulas α and $beta$, relation R, and variables $x, x_1, x_2, \ldots, x_k$:

- $R(x_1, x_2, \ldots, x_k)$, where k is the arity of relation R
- $\neg\alpha$
- $\alpha \wedge \beta$
- $\alpha \vee \beta$
- $(\forall x)\alpha$
- $(\exists x)\alpha$

Thus, for any FO formula φ we have an abstract syntax tree for that formula, with tree nodes corresponding to instances of these rules. All leaves are instances of the first rule, defining atomic terms, and all internal nodes are instances of the other rules. All subformulas of φ correspond to nodes in the abstract syntax tree, and the maximal strict subformulas of a subformula are its immediate children in the tree. We now define an encoding of this tree.

We first define two constants based on the formula. Let k be the number of nodes in the abstract syntax tree of φ. Let r be the number of distinct variables used in the formula φ. By renaming bound variables, this number can be reduced to the maximum number of free variables in any subformula of φ. We assume this number is greater than the maximum arity of any relation R in the formula's vocabulary.

We also encode the vocabulary of the formula: a finite list of relations $R_1, \ldots, R_{rel}$ that includes all relations referenced by the formula. This list may include relations not actually used by the formula. These will be part of the data structure the formula is evaluated over, but the formula's value will not actually depend on these relations. We shall assume that the base arithmetic relations used by our formula are included in this list, so that R_1, R_2, and R_3 will be the binary relations $=$, $<$, and BIT. The values of these relations will be computed internally in $\mathcal{U}$, and need not be given in the encoding of the input data structure.

These basic structural constants will be represented in unary at the beginning of our encoding of φ. Thus, the representation of φ as a binary string will start with $1^k01^r01^{rel}0$. This will be followed by the arities of the relations $R_1, \ldots, R_{rel}$, encoded as rel positive integers of $\lceil \log r \rceil$ bits each. Since the number of relations may be linear in the size of the input to $\mathcal{U}$, in a pathological case, we will require that the relations be sorted in order of increasing arity (after the first three). Otherwise, we could not decode the input structure without sorting a large number of items, which cannot be done by an FO formula1a.

Next, we encode the k nodes of the abstract syntax tree into a list of k fixed-size data structures. These need to record the type of formula constructor used at this node, and the arguments of this constructor. The constructor of an atomic term needs to record the name of the relation used and the names of the variables that are the arguments to the relation. This requires $\log rel + r \log r$ bits, since the arity of relations is bounded by r. Name of relations, and pointers in general, are represented as integers with the required number of bits. A tree node representing a formula as the AND of two subformulas needs to record pointers to the two subformulas; this requires $2 \log k$ bits. Thus the size of a data structure representing an abstract syntax tree node, and thus representing a subformula, is $s = 3 + \max(\log rel + r \log r, 2 \log k)$ bits.

So the formula φ is represented by ks bits attached to the above prefix, representing the k nodes in the abstract syntax tree of φ. Node 0, the first node, is the root of the tree, representing the entire formula φ.

The entire representation w_φ of φ has length $3 + k + r + rel + rel \log r + ks$, which is of course a constant for any fixed formula φ. Note this encoding can handle formulas with any number of variables, relations of any arity, and any quantifier depth.

We next discuss the encoding w_S of the input data structure S, which interprets the rel relations $R_1, \ldots, R_{rel}$ as relations of the appropriate arity over the finite universe $\{0, 1, \ldots, n-1\}$. We assume that $n \geq 2$, so that the universe has at least two elements. The size of the finite universe, n, is independent of the formula φ, but the number and arities of the relations R_i must be that given by the encoding of the formula. We simply encode the relations as boolean arrays, and encode them as linear strings by writing them in row-major order. The representation of the entire structure is then the unary representation of n as 1^n0 followed by the encodings of the relations $R_4, \ldots, R_rel$. Remember that the first three relations are the given numeric relations $=$, $<$, and BIT, which can be computed by $\mathcal{U}$. This is then an encoding of the structure with length depending on n and the number and arity of the ! relations R_i. The computation of this total length, and of the starting point of each relation in the string, are computable in first order from the encoding w_φ and n, since the relations are sorted by arity. The maximum arity of an input relation is also bounded by the logarithm of the total length of the input, since $n \geq 2$ and the input contains w_S. Therefore, computing the size of w_S can be done using a polynomial in n with degree (and number of terms) bounded by the log of the input size.

The final part of the input is the padding, where the relations computed by subformulas of φ are computed and stored. We will store a relation of arity r over

the universe $\{0, \ldots, n\}$ for each subformula, thus requiring kn^r bits of padding.

This appendix shows a latch

gadgets serving as persistent memory, called latches. We use the traditional method of implementing memory as bistable logical networks. We can construct a cyclic circuit with two external inputs, set and $\overline{reset}$. The circuit contains an and gate and an OR gate, forming a cycle of length two. If gate 1 is an AND gate, and gate 2 is an OR gate, the inputs to gate 1 are the output of gate 2 and the external input $\overline{reset}$, and the inputs to gate 2 are the output of gate 1 and the external input set. [Show picture]. If the signals set and $reset$ are 0, then the circuit has two stable states and one oscillating state. ! The values of the gates may both be 0, both be 1, or they may have different values. If the gates have different values, the values of the gates oscillate between 0 and 1 with every time step. The circuit can be forced into state 0 by holding $\overline{reset}$ to 0 for two steps, and can be forced into state 1 by holding set to 1 for two steps.