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 7, 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

We define a dynamic problem, A, as a family of regular languages, parameterized by a size parameter n. For each value of n, the dynamic problem contains a regular language over a fixed alphabet of operations, of size $n^{O(1)}$ (size bounded by a polynomial in n). We denote this alphabet by Σ_n, and the regular language over it by A_n. If a 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. To avoid complications in our definition of reductions between problems, we state as part of the definition that the empty string is accepted 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

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

The operation $\text{Insert}(i,j)$ inserts a directed edge from node i to node j in the graph, and the operation $\text{Delete}(i,j)$ deletes such an edge. The operations $\text{SetS}(i)$ and $\text{SetT}(i)$ 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 $\text{Set}(i)$ and $\text{Reset}(i)$, 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.

*Work partially supported by NSF grant CCR-9877078.

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 best suited to describing dynamic computations which maintain a data structure of a fixed size, polynomial in the size parameter n.

We define a dynamic computation to be a uniform family of deterministic finite automatata (DFAs). For each value of the size parameter n, we will construct a DFA $M_n = (Q_n, \Sigma_n, \delta_n, s_n, F_n)$. Each component of the DFA will be described by a finite computational structure, specifying how to construct it from the value n.

The state space Q_n will be the set of all relational data structures with a given vocabulary over the finite universe $\{0, \ldots, n-1\}$. Each state is thus a particular instance of a polynomial size data structure. The finite description of this component of the dynamic computation is a finite set of relation names, each with an associated arity.

Σ_n, the alphabet of the DFA M_n, is the set of operations of a dynamic problem of size n. It is given by a finite set of operator names, each with a given arity. An operation is encoded as an operator name and a tuple of parameters, drawn from the universe $\{0, \ldots, n-1\}$. In the example of DynREACH above, we had the operator names Insert and Delete, both of arity two. This encoding gives us a polynomial bound on the number of operations.

The transition function δ_n is given as a set of logical formulas. For each relation name and operator name, we have a formula expressing the contents of that relation after the transition specified by the operator. In this paper, we work with first-order logic and first-order logic with generalized quantifiers, in which our formulas contain free and bound variables ranging over our finite universe.

Specifically, if the relation names are $R_1, \ldots, R_k$, with arities $u_1, \ldots, u_k$, and O_j is an operator name with arity u, then our transition function $\delta_n((R_1, \ldots, R_k), O(p_1, \ldots, p_u))$ yields a state in which the relations have new values given by formulas:

$$R_i := \varphi_{i,j}(R_1, \ldots, R_k, p_1, \ldots, p_u, x_1, \ldots, x_{u_i}) \ .$$

The previous values of the relations $R_1, \ldots, R_k$ and the parameters $p_1, \ldots, p_u$ of the operation are substituted (??) in the formula, and the result is a formula with u_i free variables $x_1, \ldots, x_{u_i}$. This resulting formula expresses a relation over these free variables, and this relation is the new value of relation R_i in the new state of the system. Thus the new value of relation R_i is a function of the previous values of relations $R_1, \ldots, R_k$ and of the parameters $p_1, \ldots, p_u$ of the operation, and this function is expressed by the logical formula $\varphi_{i,j}$. If these are first-order formulas, the new relational data structure is a first-order interpretation of the previous relational data structure augmented with constants representing the operation's parameters.

The start state of the DFA, s_n can be given as a set of logical formulas, one for each relation R_i, with u_i free variables, giving the initial value of that relation. The logical language used to express the start state may be different than the language used to express the transistion function, meaning that the initialization of our dynamic computation may have a different computational complexity than the implementation of each operation. The final set of the DFA can similarly be given as a formula using the relations $R_1, \ldots, R_k$, which evaluates to true or false depending on whether a state, considered as a relational data structure, is accepted by the DFA or not.

A dynamic computation described this way decides the dynamic problem A iff $A_n = \mathcal{L}(M_n)$. Thus the dynamic problem decided by a family of DFAs is just the family of regular languages decided by the DFAs.

3 Dynamic complexity

Dynamic complexity classes arise naturally from the above definition of a dynamic computation as a family of DFAs. The dynamic complexity of a dynamic computation is the computational complexity of its transition function δ and its final set F. These are the only computations which are carried out at every step of a dynamic computation. The other objects, such as the state space, initial state, alphabet, and final step, are either just specified as encodings, or can be considered initialization of the algorithm. The transition function δ is just the infinite family of functions $\delta_1, \delta_2, \ldots$, all described by the same formula, described in the previous section. We must talk about the computational complexity of this infinite family, of course, since all the individual functions δ_i are finite objects, and thus have trivial static complexity.

For example, if we require that the transition function and final set be described by first-order formulas, we have the dynamic complexity class DynFO. This class is roughly equivalent to the class DynFO described by Immerman and

Patnaik in [**?**]. This can be briefly summarized as the class of dynamic problems accepted by dynamic computations in which a polynomially sized data structure is updated and queried using first-order formulas. If we allow majority quantifiers in our formulas describing the transition function, we get the class DynFOM, roughly equivalent to the class Dynamic TC^0 described in Hesse [**?**]. Finally, if our transition function is described by quantifier-free formulas, then we have a number of previously undescribed classes contained in DynFO. These classes may also be described by asserting that the transition function has a certin circuit complexity, due to the equivalence of many descriptive and circuit complexity classes. We will use the notation Dyn$\mathcal{C}$ to denote the class of dynamic problems decided by dynamic computations with transition functions computable in the static complexity class $\mathcal{C}$

Our definition of dynamic complexity classes yields complexity classes that are identical to or similar to the classes described in work on dynamic complexity by Immerman and Patnaik [**?**], Libkin and Wong [**?**], and Wigderson et. al. [**?**]. The easiest way to show that a dynamic problem is in a complexity class is by giving an algorithm in our computational model.

Beyond these basics, our model of dynamic complexity achieves results similar to the results known for static complexity classes: the existence of reductions from one dynamic problem to another, preserving membership in dynamic complexity classes, and the existence of problems complete for dynamic complexity classes under these reductions.

4 Reductions

We define a bounded reduction from one dynamic problem to another as a bounded homomorphism from the languages in one problem to the languages in the other. We say that a problem A reduces to a problem B via bounded reduction f iff:

- $p(n)$ is an easily computable, polynomially bounded function, giving the polynomial expansion in problem size used by the reduction.
- f is a family of homomorphisms $f_1, f_2, \ldots$ such that $f_i : \Sigma_i^\star \to \Delta_{p(i)}^\star$ is a homomorphism from the free monoid over the alphabet Σ_i of language A_i to the alphabet $\Delta_{p(i)}$ of the language $B_{P(i)}$, such that $f_i(w) \in B_{p(i)} \iff w \in A_i$.
- There is an integer bound m on the family of homomorphisms such that for any f_i and any $w \in \Sigma_i^\star$, $|f_i(w)| \leq m|w|$.

As long as the homomorphisms have a low computational complexity, we find that such a reduction preserves dynamic complexity. We will show that for any static complexity class $\mathcal{C}$ closed under composition, the dynamic complexity class Dyn$\mathcal{C}$ is preserved by bounded reductions in the same way that static complexity classes are preserved under reductions.

Because a homomorphism always takes the empty string to the empty string, any reduction as defined above can only take a dynamic problem to another dynamic problem which agrees on the acceptance of the empty sting for all values of the size parameter (or the polynomially changed size parameter). To avoid this difficulty, which would trivially preclude the existence of complete problems under this type of reduction, we do not pay attention to the acceptance of empty strings. [1]

Here is an example of a reduction as defined above. Given a regular language $A \subseteq \Sigma^\star$, define the dynamic problem MEMBER_A to be those sequences of operations on a string of length n such that:

- The operations are from the set $\left\{\text{Set}(i, a) \;\middle|\; 0 \leq i < n,\ a \in \Sigma\right\}$. $\text{Set}(a, i)$ is interpreted as setting character i of a string to be the symbol a.
- When the sequence of operations is applied to the string a_0^n containing n copies of the first symbol in the alphabet, the resulting string is in A.

[1] An alternative solution would be to extend the mapping from $\Sigma_i^\star$ to $\Delta_{p(i)}^\star$ by adding a prefix string from $\Delta_{p(i)}^\star$, depending on i, to each homomorphic image. This gives us a larger set of reductions.

Define the dynamic problem MEMBER$'_A$ as the related problem with operations Insert(i, a) and Delete(i), which insert character a into the string at position i, increasing the length of the string, and delete the character at position i. Again, an operation sequence in in MEMBER$'_A$ iff that sequence, applied to the string a_0^n, yields a string in A. The size parameter n restricts the problem by disallowing 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, a) from the first problem can be implemented as the sequence of operations Delete(i), Insert(i, a) in the second problem. The reduction from problem MEMBER$_A$ to MEMBER$'_A$ is given as follows:

- $p(n) = n$: The reduction takes an instance of size n of MEMBER$_A$ to an instance of size n of MEMBER$'_A$.
- The family of homomorphisms $f_1, f_2, \ldots$ takes operation Set(i, a) to the sequence of operations Delete(i) Insert(i, a).
- There is an integer bound $m = 2$ on the family of homomorphisms such that for any f_i and any $w \in \Sigma_i^\star$, $|f_i(w)| \leq m|w|$.

5 The class DynFO

We pin down our definition of the class DynFO, using the model described in Section **??**. A dynamic problem is in DynFO if there is a dynamic algorithm in which the start state, transition function, and set of accepting states are given by first-order formulas. Thus, we must have:

- A set of relation symbols, $R_1, \ldots, R_k$, and their arities $r_1, \ldots, r_k$.
- An ordered list of variable symbols, $x_1, \ldots, x_c$.
- A list of FO formulas $\mathrm{start}_1(x_1, \ldots, x_{r_1}), \ldots, \mathrm{start}_k(x_1, \ldots, x_{r_k})$. These give the starting values of the relations $R_1, \ldots, R_k$. Each formula start_i has exactly r_i free variables, which are the first r_i variable symbols. It may also use bound variables from the list of variable symbols, and the fixed binary relations $<$, $=$, and BIT [2]
- An ordered list of operation symbols, $O_1, \ldots, O_m$, and their arities $o_1, \ldots, o_m$.
- A list of FO formulas defining the transition function: $\varphi_{1,1}, \ldots, \varphi_{1,k}, \varphi_{2,1}, \ldots, \varphi_{m,k}$. The formula $\varphi_{i,j}$ defines the new value for relation R_j after performing operation O_i. The formula $\varphi_{i,j}$ has $r_j + o_i$ free variables, the variables $x_1, \ldots, x_{r_j}, x_{r_j+1}, \ldots, x_{r_j+o_i}$. The variables $x_{r_j+1}, \ldots, x_{r_j+o_i}$ are instantiated with the o_i parameters to operation O_i, resulting in a formula with r_j free variables that gives the new value for relation R_j. This formula may use variables from the list of variable symbols, the relation symbols $R_1, \ldots, R_k$, used with the correct arity, and, as stated above, the fixed binary relations $<$, $=$, and BIT. These relation symbols are interpreted as the current values of the relations $R_1, \ldots, R_j$.
- An FO formula, π, defining the set of accepting states. This has no free variables, and is over the vocabulary $\{<, =, \mathrm{BIT}, R_1, \ldots, R_k\}$. Its bound variables are from the set $x_1, \ldots, x_c$.

We can further formalize the notion that these formulas define a transition function for a DFA. The state of the DFA is a structure over a vocabulary $\{R_1, \ldots, R_k\}$. We combine this structure, its universe $\{0, \ldots, n-1\}$, and interpretations of the variables $x_1, \ldots, x_k$ to get a model $\mathcal{A}$. And so on.

5.1 Subsuming initialization and query complexity

We now show that any dynamic problem in DynFO has a dynamic computation of a certain form deciding it. For every dynamic computation, there is an equavalent computation which has trivial formulas describing its start state and its set of final states. We can extend the set of relations used by a DynFO problem by adding two relations S and F of arity 0 (Boolean constants), so that the formulas defining the start state are all uniformly false, and the formula defining the final state is just the atomic formula F.

[2] define BIT here, and say $FO = FO(=, <, BIT)$.

We do this by composing the formula π which decides acceptance with the formulas computing the update, and using this boolean result to update the relation F. Thus, after each update, F is true iff the dynamic computation accepts. To initialize the relations, we prepend each update with an update which checks S, and if S is false, sets all relations to their initial values and updates S to true. This is combined with the update and query formulas to yield a new set of first-order update formulas, which includes the functionality previously performed by the initialization and query formulas. The initialization formulas only run once, of course, on the first update of the problem. The new dynamic computation initializes all relations to false.

Equal to previous work?

Reduction of all problems to unary relations (with polynomial blowup, encoding higher arity into strings)?

6 Single step circuit value

We now describe a dynamic problem, single step circuit value (SSCV), with a variant that is complete for DynFO. This will be a dynamic circuit value problem, in which the circuit is a network of gates, not necessarily acyclic. The values of the gates are updated synchronously, and may not stabilize to a fixed set of values. At each time step, each gate computes its value as the appropriate value of its inputs, and that value is the gate's output at the next time step.

The dynamic version of this problem lets us change the type of a gate, and the connections between gates. The number of gates is fixed at n, and the values of the gates are initially all false. The operations we allow are Connect(i, j), which connects the output of gate i to an input of gate j, leaving all other connections from i and to j unchanged; Disconnect(i, j), which removes this connection; And(i), which makes gate i an AND gate; Or(i); Not(i); and Step(), which propagates values one step through the circuit. The exact semantics of the problem, including the acceptance criterion, and the initial configuration of the circuit, will be given in the presentation of the dynamic computation deciding this problem.

SSCV is in DynFO, and we will show that a variant of it is complete for DynFO. SSCV is in DynFO because it is accepted by the dynamic computation described by:

- The relation symbols E^2, A^1, O^1, N^1, and V^1. (The arities of relations are given as superscripts). The binary relation E^2 (Can we conflate relations and their symbols here?) maintains the connections between gates, and $E(x, y)$ is true iff an output of gate x is connected to an input of gate y. The unary relations $A(x), O(x)$, and $N(x)$ are true if gate x is an AND, OR, or NOT gate, respectively. The relation $V(x)$ stores the current value of gate x.
- Our list of variables will be $x, y, u, v, w, z, z1, z2, \ldots$
- The starting values of our relations are simple: they are all initialized to false. Note that gates thus have no initial type until their type is set. Our update formula implies that the value of a gate with no type is always false. The complete variant of the SSCV problem initializes the the relations to create a uniform AC^0 circuit which computes the result of a universal FO formula. The values $V_0(x)$ may be assumed to be false, as the correct ones will be computed by the circuit.
- The operations as described above: Connect(i, j), Disconnect(i, j), And(i), Or(i), Not(i), and Step().
- The update computations are:
  - For Connect(u, v), the only formula which modifies a relation is

$$E(x, y) := E(x, y) \vee x = u \wedge y = v \ .$$

    To show this one case in full, we show all formulas $\varphi_{0,0}, \ldots, \varphi_{0,4}$.

$$\begin{aligned} \varphi_{0,0}(x, y, u, v) &= E(x, y) \vee x = u \wedge y = v \\ \varphi_{0,1}(x, y, u) &= A(x) \\ \varphi_{0,2}(x, y, u) &= O(x) \\ \varphi_{0,3}(x, y, u) &= N(x) \\ \varphi_{0,4}(x, y, u) &= V(x) \end{aligned}$$

– For Disconnect(i, j), the only nontrivial formula is

$$E(x, y) := E(x, y) \wedge \neg(x = i \wedge y = j) \ .$$

– For And(i), the non-trival formulas are

$$\begin{aligned} \varphi_{2,1}(x, y) &= A(x) \vee (x = y) \\ \varphi_{2,2}(x, y, u) &= O(x) \wedge \neg(x = y) \\ \varphi_{2,3}(x, y, u) &= N(x) \wedge \neg(x = y) \end{aligned}$$

– Or(i) and Not(i) are handled similarly.

– The operation Step() leaves relations $E(i, j)$, $A(i)$, $O(i)$, and $N(i)$ unchanged, and updates $V(i)$ according to the formula

$$\begin{aligned} \varphi_{5,4}(x) = \quad & A(x) \wedge (\forall y)(E(y, x) \rightarrow V(y)) \\ \vee \quad & O(x) \wedge (\exists y)(E(y, x) \wedge V(y)) \\ \vee \quad & N(x) \wedge (\exists y)(E(y, x) \wedge \neg V(y)) \ . \end{aligned}$$

Note that NOT gates are implemented as NAND gates, and that this is the only formula in the entire set of update formulas that uses quantifiers. This formula computes the new value of a gate based on its type and the values of its inputs.

- The formula defining the set of accepting states is simply $V(0)$. If the value of gate 0 is true, the computation accepts.

7 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 circuit will be a uniform AC^0 circuit that is equivalent to a universal first-order formula, plus some control circuitry. This will allow a finite sequence of operations to write an FO formula to the circuit, and evaluate that formula over auxiliary data stored in the circuit. By writing the FO formulas defining a dynamic computation to this circuit, and emulating them, we can simulate any operation in that dynamic computation with a fixed sequence of operations on the circuit. First, we define and construct a universal first-order formula.

7.1 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 nontrivial 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 subtituting 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 7.1 *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 **??**. □

We will use this formula to construct a circuit which can emulate any first-order formula. This circuit will be the inital state of our modified SSCV problem.

7.2 A universal circuit

The circuit we construct as a complete problem for DynFO will has some features not present in traditional acyclic circuits; it should properly be called a noetwork of gates, rather than a circuit. Because a dynamic algorithm maintains state, and this state must also be maintained by a complete DynFO problem emulating it, our circuit must contain gadgets serving as persistent memory. 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. Our modified SSCV problem of size n will be initially configured to contain m of these memory cells, which we will number as $A[0], \ldots, A[m-1]$. We will determine m as a function of n later, when computing the size of the entire modified circuit.

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 $1^k\,0\,1^r\,0\,1^{rel}\,0$. 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 formu1a.

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.