Computing monadic fixed-points in linear-time on doubly-linked data structures

Steven Lindell
Department of Computer Science, Haverford College
Haverford, PA USA 19041
slindell@haverford.edu

Abstract *Detlef Seese has shown that first-order queries on bounded-degree graphs can be computed in linear-time. We extend this result by using connected doubly-linked data structures (modeled in logic over a singulary vocabulary -- one which permits only monadic predicates and functions). The first result is that first-order sentences can then be evaluated by an automaton which works directly in place on these singulary models, without changing their size or shape, and using no external resources whatsoever. In particular, this evaluation algorithm satisfies the finite-visit property: the number of times each datum is read from or written to is a uniformly limited constant. The second result analyzes the complexity of monadic fixed-points in the same vocabulary and shows that they too are in linear-time (though we use a RAM model for this).*

Introduction

Ordinarily, the descriptive complexity of first-order queries is logarithmic-space on sequential tape-based models of computation, and constant-time on parallel circuit-based models [8]. But Seese has a surprising result that first-order queries on bounded-degree graphs can be computed in linear-time [13]. This motivated us to study models of computation for linear-time and a simultaneous effort to understand their descriptive complexity in terms of fixed-point logic. This paper presents results related to that investigation.

Many attempts have been made to mathematically define linear-time computability, but the starting point for this investigation was an accurate study of both the actual material resources used by a machine as it processes its input, as well as a very careful measurement of the input size. Intuitively, such a machine should take a linear number of steps in which each step involves at most a bounded amount of work. Furthermore, the cells of the tape on which it operates should each contain at most a bounded number of bits and be adjacent to at most a fixed number of neighbors. Strict observance of these restrictions would preclude the use of a random access machine because it can violate one or both of those conditions. Nevertheless, we will need to take advantage of this more powerful model when evaluating monadic fixed-point formulas in this paper.

It seems natural to model the input as a connected *data structure* in which each node holds input symbols from a fixed alphabet, with a fixed number of links to other nodes. Moreover, and this is crucial to our results, we insist that the structures are doubly-linked (making both the in-degree and out-degree bounded). That is to say, for every incoming link there is a corresponding outgoing link which inverts it. In this way our information model most closely resembles cells on a tape of arbitrary shape. The occupied cells on the tape form the nodes of a simple graph whose edges are the directional links corresponding to adjacency between cells. If the links are static and may not be changed, it makes sense to measure the size of such a structure by its mass, which in this case is proportional to the number of nodes. What distinguishes our model of information from less restrictive models prevalent in the literature is the uniformity condition on the distribution of matter (bits). Whereas other ways of accounting for input size measure the total accumulation of bits, in our model the information density per cell is bounded. This condition becomes all the more important when we want to ensure that our model of computation does not perform more than a bounded amount of work per step.

There are two aspects to this. First, the amount of information processed within each step of the computation should be bounded. Second, we don't allow information to be communicated over arbitrarily large distances in one step of the computation. To ensure the first condition, it is simple enough to use a finite state control (the head) that reads and writes exactly one symbol per step from a fixed alphabet. The second condition in essence forbids additional heads, so our model will not allow for offline auxiliary storage. (N.b. there is no apparent physical mechanism which allows for multiple heads in traditional models such as the Turing machine unless the heads do not communicate with each other.)

Therefore we must allow for the head of our automaton to operate directly on the input data by allowing:

- *reading* the data stored at each cell (one of a fixed number of symbols);
- *writing* a symbol on a designated portion of that cell (same fixed alphabet); and
- *moving* in one of the fixed directions from that cell to a neighboring cell.

We call such a model *in situ* because all of its operations are done in place directly on the input structure.

The final requirement is one that seems more arbitrary and less natural than the previous ones, but it is nonetheless essential. This is the requirement that the head visits each cell at most a fixed number of times, uniform over all the cells of all the possible inputs. Note that different machines may have different bounds on the number of visits permitted, but any particular machine will have a fixed bound over all the cells it may encounter for any size input. There are a couple of different approaches for understanding the reasonableness of this assumption, both of which rely on the notion of self-sufficiency. The first approach has to do with heat, reversibility, and information destruction. Suppose we want our models to be self-contained in that all waste heat liberated must be kept within the model. Alternatively, the second approach accounts for the energy consumed during the course of the computation, requiring it to be provided by the model itself (as part of the input for example). In either case, if one assumes that each step of the computation consumes a certain amount of energy and liberates a certain amount of heat (reasonable for most any physical implementation) both of which must be supplied by and contained by the model, then it is not too difficult to show that a model of computation embedded in a finite-dimensional space must not perform an unlimited number of operations at any one position on the tape. It is not the purpose of this paper to fully justify this condition, but suffice it to say that other "axioms" need to be presumed such as quantization of energy and the non-achievability of perfectly lossless transmission.

In any event, the finite-visit condition automatically guarantees that a model is linear-time, and our first result, regarding first-order sentences, revolves around this particular form of linear-time computability on data structures. It slightly sharpens Seese's result by showing that a first-order sentence can always be evaluated using a finite-visit automaton. Our second result shows that any monadic fixed-point can be evaluated in linear-time using a RAM model. We leave open the question of whether this latter result can be improved to satisfy the more restrictive finite-visit condition.

Comparison with previous work

Pointer machines that compute directly on their inputs (what we are calling *in situ* computing) were originally studied by Kolmogorov [9] and Schönhage [12]. Nice surveys can be found in [5] and [1]. Both models allowed pointers to change and had bounded out-degree, but the Kolmogorov model requires symmetric links which therefore implies bounded in-degree. The automatic computing machines of Turing were originally defined using static tapes with an arbitrary fixed spatial arrangement of cells, though it is important to realize that he implicitly assumed a symmetric adjacency relationship between them (just like we do explicitly in this paper). So clearly, although our machine has links, it is tape-based and not a pointer model.

As already mentioned, the uniform density restriction that we place on cell capacity limits the amount of information that can be stored in each cell, which means our primary model is symbol-based and not word-based like a RAM. Furthermore, the fact that we have limited communication distance essentially reduces this model to a single-head automaton, precluding multiple heads often found in the literature (and *a fortiori* random-access models). Placing a uniform power restriction per step limits our model to manipulation of symbols versus operations on words, which only serves to reinforce that it cannot dynamically change pointers. Finally, our self-powered assumption in a finite-dimensional space produces the finite-visit property of Hennie which is equivalent to having a uniform distribution of work over the input [7].

Summary of paper

The preliminary section introduces our models of information in a singulary vocabulary, and goes on to define local types (confined to a certain radius) and global types (expressed by threshold quantifiers) on these models. After a brief description of greedy methods for finite-visit traversals, the main results of the paper follow in the next two sections. The first uses the standard form of Hanf's lemma to construct finite-visit automata for evaluating first-order sentences. The second states an enhanced version of Hanf's lemma for formulas and applies that to construct a linear-time evaluation method for the (monadic) fixed-point of any monotone formula on a random

access model of computation. We conclude with a conjecture that the latter result can be improved from a RAM to a finite-visit automaton.

Preliminaries

Models

Our model of information, the *data structure*, is common to both the formulas used for logical definitions, and the machines used for computations. As explained in the introduction, these finite models consist of a connected network of nodes, each representing a cell that can hold any of s different symbols from a fixed alphabet $\Sigma = \{\sigma_1, \ldots \sigma_s\}$, the cells (doubly) linked to one another by d pointers drawn from a fixed set of directions $\Delta = \{\delta_1, \ldots \delta_d\}$ and their inverses. The language we use to represent these data structures as finite models is a monadic vocabulary with unary predicates and bijective unary functions. These vocabularies have been called *singulary*; a term coined by Church [2] (though he did not have the invertibility requirement).

Definition: A model of information is a data structure M with finite domain N (the nodes of the structure) which uses monadic predicate symbols P_i, $1 \le i \le s$, and monadic function symbols f_j, $1 \le j \le d$, interpreted as:

- $P_i(c)$ is the unary relation indicating the presence of symbol σ_i in cell c,
- $f_j(c)$ is the unary bijection whose value is the cell neighboring c in direction δ_j,

These come along with a distinguished constant o designating the *origin* of the structure (a place to start the automata). The only true relation allowed will be equality.

$$M = \langle N, =, P_1, \ldots P_s, f_1, \ldots f_d, f_1^{-1}, \ldots f_d^{-1}, o \rangle$$

where: $|N| < \infty$; each $P_i \subseteq N$; each f_j: $N \leftrightarrow N$; and $o \in N$.

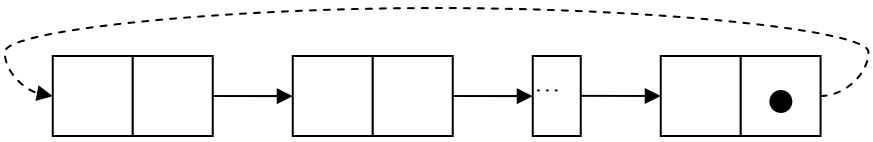
Each *node* contains a place for storing one of s different *symbols* (the data) and $2d$ (the degree) different *links* (the structure) to other nodes. In our case, the links are *bidirectional*, which means that each function f_j has a corresponding inverse function f_j^{-1} whose interpretation satisfies $f_j^{-1} \circ f_j = f_j \circ f_j^{-1} = I$. Notice that over the set of all nodes, the collection of links determines the shape of the structure and the collection of symbols determines its size $n = |N|$. A node can be illustrated as follows (inverse links are not shown):

locally represents	σ_1	symbols	σ_s	globally represents
the data.	0	…	1	*-- the size.*
the structure.	•	…	•	*-- the shape.*
	δ_1	links	δ_d	

Pictorial of an individual node (inverse links not shown)

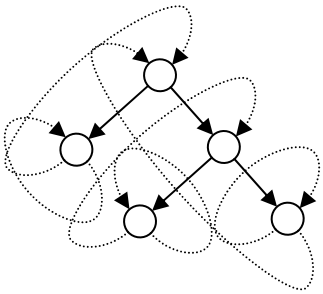
By construction, these data structures form a regular undirected graph of even degree. Realize that a nil pointer is never allowed because that would imply that a link function would be undefined at that node. Even if a special "ground" element was defined, it would be impossible to make nil pointers invertible, as required by the signature.

In order to deal with *nils* that are commonly used to terminate a data structure, it is necessary to convert them to *threads* (which has no bearing on definability). For each link direction add a new monadic predicate which indicates whether a link is non-nil. Whenever a nil link is desired, direct that link to the most distant ancestor in the opposite direction (guaranteed to exist since our structures are finite). For example, in a list the last element would be connected to the first element, forming a cycle. However, that link would be specially marked (shown as a dot in the box, and drawn as a dotted line in the figure below). The reverse links are treated similarly.



Converting a nil to a thread in a list (only one direction is shown)

Ensuring each function is injective means that it forms a disjoint collection of cycles in that direction. Note that self loops are allowed. Here is an example for the tree:



Converting nils to threads in a tree (inverses not shown)

Quantifier-free formulas

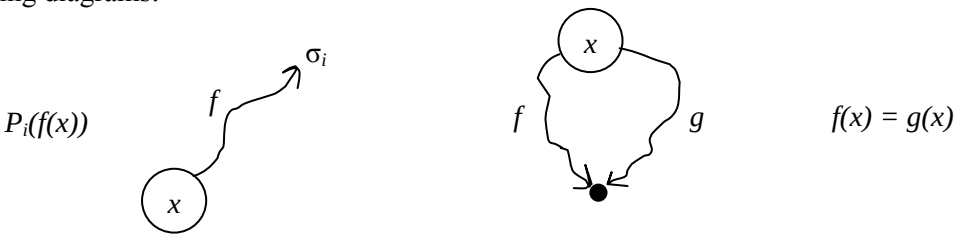
In a singulary vocabulary, it is relatively easy to see how quantifier-free single variable formulas are known as *local*. Consider the infinite set of quantifier-free formulas in a single variable over a fixed vocabulary,

$$\Lambda = \{\lambda(x) : \lambda(x) \text{ is quantifier-free}\}.$$

Hence any member of Λ must be a Boolean combination of formulas of the form:

$$P_i(f(x)) \qquad P_i(f(o)) \qquad f(x) = g(x) \qquad f(x) = g(o) \qquad f(o) = g(o)$$

where f and g stand for any sequence (possibly empty) of function symbols from $\{f_1, \ldots f_d\}^*$. For example, consider the following diagrams:



The relation $P_i(f(x))$ asserts that σ_i occurs at the node whose location is specified by the sequence of directions in Δ^* corresponding to f (which says something about the symbolic data contents near x) and the equation $f(x) = g(x)$ asserts that $f \circ g^{-1}$ determines a cycle which begins and ends at x (this says something about the shape of the structure about x). Each $\lambda(x)$ is known as a *local* formula, since satisfaction of $\lambda(m)$ depends only on a fixed size neighborhood about m (where we define the *neighborhood* of radius r about a node m to be the submodel induced by restriction to nodes of distance at most r away from m). If we take $\lambda(x)$ in Λ_r, defined as the restriction of Λ to those formulas whose function symbol composition is limited to length at most r, then we say $\lambda(x)$ is an *r-local* formula, one whose satisfaction depends only on the neighborhood of radius r. Conversely, any *r-local* formula (even one with quantifiers) is equivalent to one in Λ_r (since our functions are always bijective). Because of this one-to-one correspondence between neighborhoods and local formulas, there are only finitely many inequivalent formulas in Λ_r.

Local types

Define the local type of a node to be the conjunction of all r-local quantifier-free formulas that are true of it.

Definition: For a given radius r, the *local type* of a node m in M is the formula $\lambda(x) =$

$$\tau_r^M(m) = \wedge\{\psi(x) \in \Lambda_r : M \models \psi(m)\}$$

Note that since Λ_r is finite, $\lambda(x)$ is itself a member of Λ_r. Nodes with the same r-type look identical out to distance r in that the neighborhoods of radius r about two nodes with the same r-type are isomorphic. When r and/or M are/is understood, we will drop the subscript and/or superscript referring to $\tau(m) = \lambda(x)$ as the local type of m.

The key observation to make is that there are a bounded number of nodes in any neighborhood of radius r, and therefore only finitely many non-isomorphic neighborhoods of that radius. Hence there are only finitely many different distinct r-types over all singulary models in a given vocabulary.

Counting quantifiers

An *r-local* formula $\lambda(x)$ is realized by an element m if it makes $\lambda(m)$ true, and define the *index* of $\lambda(x)$ in M to be the number of times it is realized in M:

$$|\{m \in M: M \models \lambda(m)\}|.$$

If the index is zero, then it is not realized in M. On the other hand, indices can be as large as the size of M. Now we turn to expressing these indices by fixing r and consider the infinite collection of global sentences,

$$\Gamma = \{(\exists^{\geq t'}x)\ \lambda(x): \lambda(x) \in \Lambda_r\} \cup \{(\exists^{<t'}x)\ \lambda(x): \lambda(x) \in \Lambda_r\}$$

where $\exists^{\geq t'}$ ($\exists^{<t'}$) is the *threshold quantifier* which says that there are at least t' (resp. less than t') distinct elements satisfying the formula that follows it. So the sentences of Γ are able to say how many occurrences there are of any particular r-type. Since our vocabulary includes equality, each member of Γ is clearly expressible as a first-order sentence. But to restrict our attention to finite collections we limit the threshold, only allows counting r-types only up to a specified t,

$$\Gamma_r^t = \{(\exists^{\geq t'}x)\ \lambda(x): t' \leq t, \lambda(x) \in \Lambda_r\} \cup \{(\exists^{<t'}x)\ \lambda(x): t' \leq t, \lambda(x) \in \Lambda_r\}$$

Global types

We are now ready to define the global type of a structure for a given radius r and threshold t by counting its r-local types up to t.

Definition: Given a radius r and threshold t, the *global type* of M is the sentence $\gamma =$

$$T_r^t(M) = \wedge\{\theta \in \Gamma_r^t: M \models \theta\}$$

When r and t are understood, we drop the subscript and superscript and just refer to $T(M) = \gamma$ as the global type of M. Note well that for a given radius and threshold there are only finitely many global types possible over all models M in a fixed vocabulary.

Two models with the same global type have the same multiplicity of r-types up to a certain threshold t. So if you traveled around each model and tallied the different kinds of radius r neighborhoods you could stop counting each kind at the threshold t in order to determine that they had the same global type. Only a fixed finite amount of information is required for this comparison, regardless of their sizes. We shall see in the next section that two models with the same global type will satisfy the same first-order sentences up to a certain quantifier depth, by applying Hanf's Lemma on bounded-degree graphs to data structures.

Finite-visit machines

The somewhat longwinded explanation given in the introduction partially justifies our chosen model of computation: the finite-visit automaton. Here is a more concise definition of this machine model.

Definition: Let L be a singulary vocabulary. A *finite-visit automaton* over L is a finite state machine that can operate by moving around any data structure in the signature L, reading or writing symbols from the alphabet (i.e. modifying the contents of nodes, but not the links between them), with the additional property that over all such structures, each element in the structure is visited $O(1)$ times.

Note that the (symmetric) bounded-degree property of our singulary structures ensures it doesn't matter whether we count visits to nodes or edges.

Finite-visit tours

Before beginning, it is necessary to first illustrate how one would traverse a data structure in a finite-visit fashion, using the ancient depth-first search technique, by thinking of it as a regular graph in which each node has $2d$ labeled edges. This is a fundamental component of our evaluation algorithm, and we will do this by implementing a depth-first search directly on nodes without using recursion or any auxiliary storage like a stack. All that is necessary is to use a greedy method to mark edges as they are traversed in each direction, similar to the idea described in Trakhtenbrot [14, chapter 3]. (For an historical discussion of the 19th century method for navigating a maze see [3].) Technically, edges cannot be marked, so we shall mark each endpoint using numbers from the set $\{1, \ldots, 2d\}$ (one for each possible edge f_j, including inverses f_j^{-1}, $1 \le j \le d$). We write these additional symbols "on top of" any symbols already present at a node. This is no different than allowing multiple predicates to be active, and whereas the input predicates are read-only, these output predicates must be read-write.

First let us define our moves:

- Moving *forwards* means crossing an edge that has not been traversed in either direction.
- Moving *backwards* means crossing an edge in the direction opposite from which it has already been traversed.

Moreover, edges are never crossed twice in the same direction. This simultaneously guarantees the finite-visit property and eliminates the need to mark a node more than once with the same symbol.

Initially, all nodes are unvisited. Mark a node *visited* when leaving it for the first time. Upon encountering an already visited node, to go "*back*" means to reverse direction, i.e. move backwards along the edge that was just traversed in the forwards direction (keep track of previous move in the finite control). Here is the algorithm in table format.

Depth-First tour

Previous move	**Comment**	**Next move**
None	Starting [at origin]	visit; move *forwards*
Forward	Exploring	**If** already visited **then** go *back* **else** visit; move *forwards* (see note below)
Backward	Retracing	**If** possible, move *forwards*, **else** **If** possible, move *backwards*, **else** *Stop*

Note: one can always go forward upon first entering a node

With just a moment's reflection the reader can check that this is equivalent to a standard depth-first tour. In reality, there are many ways to design a finite-visit traversal of a connected bounded-degree graph, and any method suffices for our purposes. In what follows, the *visit* procedure will be replaced by subroutines that perform various calculations.

First-order sentences

The starting point of our investigation is to observe that every first-order sentence is equivalent to a disjunction of global types (for some radius and threshold). This follows from Hanf's Lemma for sentences, reproduced here for the singulary vocabulary of data structures.

Hanf's Lemma for sentences

Lemma: Given a quantifier-depth q, there is a radius r ($= 2^q$) and a threshold t ($= qd^r+1$) such that if M_1 and M_2 are any models which share the same global type, then they satisfy the same first-order sentences θ of quantifier-depth q.

$$\mathrm{T}^t_r(M_1) = \mathrm{T}^t_r(M_2) \qquad \Rightarrow \qquad M_1 \models \theta \;\Leftrightarrow\; M_2 \models \theta$$

Proof: The same pebble-game argument shown for ordinary graphs ([8], p. 103) can be used without modification because although additional unary relations (or constants) increase the variety of local formulas (and therefore types), they do not change the distance metric or degree of the graph. Furthermore, our use of links (bidirectional monadic functions) instead of edges (a binary relation) in no way changes the substance of the argument.

Corollary: Every first-order sentence is equivalent to a (finite) disjunction of global types.
Proof: This follows immediately from Hanf's Lemma via the construction:

$$\theta \qquad \Leftrightarrow \qquad \vee\{\mathrm{T}^t_r(M) : M \models \theta\}$$

Finite-visit algorithms for first-order sentences

We are now in the position to provide a method of evaluating first-order sentences by computing directly on the input structure, changing neither its shape (links are not assignable) nor its size (number of nodes is fixed).

Theorem: In a singulary vocabulary, every first-order sentence can be evaluated by a finite-visit automaton computing directly on the data structure.

Proof: By the corollary above, any sentence is equivalent to a finite Boolean combination of sentences:

$$(\exists^{\geq t} x)\ \lambda(x) \quad \text{or} \quad (\exists^{<t} x)\ \lambda(x)$$

where $\lambda(x)$ is a local formula in Λ_r. So it suffices to show how to compute the threshold quantification of a local formula, by examining the local neighborhood of each node along the tour described earlier.

Basically, each time we visit a node, we examine its local neighborhood to see if it satisfies $\lambda(x)$ by looking at the symbols along all paths of length at most r leading away from x, together with determining which paths of length at most $2r$ are cycles which begin and end at x. This procedure is a preprogrammed routine which gathers a constant amount of information about the local neighborhood in a read-only way, and is clearly $O(1)$ work. Each such examination induces a bounded number of visits to the nodes in that local neighborhood. On the whole tour, each node is visited a uniformly bounded number of times by the nodes in its own local neighborhood, maintaining the finite-visit property. In the finite control, we can calculate $\lambda(x)$ based on information gathered from each examination, and hence keep track of how many nodes satisfy $\lambda(x)$ by tallying them up to t, which is a constant independent of the number of nodes. Once we get beyond the threshold it is not necessary to keep track anymore. This completes the proof.

As a corollary, this proves Seese's theorem, since finite-visit machines clearly take a linear number of total steps. Extending this idea to monadic fixed-points is the subject of the next section.

Fixed-point formulas

Let $\varphi(x, P)$ be a first-order formula of one free variable x in the singulary vocabulary of data structures with equality. An additional unary relation not present in the original vocabulary is represented by the monadic predicate P, assumed to appear only positively in φ, (i.e. under an even number of negation signs), in which case we say that φ

is *monotone* in P [11]. Let $m \in N$ refer to any node in the domain of M, and let $S \subseteq N$ be any subset thereof. Let $\varphi^M(S) = \{m : M \models \varphi(m; S)\}$ written $\varphi(S)$ whenever M is understood.

Definition: The *least fixed-point* (LFP) of φ, denoted φ^∞, is the smallest subset S of N satisfying $\varphi(S) = S$. It can be computed inductively in stages by iterating

$$\varphi^{i+1} = \varphi(\varphi^i)$$

from the empty set $\varphi^0 = \varnothing$ until achieving a stable value $\varphi^l = \varphi(\varphi^{l-1}) = \varphi^\infty$ after l stages.

The naïve algorithm for computing φ^∞ on M would mark the vertices satisfying S (initially none) and repeatedly apply $S \leftarrow \varphi^M(S)$ until no more are marked (which must happen in at most $n = \|M\|$ stages). Since each stage requires $O(n)$ time, the whole thing is $O(n^2)$. Our algorithm avoids this breadth-first calculation, relying instead on both the monotonicity and locality of φ to compute φ^∞ depth-first *without* going stage by stage.

Some preliminaries are required before we can prove the main result. The first is a locality result which shows that deciding which elements satisfy the induction can be determined solely by the current global type of the model together with the current local type of the element. It is an extension of the locality result in the previous section. The second shows how monotonicity can be exploited to compute the least fixed-point using a greedy method of evaluation. It is peculiar to fixed-point logic.

Hanf's lemma for formulas

Hanf's Lemma can be extended from sentences to formulas of one free variable.

Lemma L: Given $\varphi(x)$ of quantifier-depth q, there is a radius r' and a threshold t' such that if $T_{r'}^{t'}(M_1) = T_{r'}^{t'}(M_2)$ and $\tau_{r'}(m_1) = \tau_{r'}(m_2)$ then $M_1 \models \varphi(m_1) \Leftrightarrow M_2 \models \varphi(m_2)$.

Proof: We will be applying Hanf's lemma for r and t as follows. Choose $r' = 2r$ (twice the radius required) and $t' = t + d^{r+1}/(d-1)$ (a larger threshold also). Clearly, $T_{r'}^{t'}(M_1) = T_{r'}^{t'}(M_2) \Rightarrow T_r^{t'}(M_1) = T_r^{t'}(M_2)$ and so M_1 and M_2 have the same number of r-types up to t'. Expanding the vocabulary by a distinguished constant will only affect the r-types of nodes within distance r of that constant. But by hypothesis, the neighborhoods of radius $2r$ around m_1 and m_2 are isomorphic, giving a one-to-one correspondence between the radius r neighborhoods around nodes of distance r from m_1 and the radius r neighborhoods around nodes of distance r from m_2 in both models. This means that in each model, the multiplicity of local r-types is changed in exactly the same way. This maintains the global type equivalence, causing the gain of an identical number of brand new types and the loss of up to $d^{r+1}/(d-1)$ old types. But because t' is large enough, we still have $T_r^t(M_1) = T_r^t(M_2)$. Now apply Hanf's lemma to get $(M_1, m_1) \models \varphi(x)[m_1] \Leftrightarrow (M_2, m_2) \models \varphi(x)[m_2]$, which is the same as showing $M_1 \models \varphi(m_1) \Leftrightarrow M_2 \models \varphi(m_2)$.

So the only thing we need to know about a model M to see if it satisfies $\varphi(x)$ for a particular element m is the quantity up to t' of each r'-type, together with r'-type of m.

Corollary L: Let $\lambda(x) = \tau_{r'}(m)$ be the local type of m in M, and let $\gamma = T_{r'}^{t'}(M)$ be the global type of M. Then γ together with $\lambda(x)$ determine whether $M \models \varphi(m)$.

Proof: This follows immediately from Lemma L.

This means that in the midst of an induction computing S, determining $(M, S) \models \varphi(m)$ depends entirely on a finite amount of information. In order to utilize that information effectively, we need flexibility in the order of application, which leads us to our next lemma and corollary.

Greedy evaluation via monotonicity

A basic consequence of positivity in P is that the operator induced by φ is monotone: $S \subseteq S' \Rightarrow \varphi(S) \subseteq \varphi(S')$. It follows that the stages $\varnothing = \varphi^0 \subset \ldots \varphi^i \subset \varphi^{i+1} \ldots \subset \varphi^l = \varphi^\infty$ form a strictly increasing sequence which stays below the least fixed-point φ^∞ [11]. In this section we will show that *any* other monotone method of using φ to compute φ^∞ shares the same basic properties.

Lemma M: If $S \subset \varphi^\infty$ is strictly below the fixed-point, then:

(a) $\exists a \in \varphi(S) \bullet a \notin S$ (φ finds something new); and
(b) $\forall a \in \varphi(S) \bullet a \in \varphi^\infty$ (everything φ finds is safe).

Proof: As a direct consequence of monotonicity, $S \subseteq \varphi^\infty$ implies $\varphi(S) \subseteq \varphi(\varphi^\infty) = \varphi^\infty$, which is a restatement of (b). The fact that $S \neq \varphi^\infty$ implies there is a largest i such that $\varphi^i \subseteq S$. Since $\varphi^{i+1} = \varphi(\varphi^i) \subseteq \varphi(S)$ by monotonicity, the maximality of i implies that $\varphi(S) - S \neq \varnothing$ which shows (a). QED

In other words, for all S, $S \subset S \cup \varphi(S) \subseteq \varphi^\infty$, which means that as long as we always increase along an upward path according to φ, we will eventually reach the least fixed-point, even though nodes may be added in a completely different order than usual. In particular, a greedy method can be used in which only one element is thrown in at each step.

Corollary M: Take any maximal length sequence $m_1,\ldots, m_k$ of distinct nodes for which $m_i \in \varphi(\{m_1,\ldots, m_{i-1}\})$. Then the monotone series of relations $S_i = \{m_1,\ldots, m_i\}$, for $i = 0\ldots k$, reaches the fixed-point of φ.

$$\varnothing = S_0 \subset \ldots S_i \ldots \subset S_k = \varphi^\infty$$

Proof: In the lemma, part (a) prevents premature termination so that there always is an $m_i \in \varphi(S_{i-1})$, for each $i = 1\ldots k$, and part (b) ensures there is no overshoot so that $m_i \in \varphi^\infty$ for every $i = 1\ldots k$.

The linear-time fixed-point evaluation algorithm

Our assumptions are to use a unit-cost RAM model with $O(\log n)$-bit word size, whose locations can be used to store pointers and numbers.

Given a P-positive formula $\varphi(x, P)$, the key to our approach is to work with types over the expanded vocabulary, dynamically maintaining a list of nodes $L_{\lambda(x)}$ for each r-type $\lambda(x)$ in Λ_r (using the proper radius r), together with its size $|L_{\lambda(x)}|$. Moreover, each node m in M is doubly-linked to its unique corresponding position in the list $L_{\tau(m)}$. Insertions into a list are done at the head, deletions are done in place, and the list's size is incremented or decremented accordingly. Each operation takes $O(1)$ time for a unit cost RAM.

Theorem: If $\varphi(x, P)$ is a P-positive first-order formula in a singulary vocabulary, then its least fixed-point φ^∞ can be evaluated in linear-time by a random-access model of computation.

Proof: Our method starts with S empty, and marks one node at a time until S reaches the fixed-point φ^∞. We will always select a new element within constant time from among the candidates that currently satisfy $\varphi(S)-S$, keeping track of S by directly marking nodes in M.

Step 0**:** *Initialize*

At the outset, scan the radius r neighborhood of each node m to determine its local type $\lambda(x) = \tau(m)$, and insert m into $L_{\lambda(x)}$. Since S is empty, every node is unmarked and hence all the lists for marked types will be empty (actually, these lists do not need to be maintained, but we still need their sizes). This step takes linear time but is executed only once. A type is called *marked* if $\lambda(x)$ implies $P(x)$.

Step 1**:** *Determine the types*

Measure all the sizes $\{|L_{\lambda(x)}| : \lambda(x) \in \Lambda_r\}$ up to t (for the proper threshold) to get $\gamma = \mathrm{T}(M)$, from which we can determine by table lookup in constant-time the local types

$$Q = \{\, \lambda(x) \in \Lambda_r : \gamma \models \lambda(x) \rightarrow \varphi(x)\}$$

telling us which elements should be marked. If we have not reached the fixed-point then some elements satisfying a formula in Q have yet to be marked.

Step 2**:** *Choose a node to mark*

If every type in Q is already a marked type, we are done. Otherwise, select the first node m_0 on some non-empty list of an unmarked type in Q and mark it. N.b. that every node on such a list is necessarily unmarked. This is clearly $O(1)$ time.

Step 3**:** *Update the types*

The key point in the algorithm derives from the symmetry of distance in a model: marking m_0 only changes the local types of the finitely many nodes within radius r. So delete each such node m from its old list $L_{\lambda(x)}$ (where $\lambda(x)$ was the local type of m before it was marked) and insert it into its new list $L_{\lambda'(x)}$ (where $\lambda'(x)$ is the local type of m after it is marked). Now go to Step 1.

The algorithm is correct since by Corollary M, it always chooses to mark a previously unmarked element satisfying φ. And each step is constant time on a RAM since by Corollary L, selecting this element is done by consulting only a fixed (finite) amount of information. The actual bookkeeping required to update the list sizes (one increment/decrement operation) and manipulate the list pointers (one insert/delete operation) is also $O(1)$.

Conclusion

Although we have shown that first-order sentences can be evaluated with a finite-visit automaton, and monadic fixed-points can be evaluated in linear-time, the nagging question remains as to whether the best of both results can be combined to demonstrate a much more elegant result which we leave as a conjecture:

Conjecture: In a singulary vocabulary, every monadic fixed-point can be evaluated by a finite-visit automaton computing directly on the data structure.

Moreover, it is not at all clear whether the main result can be extended to alternating fixed-points, in which negations of fixed-points can be nested. This would have interesting connections with the modal μ-calculus.

Acknowledgments

I owe a debt of thanks to the many conversations I had with my colleague Scott Weinstein, and I express my sincere gratitude to the help and support I have received from my wife for reviewing and proofreading this work. Her contribution has been invaluable. Appreciation also goes to a number of anonymous reviewers who made useful comments.

References

[1] Ben-Amram, Amir M. ‘What is a “Pointer Machine”?’ SIGACT News, 26(2), pp. 88-95, June 1995.

[2] Church, Alonzo *Introduction to Mathematical Logic* Princeton University Press, 1956.

[3] Even, Shimon *Graph Algorithms*, Computer Science Press, 1979.

[4] Gaifman, Haim ‘On local and non-local properties’, in J. Stern (ed.), *Logic Colloquium '81* (Amsterdam: North-Holland, 1982).

[5] Gurevich, Yuri ‘Kolmogorov machines and related issues’ Bulletin of the European Association for Theoretical Computer Science, Number 35, June 1988, 71--82.

[6] Hanf, W. ‘Model-theoretic methods in the study of elementary logic’ in *The Theory of Models* (J. Addison, L. Henkin and A. Tarski, eds.), pp. 132 – 145 (North Holland, Amsterdam 1965).

[7] Hennie, F.C. ‘One-tape, off-line Turing machine computations’ Inf. and Con. 8, 1965. 553-578.

[8] Immerman, Neil *Descriptive Complexity*, Graduate Texts in Computer Science, Springer, New York, 1999.

[9] Kolmogorov, A.N., and Uspensky V.A. ‘On the Definition of an Algorithm’, In Russian, English translation: Amer. Math. Soc. Translat., 29:2, (1963) 217-245.

[10] Libkin, L. ‘On counting logics and local properties’ ACM TOCL **1**(1) (2000).

[11] Moschovakis, Yiannis *Elementary Induction on Abstract Structures*, North-Holland, 1974.

[12] Schönhage, Arnold ‘Storage modifications machines’ SIAM Journal on Computing, 9(3):490--508, August 1980.

[13] Seese, Detlef ‘Linear time computable problems and first-order descriptions’ Mathematical Structures in Computer Science, 6(6):505-526, December 1996.

[14] Trakhtenbrot, Boris *Algorithms and automatic computing machines* (Translated and adapted from the 2d Russian ed. (1960) by Jerome Kristian, James D. McCawley, and Samuel A. Schmitt) Boston, DC Heath and Company, Lexington, Massachusetts, 1963.