Distributed computation of Web queries using automata

(extended abstract)

Marc Spielmann, University of Limburg[*]
Jerzy Tyszkiewicz, Warsaw University
Jan Van den Bussche, University of Limburg

Abstract

We introduce and study a distributed computation model for querying the Web. Web queries are computed by interacting automata running at different nodes in the Web. The automata which we consider are essentially register automata equipped with an additional communication component. We identify conditions necessary and sufficient for systems of automata to compute Web queries, and investigate the computational power of such systems.

1 Introduction

Recently, much attention has been paid to querying the Web [FLM98]. A salient feature of Web computations is their browsing nature, which led Abiteboul and Vianu [AV00] to formally define a *Web query* as a function mapping pairs (I, s) to sets of nodes in I, where I is a Web instance and s is the Web node where browsing starts (the *source*). They also introduced the *browser machine*, a Turing machine which can navigate the Web by following links.

Another recent development is that of Internet supercomputing [FK98, Fos00], where many individual computers linked to the Internet collaborate in a distributed computation. An appealing and popular example is the SETI@home project, which scans radio signals from space for signs of extraterrestrial intelligence [KWA$^+$01].

We were thus inspired to combine these two lines of research by investigating the possibilities and limitations of *Web automata*: a computation model for Web querying that, like the browser machine, is still purely navigational, but which is also distributed. Starting at the source, a finite portion of the Web reachable from the source by links is populated with lightweight processes. This finite portion is typically determined by specifying a maximum number of links to follow down (as commonly done in tools for off-line browsing and Web mirroring). The processes work concurrently, following a program specified essentially as a finite register automaton [KF94]. The processes report back to the source process by sending messages upwards along the edges of a spanning tree (a standard network topology used in computer networks and distributed computation [Tan96, AW98]).

We offer the following contributions:

(i) We define a fair, efficient, and easy to enforce communication protocol by which the distributed computation proceeds in rounds.

[*]Contact address: Marc Spielmann, University of Limburg (LUC), Departement WNI, Universitaire Campus, B-3590 Diepenbeek, Belgium. Phone: +32-11-268209. Fax: +32-11-268299. Email: marc.spielmann@luc.ac.be.

Each round has a layered structure according to the levels of the spanning tree. The processes at each level work concurrently, and each level takes only constant parallel time. After each round, the source process is guaranteed to receive enough data to decide whether to continue for another round, or to terminate the computation. In addition, the source process may produce output. We identify a decidable property of Web automata, called *productivity*, which enables this protocol. Testing productivity is PSPACE-complete.

(ii) Because the order of upward communications within a layer is not fixed, there are many possible runs for a given spanning tree. On top of that, the spanning tree itself arises out of the computation and is thus not a priori fixed. A Web automaton which produces the same output for every possible run on every possible spanning tree is called *sound.* Every sound Web automaton computes a well-defined Web query. We show that soundness is undecidable (this is not entirely evident given the finite nature of Web automata and the rather rigid communication protocol they must follow).

(iii) A sound Web automaton computes a Web query, but the user is not guaranteed to see new output every round, which can be quite undesirable in practical scenarios. A Web automaton that does produce new output every round is called *continuous.* In particular, a continuous Web automaton that computes a yes/no query already knows the answer in one round. Continuity is also shown to be undecidable. Note the analogy with Abiteboul and Vianu's distinction between *finitely* and *eventually* computable Web queries: of a query that is only eventually computable, the user is never sure that he has seen all of the output.

(iv) When an order on the outgoing links from every node is available (a very natural assumption in the context of the Web), we show that every logarithmic-space computable Web query is computable by a Web automaton.

(v) We introduce a natural syntactic subclass of Web automata called DAF (Decide And Forward). As the name suggests, a DAF automaton makes all the crucial decisions already after the first round; the subsequent rounds are pure forwarding rounds which merely flush the remaining contents of the communication queues to the output. We show that soundness and continuity of DAF automata becomes decidable in the monadic case i.e., when the tests performed by automata are based on unary predicates of Web nodes only. Furthermore, we give a characterization of the Web queries that are continuously computable by these monadic DAF automata, in terms of a fragment of first-order logic.

(vi) Finally, we introduce a restricted version of the browser machine, which can use its Turing tape only in a stack-like manner similar to the working of the 'back' and 'forward' buttons of common Web browsers, and has only a finite work memory. When in addition the depth of the stack is bounded (so that the machine cannot get 'lost in hyperspace'), we show that every Web query computable by such a browser stack machine is also computable by a Web automaton.

As for related work, practical distributed Web querying systems similar to our theoretical model are already being proposed, e.g., the DIASPORA system [GHR00]. Few theoretical studies of navigational Web querying have been published; that is of course, after the original papers by Abiteboul and Vianu [AV00], and Mendelzon and Milo [MM98]. These two papers focused on computational completeness, while we work with a limited computation model and focus on distribution and efficiency. A very recent proposal of a formal model for Web querying using concurrent agents was made by Sazonov [Saz]; his model is based not on finite automata but on a set-theoretic term language. Finite automata working over abstract domains (Web nodes in our case) rather than over finite alphabets have, of course, been considered before, e.g., in the study of regular languages over infinite alphabets [KF94, NSV01]. Finally, we mention that in our definition of distributed runs of Web automata we were inspired by Gurevich's definition of partially ordered runs of distributed abstract state machines [Gur95].

2 Web Queries

Web Instances. Fix a vocabulary Υ consisting of predicate symbols of various arities, including at least a binary symbol *Link*. We define a *Web instance* as a finite relational structure over Υ. The elements of the instance are called *nodes*, and are abstractions of Web pages, Web sites, or other objects on the Web. The ordered pairs in the *Link* relation represent links in the Web.

The other predicate symbols are abstractions of the semantic predicates a Web query will apply to nodes. To give a few random examples, a unary predicate $P_1(x)$ could stand for "Web page x contains the keyword Madison"; a binary predicate $P_2(x, y)$ could stand for "the link from x to y is labeled Madison"; and a ternary predicate $P_3(x, y, z)$ could stand for "on page x, all links to y precede all links to z". The vocabulary will thus vary from query to query.

Although a Web instance is nothing but a standard relational database with at least one binary relation, there is a crucial difference with standard relational database queries: to answer a Web query, we can examine this 'relational database' only by following links starting at some source node. This brings us to the next basic definition.

Web Queries. A *Web query* Q over Υ is a mapping that assigns to every pair $(\mathcal{I}, s)$, where $\mathcal{I}$ is a Web instance over Υ and s is a node in $\mathcal{I}$ called the *source*, a set of nodes in $\mathcal{I}$. Following the standard genericity criterion for database queries, Q must preserve isomorphisms, i.e., if $(\mathcal{I}', s')$ is isomorphic to $(\mathcal{I}, s)$ via an isomorphism ι, then $Q(\mathcal{I}', s') = \iota(Q(\mathcal{I}, s))$. Moreover, since we will consider purely navigational computation models only, it is only fair to accordingly require that $Q(\mathcal{I}, s) = Q(\text{Reach}(\mathcal{I}, s), s)$, where $\text{Reach}(\mathcal{I}, s)$ denotes the substructure of $\mathcal{I}$ generated by the nodes reachable from s by following links.

Note that every first-order formula $\varphi(s, x)$ over Υ defines a Web query $Q_\varphi : (\mathcal{I}, s) \mapsto \{n : \text{Reach}(\mathcal{I}, s) \models \varphi[s, n]\}$.

3 Web Automata

We begin the introduction of our computation model by defining the class of programs that can be executed by Web automata, and defining local runs of automata at individual Web nodes. In the next section we then define distributed runs of systems of automata.

Web Automata. Formally, a *Web automaton* is a triple $(\Upsilon, \bar{r}, \Pi)$, where Υ is the vocabulary, $\bar{r}$ is a tuple $(r_1, \dots, r_\ell)$ of *register variables*, and Π is a*program* over Υ and $\bar{r}$, which we define next.

To define programs, we add to Υ constant symbols 0, 1, and $\perp$, and the unary predicate symbol *Source*, yielding an expanded vocabulary Υ^+. The symbols 0 and 1 will represent the two bits, while *Source* will indicate the source node. The symbol $\perp$ will stand for the empty queue.

A *guard* is a quantifier-free first-order formula $\varphi(x, y, \bar{r})$ over Υ^+. As will become clear soon, x will be interpreted as the node at which the automaton is running locally, and y will be interpreted as the head of the queue of incoming messages. The variables r_i will be interpreted as the contents of the registers of the automaton.

A *rule* is an expression of the form `if` φ `then` *action*, where φ is a guard and *action* is an *update action*, of the form $(r_i := t)$, or a *send action*, of the form $\mathit{send}(\bar{t})$. Here, t is a term in $\{x, y, r_1, \ldots, r_\ell, 0, 1\}$, and $\bar{t}$ is a sequence of such terms, possibly empty.

A *program* is now simply a finite set of rules.

Example 3.1. An example of a program for a Web automaton over a vocabulary containing a unary predicate *Interesting* is the following:

```
if $r_1 = 0 \wedge \mathit{Interesting}(x)$ then $\mathit{send}(x)$
if $r_1 = 0$ then $r_1 := 1$
if $r_1 = 1 \wedge y \neq \bot$ then $\mathit{send}(y)$
if $r_1 = 1 \wedge y = \bot$ then $\mathit{send}()$
```

$\square$

Local Runs. We next define a notion of local run which reflects the behavior of an automaton $A = (\Upsilon, \bar{r}, \Pi)$ when observed at a particular node.

Fix an instance $\mathcal{I}$ over Υ and let s be a source node in $\mathcal{I}$. Expand $\mathcal{I}$ to a structure $\mathcal{I}^+$ over Υ^+ by adding the three new elements 0, 1, and $\bot$, and by interpreting the unary predicate *Source* as the singleton $\{s\}$. In the following, the words *queue* and *message* both refer to a finite sequence of bits and nodes in $\mathcal{I}$. By the *head* of a queue we mean its first element, and by the *tail* the sequence of the remaining elements; we define the head of the empty queue to be $\bot$.

Let n be a node in $\mathcal{I}$. A *configuration of A at n* is a triple $(n, q, \bar{a})$ where q is a queue and $\bar{a}$ is an ℓ-tuple $(a_1, \ldots, a_\ell)$ of bits and nodes in $\mathcal{I}$ (recall that ℓ is the number of registers). Intuitively, a_i is the content of register r_i in this particular configuration. A program rule in Π with guard $\varphi(x, y, \bar{r})$ is said to be *enabled in* $(n, q, \bar{a})$ if $\mathcal{I}^+ \models \varphi[n, \mathrm{head}(q), \bar{a}]$. The *successor configuration* of $(n, q, \bar{a})$ is the configuration $(n, q', \bar{a}')$ such that

- if q is not empty, then $q' = \mathrm{tail}(q)$; otherwise, $q' = q$; and
- for each $i \in \{1, \ldots, \ell\}$, if there is precisely one r_i-update rule in Π which is enabled in $(n, q, \bar{a})$, then $a'_i = t[x/n, y/\mathrm{head}(q), \bar{r}/\bar{a}]$, where $(r_i := t)$ is the right-hand side of this rule; otherwise, $a'_i = a_i$.

We further say that A *sends* a message m in $(n, q, \bar{a})$ if there is precisely one send rule in Π which is enabled in $(n, q, \bar{a})$, and $m = \bar{t}[x/n, y/\mathrm{head}(q), \bar{r}/\bar{a}]$, where $\mathit{send}(\bar{t})$ is the right-hand side of this rule.

Define a *local run of A at node n* (in $\mathcal{I}$ with source s) to be a finite or infinite sequence $(C_i)_{i \in \kappa}$ of configurations of A at n such that for every $i + 1 \in \kappa$

- C_{i+1} is a successor configuration of C_i, and
- if A sends the empty message in C_i, then C_{i+1} is the last configuration of $(C_i)_{i \in \kappa}$.

Remark 3.2. One may wonder why our automata need to be able to send messages of length longer than 1. After all, an automaton could send the components of a message one after the other as messages of length 1? However, we will consider systems of communicating automata, where a receiving automaton may obtain messages from many different automata, and these messages can be intermingled. Then it is important that the receiver is able to see which messages were sent by one and the same sender. Note that messages can be flanked by separators (e.g., special bit sequences) thereby enabling a receiver to distinguish between different messages. Note also that the automaton in Example 3.1 is a very simple one which indeed sends messages of length 1, or empty messages, only. □

We are particularly interested in automata where the time between two send actions is bounded by a constant:

Definition 3.3. Let $k \geqslant 1$ be a natural number. A Web automaton A is *k-productive* if for every local run $(C_i)_{i \in \kappa}$ of A where in C_0 each register has contents 0, in any k consecutive configurations, there is at least one in which A sends a message. A is *productive* if it is k-productive for some k. □

For example, the automaton of Example 3.1 is 2-productive. We have:

Theorem 3.4. *Both the problem of deciding k-productivity for a fixed k, and the problem of deciding productivity, are* Pspace*-complete.*

4 Distributed Computations

Before we define distributed runs of systems of Web automata formally, we provide some intuition. A productive automaton A, when started at some source node s of an instance $\mathcal{I}$, begins by distributing copies of itself to all other nodes, using an obvious recursive procedure: upon creation at a node n, A equips every node which n links to with a copy of itself, except when the node is already equipped with a copy. This procedure traces out some spanning tree of the link graph. Note that in practice, we will go only a fixed number of levels deep in the Web. Also, all automata at nodes located at the same server might be implemented by a single process running at this server.

Each process now starts running concurrently with the others, and sends its messages to the process that created it. But they all follow a simple protocol based on two principles: *start running only if you can, and stop once you have sent a message.* This naturally organizes the distributed computation in *rounds*, where in each round, every automaton still active sends one message. Since an automaton reads a new message each time it moves (unless it is a leaf automaton), it must indeed wait until it has received enough incoming messages so that it can run long enough to send a message itself.

Since our automaton program is productive, rounds can always be performed in parallel time linear in the depth of the spanning tree. For instance, processes at leafs of the spanning tree never receive any messages and can always start the round, thus enabling other processes to start, etc. All nodes sent by the source process belong to the output. In every subsequent round, a process picks up its local run where it left it at the previous round. When it sends the empty message, it exits the computation and will not participate in later rounds. When the source automaton exits, the whole computation terminates.

To the formal definition. Let $\mathcal{I}$ be an instance with node set N and link set L, and let s be a source node in $\mathcal{I}$. We assume that $\text{Reach}(\mathcal{I}, s) = \mathcal{I}$; if not, replace $\mathcal{I}$ with $\text{Reach}(\mathcal{I}, s)$ in what follows. Let $\mathcal{T}$ be a spanning tree of (N, L) rooted at s.

A *global configuration* of A is a mapping γ that assigns to each node n a configuration of A at n. The *initial global configuration* maps each n to $(n, \varnothing, \bar{0})$.

Let γ and γ' be two global configurations and let n be a node. γ' is called a *successor configuration of γ via a move at n* if the following three conditions hold:

1. There exists a finite local run $(C_0, \ldots, C_k)$ of A at n such that (i) $C_0 = \gamma(n)$, (ii) $k \geqslant 1$, (iii) no message is sent in this local run before C_{k-1}, (vi) A does send some message m in C_{k-1}, and

(v) $C_k = \gamma'(n)$.

2. If $n \neq s$, let p be the parent of n in $\mathcal{T}$. Then, if $\gamma(p) = (p, q, \bar{a})$, $\gamma'(p) = (p, qm, \bar{a})$. (That is, m is appended to the queue of the parent process.)

3. For every node o different from n and p (if p exists), $\gamma'(o) = \gamma(o)$.

Let d be the depth of $\mathcal{T}$. For every $i \in \{0, \ldots, d\}$, let level(i) denote the set of nodes whose distance from s in $\mathcal{T}$ is i. Let M be a subset of N containing s. An *M-round* (along $\mathcal{T}$) is a finite sequence $(\gamma_i)_{i \leqslant l}$ of global configurations such that

- for every $i+1 \leqslant l$, γ_{i+1} is a successor configuration of γ_i via a move at some node n_{i+1}
- the sequence $n_1 \ldots n_l$ is an enumeration of M, and
- for each $i \in \{0, \ldots, d\}$ there exists an enumeration $\bar{e}_i$ of level(i) $\cap M$ such that $\bar{e}_d \ldots \bar{e}_0 = n_1 \ldots n_l$.

The *output* produced during this round is the set of nodes occurring in the message sent at s.

A *one-round run* ρ (on $\mathcal{I}$ with source s) is an N-round which starts with the initial global configuration. Finally, a *multiple-round run* ρ^* (on $\mathcal{I}$ with source s) is a finite or infinite sequence $(\rho_i)_{i \in \kappa}$ of rounds along the same spanning tree such that ρ_0 is a one-round run and for every $i+1 \in \kappa$

- ρ_{i+1} starts with the last configuration of ρ_i, and
- if ρ_i is an M-round and $M_0 \subseteq M$ is the set of nodes at which A has sent the empty message during ρ_i, then ρ_{i+1} is an $(M - M_0)$-round.

Because M-rounds are only defined when $s \in M$, ρ^* is finite iff the root automaton sends the empty message during some round ρ_i, in which case ρ_i is the last round of ρ^*. The *output* produced during ρ^* is the union of the outputs of all rounds of ρ^*.

Because A is productive, for every choice of $\mathcal{I}$, s, and $\mathcal{T}$, there exists a multiple-round run of A on $(\mathcal{I}, s)$ along $\mathcal{T}$.

Example 4.1. Recall the simple Web automaton in Example 3.1. When run, this automaton outputs all 'interesting' nodes reachable from the source, without duplicates. In each round, the source automaton outputs precisely one node. Note that the output order depends on the choice of the spanning tree and on the order in which the various automata communicate during each round. □

5 Automata Computing Web Queries

Henceforth, Web automata are by default assumed to be productive.

On a given Web instance and source node, a Web automaton can make many different possible distributed runs. The order of communications within one round can be arbitrary. Moreover, the choice of the spanning tree is also arbitrary. Different runs might produce different output sets, as the next example shows.

Example 5.1. Consider the following program. (For readability, we use some syntactic sugar and write "*this_node*" and "*head*" instead of x and y.)

```
if $r_1 = 0$ then
  if $head = \bot$ then $send(this\_node)$
             else $send(head)$
  $r_1 := 1$
if $r_1 = 1$ then $send()$
```

On the 3-node instance $n_1 \longleftarrow s \longrightarrow n_2$, the output could be either $\{n_1\}$ or $\{n_2\}$, depending which of the two children of s gets its message first in the queue of s. By adding links $n_1 \longrightarrow n_2$ and $n_2 \longrightarrow n_1$, we obtain also dependence on the choice of the spanning tree. If the tree $s \longrightarrow n_1 \longrightarrow n_2$ is selected, the output is $\{n_2\}$, while if the tree $s \longrightarrow n_2 \longrightarrow n_1$ is selected, the output is $\{n_1\}$. $\square$

Example 5.2. On the other hand, again consider the automaton in Example 3.1. Although the *output order* can differ, the actual *output set* is always the same for every possible run. $\square$

We are thus led to define:

Definition 5.3. A Web automaton A is called *sound* if on every pair $(\mathcal{I}, s)$, every multiple-round run of A on $(\mathcal{I}, s)$ produces the same output.

If A is sound, we can speak of *the Web query computed by A* which maps every $(\mathcal{I}, s)$ to the output of any run of A on $(\mathcal{I}, s)$. $\square$

Unfortunately:

Theorem 5.4. *Soundness is undecidable.*

An indication of the querying power of Web automata is provided by the following result. Suppose that the vocabulary contains the distinguished ternary predicate symbol $\prec$. We call an instance $\mathcal{I}$ *ordered* if for each node n in $\mathcal{I}$, $\prec^{\mathcal{I},n}$ totally orders the children of n.

Theorem 5.5. *Any Web query restricted to ordered instances and computable in logarithmic space is computable by a Web automaton.*

An undesirable behavior of Web automata, even of sound ones, is that the user might have to wait many rounds before seeing any new output that he has not seen before. In the worst case, he might even wait only to learn later that there is no new output at all. Since each round takes only linear parallel time in the depth of the portion of the Web we are looking at, it would be particularly interesting to have the following behavior, which we call *continuity*:

Definition 5.6. A Web automaton A is called *continuous* if in any multiple-round run on any input, every round, except the last one, outputs at least one node that was not yet output. $\square$

Unfortunately:

Theorem 5.7. *Continuity is undecidable.*

Remark 5.8. Theorems 5.4 and 5.7 hold already for automata which test only one unary predicate symbol (in particular, which do not test the link predicate). Alternatively, both theorems remain true for automata which test the link predicate only. Moreover, undecidability is encountered even if one restricts attention to tree-like Web instances (where there is only one possible choice of a spanning tree). Finally, both theorems remain true for *finite* Web automata, that are, Web automata which cannot store nodes in their registers and therefore have only a finite number of different internal states. (Note that the automaton in Example 3.1 is finite in that sense). $\square$

The next result provides a class of Web queries computable by continuous Web automata, in terms of a fragment of first-order logic, which we call *at-most-at-least logic.* This fragment might seem artificial, but later we will see that it is associated to a natural subclass of Web automata.

Let $\alpha(x)$ be quantifier-free formula and let k be a natural number. An *α-at-most formula* is a formula of the form $(|\alpha| \leqslant k) \wedge \gamma_\alpha(s, x)$, where

- $(|\alpha| \leqslant k)$ abbreviates the first-order formula $\neg\exists^{>k} x\, \alpha(x)$, and

- γ_α is a boolean combination of formulas of the form $(\exists y_1 \in \alpha) \dots (\exists y_n \in \alpha)\beta$ where $\beta(s, x, \bar{y})$ is quantifier-free.

An *at-most-at-least formula* is a formula of the form $\alpha(x) \wedge \delta_\alpha(s, x)$ where δ_α is a boolean combination of α-at-most formulas and atomic formulas $\beta(s)$.

Example 5.9. With unary predicate symbols U and V and a binary predicate symbol W, an example of an at-most formula is

$$|U| \leqslant 42 \wedge \neg(V(x) \wedge (\exists y \in U) W(x, y))$$

An example of an at-most-at-least formula then is the conjunction of $U(x)$ with the negation of the above at-most formula, which can also be written as

$$U(x) \wedge \big(|U| \leqslant 42 \to (V(x) \wedge (\exists y \in U) W(x, y))\big)$$

□

Theorem 5.10. *Any Web query definable in at-most-at-least logic is computable by a continuous Web automaton.*

The converse direction of the above theorem does not hold. For instance, the query defined by the following formula is continuously computable, but the formula is not equivalent to any at-most-at-least formula:

$$\begin{aligned}&(\mathit{Green}(x) \wedge |\mathit{Green}| \geqslant 42)\\ &\qquad \vee (\mathit{Red}(x) \wedge |\mathit{Red}| \geqslant 10)\end{aligned}$$

Remark 5.11. There is an interesting variant of Theorem 5.10 which reads as follows. A *generalized at-most-at-least formula* is simply a boolean combination of at-most formulas and quantifier-free formulas $\beta(s, x)$. In particular, the at-most subformulas do not need to have the same α. We can show that any Web query definable by a generalized at-most-at-least formula is computable by a Web automaton *with discard action*, i.e., a Web automaton which can discard his queue in addition to performing update and send actions. □

The next observation gives an example of a query that is easily computable by a Web automaton, but not by a continuous one.

Proposition 5.12. *Let P be a binary predicate symbol. The Web query $\{x : P(s, x)\}$ is not computable by a continuous Web automaton.*

Indeed, to be continuous, the source automaton must start outputting already in the first round. However, by productivity, it can see only a constant number of nodes in each round. If the communication order is unfortunate, none of the nodes seen in the first round qualify for output.

Remark 5.13. An argument similar to the one just given shows that the at-most-at-least query $U(x) \wedge |U| \leqslant 2$, while computable by a continuous Web automaton, is not computable by a continuous Web automaton that can send messages of length at most 1. (Recall Remark 3.2.) □

6 DAF Automata

Call an automaton *link-free* if it does not test the link predicate. Link-freeness simplifies things considerably, as exemplified by the following 'flat-tree' property:

Proposition 6.1. *Let A be a sound and link-free Web automaton, and let Q be the Web query computed by A. Then for any input $(\mathcal{I}, s)$ we have $Q(\mathcal{I}, s) = Q(\mathrm{flatten}(\mathcal{I}, s), s)$, where $\mathrm{flatten}(\mathcal{I}, s)$ is the instance obtained from $\mathcal{I}$ by changing the link graph of $\mathcal{I}$ into a flat tree with root s and all other nodes children of s.*

For on flat trees sound Web automata can easily be shown to run in logarithmic space, the flat-tree property implies that Web queries computable by link-free Web automata are logarithmic-space computable. Whether this also holds in general is an open problem. (The flat-tree property itself certainly does not hold in general.)

Remark 6.2. Referring back to the Introduction, we point out (only half seriously though) that the Flat Tree Lemma provides some kind of a-posteriori justification of the way Internet supercomputing works, where indeed the standard mode of operation is that all computers that participate in the distributed computation report directly to a central source computer. □

Let $\Pi_{forward}$ be the following program:

```
if $head \neq \bot$ then
    $send(head)$
else
    $send()$
```

A Web automaton is called *DAF* (Decide And Forward) if its program has the form

```
if $first\_round$ then
    $\Pi$
else
    $\Pi_{forward}$
```

where *first_round* is a boolean register initialized with *true*, and Π is a program in which every send rule has the following form:

```
if $\varphi$ then $send(\bar{t})$; $first\_round := false$
```

where φ is a guard of the form $\varphi' \wedge \bigwedge_i (t_i \notin \{0,1\})$.

In other words, a DAF automaton makes all the crucial decisions in the first round, sends only nodes (no bits), and runs in all subsequent rounds (if any) merely as a forwarder that flushes the remaining contents of the communication queues to the output.

For example, the automaton of Example 4.1 is not DAF, but can easily be rewritten into an equivalent DAF form.

A Web automaton is called *monadic* if it does not test any predicate of arity $\geqslant 2$. In particular, a monadic automaton is link-free.

We have the following decidability results:

Theorem 6.3. *For monadic DAF automata, emptiness is decidable. (It is undecidable in general.)*

Theorem 6.4. *The problem of deciding whether a monadic DAF automaton is sound and continuous is decidable.*

Furthermore, we have the following characterization in terms of (monadic) at-most-at-least logic:

Theorem 6.5. *A Web query is computable by a continuous monadic DAF automaton if and only if it is definable in monadic at-most-at-least logic.*

Corollary 6.6. *Continuous DAF automata are strictly weaker than general continuous Web automata.*

Indeed, the query we used to show that the converse direction of Theorem 5.10 does not hold in general, is monadic.

7 Browser Stack Machines

Since the work by Abiteboul and Vianu [AV00] was one of the main inspiration for the present work, it seems only fitting to conclude this paper with a relationship between Web automata and A&V's browser machines [AV00]. In this section, we introduce a restricted variant of the browser machine and show that, in depth-bounded regions of the Web, machines of the restricted type can be simulated by Web automata. There are two main restrictions which we impose on browser machines:

- the work tape is replaced with a finite number of registers, and
- the browsing tape is organized like a stack, forcing a machine to explore the Web only by means of the three familiar surf actions of common Web browsers: 'follow this link', 'go back', and 'go forward'.

Browser Stack Machines. Let Υ and $x, \bar{r}$ be as in the definition of Web automata. (This time, x will denote the stack element which the cursor of the machine is currently pointing to.) A *guard* is a quantifier-free formula $\varphi(x, \bar{r})$ over $\Upsilon \cup \{0, 1\}$ with free$(\varphi) \subseteq \{x, \bar{r}\}$. A *program* Π is a finite set of rules of the form (`if` φ `then` *action*) where φ is a guard and *action* is an expression of the form *up*, *down*, *expand*, $(r_i := t)$ or *output*(t), with $t \in \{x, \bar{r}, 0, 1\}$.

A *browser stack machine (BSM)* is a triple $(\Upsilon, \bar{r}, \Pi)$ where Υ is a Web vocabulary, $\bar{r}$ is a tuple of distinct register variables, and Π is a program over Υ and $\bar{r}$.

Runs. Let $M = (\Upsilon, \bar{r}, \Pi)$ be a BSM and let $\mathcal{I}$ be an ordered Web instance (recall our convention prior to Theorem 5.5). In the following, the term *stack* refers to a finite sequence of 0's and nodes in $\mathcal{I}$. (An occurrence of 0 on a stack will serve as a separator between different segments of the stack.) Suppose that *st* is a stack of length k. A *cursor* c on *st* is a natural number between 1 and k.

A *configuration* of M is a quadruple $(st, c, \bar{a}, O)$ where *st* is a stack, c is a cursor on *st*, $\bar{a}$ is an ℓ-tuple consisting of bits and nodes in $\mathcal{I}$, and O is a set of nodes. Intuitively, O is the output produced so far.

The *successor configuration* $(st', c', \bar{a}', O')$ is defined in the obvious way; we only give some details for the actions *up*, *down*, and *expand*. Suppose that there is precisely one stack rule in Π which is enabled in $(st, c, \bar{a}, O)$. If the right-hand side of the rule is *up* or *down*, then $st' = st$ and c' is obtained from c as usual. If the right-hand side of the rule is *expand*, partition *st* into two st_1 and st_2 such that the length of st_1 is c. If c points to a node, say, n, and st_3 is a sequence of all child nodes of n ordered according to the order at n, then $st' = st_1 0 st_3$ and $c' = c$. Otherwise, $st' = st$ and $c' = c$.

Let s be a source node in $\mathcal{I}$. A *run* ρ of M (in $\mathcal{I}$ with source s) is a finite or infinite sequence $(C_i)_{i \in \kappa}$ of configurations of M such that $C_0 = (s, 1, \bar{0}, \varnothing)$ and for every $i + 1 \in \kappa$,

- C_{i+1} is a successor configuration of C_i, and
- if M attempts to move the cursor below the stack bottom in C_i, then C_{i+1} is the last configuration of $(C_i)_{i \in \kappa}$.

Note that ρ is uniquely determined by $(\mathcal{I}, s)$. If ρ is finite, we say that M *halts on* $(\mathcal{I}, s)$. In that case, the output component of the final configuration of ρ is the *output* of M on $(\mathcal{I}, s)$. M *computes* the Web query Q if M halts on every input $(\mathcal{I}, s)$ with output $Q(\mathcal{I}, s)$.

An indication of the querying power of BSMs is the following:

Proposition 7.1. *Any Web query definable by a generalized at-most-at-least formula (in the sense of Remark 5.11) is computable by a BSM.*

Depth-Bounded BSMs. Let d be a natural number. A BSM M is called *d-bounded* if it maintains a counter of the number of separators between the stack bottom and the current cursor position. Whenever this counter equals d, M ignores all expand actions.

We can show:

Theorem 7.2. *Any Web query computable by a depth-bounded BSM is computable by a Web automaton.*

References

[AV00] S. Abiteboul and V. Vianu. Queries and computation on the web. *Theoretical Computer Science*, 239(2):231–255, May 2000.

[AW98] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.

[EF95] H. D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.

[FK98] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[Fos00] I. Foster. Internet computing and the emerging grid. *Nature*, December 2000.

[GHR00] N. Gupta, J.R. Haritsa, and M. Ramanath. Distributed query processing on the Web. In *Proceedings of 16th International Conference on Data Engineering*, page 84. IEEE Computer Society, 2000.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Imm87] N. Immerman. Languages That Capture Complexity Classes. *SIAM Journal of Computing*, 16(4):760–778, 1987.

[KF94] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, November 1994.

[KWA$^+$01] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Lebofsky. SETIHOME—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[MM98] A. Mendelzon and T. Milo. Formal models of web queries. *Information Systems*, 23(8):615–637, 1998.

[NSV01] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *Proceedings of 26th International Symposium on Mathematical Foundations of Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 560–572. Springer, August 2001.

[Saz] V. Sazonov. Using agents for concurrent querying of web-like databases via a hyper-set-theoretic approach. To appear in Proceedings of 4th International Conference on Perspectives of System Informatics, July 2001, Novosibirsk, Russia.

[Spi00] M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. PhD thesis, RWTH Aachen, 2000.

[Tan96] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 3rd edition, 1996.

Appendix

We sketch the proofs of our main results.

A.1 Productivity

Proposition A.3. *Let A be a Web automaton with (at most) ℓ registers. A is productive iff A is 3^ℓ-productive.*

Proof of Theorem 3.4. First, consider the problem of deciding k-productivity. Containment is proved by reduction to the finite satisfiability problem for existential transitive-closure logic, FIN-SAT(E+TC), which is in PSPACE if we focus on formulas over relational vocabularies [Spi00]. For every Web automaton A and every natural number k, one can construct (in polynomial time) an (E+TC) sentence $\varphi_{A,k}$ which has a finite model iff A is not k-productive.

Hardness is proved by reduction from a restriction of FIN-SAT(E+TC). We call a formula of the form $[\mathrm{TC}_{\bar{x},\bar{x}'}\varphi](\bar{t},\bar{t}')$ *simple* if φ is quantifier-free and $\bar{t}=\bar{0}$ and $\bar{t}'=\bar{1}$. The problem of deciding whether a given simple TC formula has a (finite) model is already PSPACE-hard [Spi00].

Consider a simple TC sentence $\psi = [\mathrm{TC}_{\bar{x},\bar{x}'}\varphi](\bar{0},\bar{1})$. Suppose that $\bar{x}$ (and thus $\bar{x}'$) consists of ℓ variables. Below, we outline the program of a Web automaton which is k-productive iff ψ is not satisfiable. It is assumed that initially $\bar{x}=\bar{0}$ and $i=1$.

```
if $\bar{x} \neq \bar{1}$ then
    $send(0)$
    if $i \leq \ell$ then
        $x'_i := head$
        $i := i + 1$
    else
        $i := 1$
        if $\varphi(\bar{x}, \bar{x}')$ then $\bar{x} := \bar{x}'$
```

Now consider the problem of deciding productivity. Hardness is implied by the same reduction which shows hardness of deciding k-productivity. It remains to prove containment in PSPACE. Given a Web automaton A with ℓ registers we can set $k = 3^\ell$, store k in space polynomial in the size of A, and then run our polynomial-space algorithm for deciding k-productivity. This procedure is still in polynomial space and, by Proposition A.3, decides productivity of A. □

A.2 Undecidability Results

Proof of Theorem 5.4. The proof is by reduction from the emptiness problem for deterministic one-way two-head automata (2-DFAs). It suffices to consider *simple 2-DFAs*, that are, 2-DFAs whose input alphabet is $\{0,1\}$ and whose program ensures that every computation progresses in two distinguished phases. During the first phase, a simple 2-DFA M uses its first input head to scan an initial segment of the input tape. The second input head remains idle. After each computation step, M may or may not switch to the second phase, depending on its current configuration. If and when M switches to the second phase, the first input head is placed somewhere on the tape, while the second input head is still on the first tape cell.

During the second phase, M can do whatever 2-DFAs are entitled to do, with the restriction that, in every computation step, M must move both input heads, each one to the next tape cell. A computation of M stops if the input is accepted or if the first input head reaches the end of the input tape. One can show that for simple 2-DFAs the emptiness problem is undecidable (by reduction from the word problem for Turing machines).

Let M be a simple 2-DFA. We construct a Web automaton A_M over $(\{L\}, \varnothing)$ such that

- if $L(M) = \varnothing$, then A_M continuously computes Q_{true} and
- if A_M is sound (or, alternatively, continuous), then $L(M) = \varnothing$.

This will reduce the emptiness problem for simple 2-DFAs to the problem of deciding soundness (or, continuity).

To the construction of A_M. In the first round, A_M performs the following two tasks in parallel. First, it checks whether it is executed along a spanning tree which has the form of a path. Second, it pretends that the first test was successful, views the spanning tree (which is now assumed to be a path) as an input tape (where link self loops represent set input bits), and simulates the first phase of M on that input tape. If the first test fails, the source instance of A_M switches to a 'forwarding' mode, which means that in every subsequent round it just outputs all nodes (reachable from the source node). The same happens if during the simulation of M the first input head reaches the end of the (virtual) input tape.

If the source automaton survives the first round without switching to forwarding mode, then, in all subsequent rounds, A_M simulates the second phase of M and, in parallel, outputs all nodes. Except if the source automaton discovers during the simulation that M accepts. In that case, the source automaton switches to a 'spoiling' mode, which means that it stops outputting nodes and instead sends some dummy messages. □

Theorem 5.7 follows immediately from the reduction in the above proof.

A.3 Main Decidability Result

We briefly outline the proof of Theorem 6.4. Theorem 6.3 is a consequence of intermediate results.

In the following, fix a DAF automaton A. We assume that A is k-productive. A Web instance $\mathcal{I}$ is called *tree-like* if the link graph of $\mathcal{I}$ is a tree (where each leaf is reachable from the root). By a run of A on a tree-like $\mathcal{I}$ we mean a run of A on $(\mathcal{I}, r)$ where r is root of $\mathcal{I}$.

Lemma A.4. *Suppose that A is monadic. There exists a (computable) constant $c_{A,k}$ such that for every one-round run ρ of A there exists a one-round run ρ' of A on a tree-like Web instance of size at most $c_{A,k}$ such that the message sent by the source automaton during ρ' is identical with the message sent by the source automaton during ρ.*

We conclude:

Theorem A.5. *For monadic Web automata the first-round emptiness problem is decidable.*

Proof of Theorem 6.3. Verify that the identity is a reduction from the emptiness problem for DAF automata to the first-round emptiness problem for Web automata. The theorem then follows from Theorem A.5. □

The main idea in the proof of Theorem 6.4 is to reduce the problem of deciding soundness to the problem of deciding soundness on flat trees. A tree-like $\mathcal{I}$ is called *flat* if the link depth of $\mathcal{I}$, measured from the root, is at most 1.

Definition A.6. A is *flat-tree sound* if for every flat $\mathcal{I}$ appropriate for A and for any two multiple-round runs ρ_1^* and ρ_2^* of A on $\mathcal{I}$, $\mathrm{out}(\rho_1^*) = \mathrm{out}(\rho_2^*)$.

Recall the definition of $\mathrm{flatten}(\mathcal{I}, s)$ in Proposition 6.1. A is *flat-invariant* if A is flat-tree sound and for every tree-like $\mathcal{I}$ appropriate for A and for every multiple-round run ρ^* of A on $\mathcal{I}$, $\text{out}(\rho^*) = \text{out}(A, \text{flatten}(\mathcal{I}, r))$, where r is the root of $\mathcal{I}$. □

Lemma A.7. *Flat-tree soundness is a decidable property of DAF automata.*

Lemma A.8. *Suppose that A is monadic. A is sound iff A is flat-invariant.*

The proof of the last lemma is based on Proposition 6.1. The rest of the construction concerns a procedure for deciding flat-invariance.

Alpha Nodes. Let $\alpha(x)$ be a quantifier-free formula such that for every tree-like $\mathcal{I}$ appropriate for A and for every leaf node n in $\mathcal{I}$, A at n sends a non-empty message during a one-round run of A on $\mathcal{I}$ iff $\mathcal{I} \models \alpha[n]$. A node n in some $\mathcal{I}$ appropriate for A is called *α-node* if $\mathcal{I} \models \alpha[n]$.

Alpha-Sending Automata. We call A *α-sending* if during every one-round run of A, every non-empty message sent by A at a non-source node contains only pairwise distinct α-nodes.

Lemma A.9. *If A is α-sending, then A is continuous.*

Lemma A.10. *It is decidable whether a given Web automaton is α-sending.*

Alpha-Outputting Automata. We call A *α-outputting* if for every $(\mathcal{I}, s)$ appropriate for A and for every multiple-round run ρ^* of A on $(\mathcal{I}, s)$, if N_α is the set of α-nodes in $\mathcal{I}$, then $\text{out}(\rho^*) \subseteq \{s\} \cup N_\alpha$.

Lemma A.11. *If A is monadic and sound, then A is α-outputting.*

Lemma A.12. *It is decidable whether a given α-sending DAF automaton is α-outputting.*

A tree-like $\mathcal{I}$ is called *sparse* if there are at most k non-root α-nodes in $\mathcal{I}$.

Definition A.13. A is *sparse-tree sound* if for every sparse $\mathcal{I}$ appropriate for A and for any two multiple-round runs ρ_1^* and ρ_2^* of A on $\mathcal{I}$, $\text{out}(\rho_1^*) = \text{out}(\rho_2^*)$. □

Lemma A.14. *Sparse-tree soundness is a decidable property of α-sending DAF automata.*

The next lemma is central to our construction. Its proof is based on Lemma A.4.

Lemma A.15. *Flat-invariance is a decidable property of flat- and sparse-tree sound, α-sending and -outputting, monadic DAF automata.*

We are now in the position to prove our main decidability result.

Proof of Theorem 6.4. Consider a monadic DAF automaton A. Call A *bounded* if for every flat $\mathcal{I}$ appropriate for A with precisely k non-root α-nodes and for every one-round run ρ of A on $\mathcal{I}$, ρ is terminating, i.e., the source automaton sends the empty message during ρ. Otherwise, call A *unbounded.*

First determine whether A is bounded or unbounded (simply by testing all non-isomorphic small flat trees). Suppose that A is unbounded. One can show that, if A is sound and continuous, then A must be α-sending. Check whether A is α-sending (see Corollary A.10). If the test fails, reject A. Otherwise, check whether A is α-outputting (see Lemma A.12). If this test fails, reject A (because A is not sound according to Lemma A.11). Otherwise, check whether A is flat- and sparse-tree sound (see Lemmata A.7 and A.14). If one of the two tests fails, reject A (clearly, A cannot be sound in that case). Otherwise, check whether A is flat-invariance (see Lemma A.15). If this test fails, reject A (because A is not sound according to Lemma

A.8). Otherwise, accept A, for it is sound and continuous due to Lemmata A.8 and A.9.

Now suppose that A is bounded. Note that A may not be α-sending in this case. An analysis of bounded Web automata (as complex as for α-sending automata) leads to a decision procedure similar to the one outlined above. $\square$

A.4 Computational Power

This subsection concerns the proofs of Theorems 5.5, 5.10, and 6.5.

Lemma A.16. *There exists a Web automaton A_{enum} such that for every ordered $\mathcal{I}$ and for every source node s in $\mathcal{I}$, every multiple-round run $(\rho_i)_i$ of A on $(\mathcal{I}, s)$ satisfies the following three conditions:*

1. *$(\rho_i)_i$ is infinite.*
2. *For every index i, $\mathrm{out}(\rho_i)$ is either empty or a singleton set.*
3. *There exists an enumeration e of all nodes in $\mathrm{Reach}(\mathcal{I}, s)$ such that, if $(o_j)_j$ is obtained from $(\mathrm{out}(\rho_i))_i$ by removing all empty sets, then $(o_j)_j$ can be seen as an infinite repetition of e.*

Proof of Theorem 5.5. Let $\varphi(x_1, \ldots, x_k)$ be a formula of deterministic transitive-closure logic [EF95]. We construct a Web automaton A_φ which, on input $(\mathcal{I}, s)$, enumerates $\{\bar{a} : \mathrm{Reach}(\mathcal{I}, s) \models \varphi[\bar{a}]\}$ in the following sense. In every round, A_φ at s sends either a 'wait' message or a message $(a_1, \ldots, a_k)$ satisfying φ. Eventually, all messages satisfying φ are sent by A_φ at s. The theorem is then implied by a well-known result from descriptive complexity theory [Imm87, EF95].

The construction of A_φ is by induction on φ and uses A_{enum} in Lemma A.16. For instance, if $\varphi(\bar{x}) = R(\bar{x})$, then A_φ simulates A_{enum}, turns the repetitive enumeration of all nodes into an enumeration of all k-tuples of nodes, and checks whether $R(\bar{x})$ holds for each k-tuple. $\square$

In the following, $\lambda_\alpha(s, x)$ denotes an *α-at-most literal*, i.e., a formula of the form $(|\alpha| \leq k) \wedge \gamma_\alpha(s, x)$ or the form $(|\alpha| \leq k) \rightarrow \gamma_\alpha(s, x)$.

Proposition A.17. *Any conjunction of α-at-most literals (in the variables s and x) is equivalent to an α-at-most literal. The same holds true for disjunctions of α-at-most literals.*

Lemma A.18. *Let $\varphi(s, x)$ be a formula of the form $\alpha(x) \wedge \lambda_\alpha(s, x)$. The Web query Q_φ is computable by a continuous Web automaton.*

Crux. Suppose that λ_α is a positive literal, say, $\lambda_\alpha = (|\alpha| \leq k) \wedge \gamma_\alpha(s, x)$. We describe briefly a continuous Web automaton A_φ that computes Q_φ. A_φ is $(k+1)$-productive and sends messages of length $\leq k+1$. If A_φ is executed at a node different from the source node, A_φ forwards in each round as many as possible (but at most $k+1$) nodes satisfying α to its parent automaton. If A_φ is executed at the source node, A_φ attempts to see $(k+1)$ nodes satisfying α. If it succeeds, it sends the empty message, thereby terminating the computation. Otherwise, it knows all nodes (reachable from the source node) that satisfy α. In particular, there are at most k such nodes. For each such node n, the source automaton checks whether $\gamma_\alpha(s, n)$ holds and, if successful, outputs n.

Now suppose that λ_α is a negative literal, say, $\lambda_\alpha = (|\alpha| \leq k) \rightarrow \gamma_\alpha(s, x)$. Modify A_φ so that, if the source automaton discovers that there are at least $(k+1)$ nodes satisfying α, then, instead of sending the empty message, it outputs all nodes in its queue, plus the source node if the source node satisfies α. $\square$

Color Types. Let x be a variable and let Υ be a vocabulary. A *color type in x* over

Υ is a maximal consistent set of atomic and negated atomic formulas in x over Υ. In the following, $c(x)$ denotes a color type in x.

Observe that every quantifier-free formula $\alpha(x)$ is equivalent to a disjunction of color types in x.

Proof of Theorem 5.10. Let $\varphi(s,x)$ be an at-most-at-least formula. We construct a continuous Web automaton A_φ that computes Q_φ. Suppose that $\varphi(s,x) = \alpha(x) \wedge \delta_\alpha(s,x)$. Using Proposition A.17 one can show that δ_α is equivalent to a formula of the form

$$\bigvee_i \big(c_i(s) \wedge \lambda_{\alpha,i}(s,x)\big) \qquad (1)$$

where each $c_i(s)$ is a color type in s over the vocabulary of δ_α and $c_i \equiv c_j$ iff $i = j$. According to Lemma A.18, there exists for each index i in formula (1) a continuous Web automaton that computes $Q_{\alpha\wedge\lambda_{\alpha,i}}$. It is now an easy exercise to combine these automata to a continuous Web automaton A_φ computing Q_φ. $\square$

Remark A.19. The proofs of both Lemma A.18 and Theorem 5.10 can be arranged so that the constructed Web automata are link-free and DAF. $\square$

Proof of Theorem 6.5. Let Q be a Web query. Suppose that Q is definable by a monadic at-most-at-least formula. According to Theorem 5.10, Q is computable by a continuous Web automaton. By Remark A.19, this automaton is monadic and DAF.

Now suppose that Q is computable by a continuous monadic DAF automata A. Furthermore, suppose that A is $(k+1)$-productive. Let s and x be two variables and let $c_1(s),\dots,c_l(s)$ be an enumeration of all color types in s over the vocabulary of A (up to isomorphism). Clearly, $\bigvee_i c_i(s) \equiv (s = s)$. We are going to construct (i) a quantifier-free formula $\alpha(x)$ and (ii) for each $i \in \{1,\dots,l\}$ an α-at-most literal $\lambda_{\alpha,i}(s,x)$ such that the formula

$$\alpha(x) \wedge \bigvee_i \big(c_i(s) \wedge \lambda_{\alpha,i}(s,x)\big)$$

defines Q.

Define $\alpha(x)$ as in the previous subsection (see below Lemma A.8). Intuitively, α specifies those (colorings of) leaf nodes which A at the source node can possibly see (recall Lemma A.8).

The definition of $\lambda_{\alpha,i}(s,x)$ is based on various tests revealing the behavior of A when executed at c_i-colored source nodes. Choose pairwise distinct $c'_1(x),\dots,c'_m(x) \in \{c_1(x),\dots,c_l(x)\}$ so that $\alpha(x) \equiv \bigvee_j c'_j(x)$. Consider a flat $\mathcal{I}$ appropriate for A such that (i) the root node r of $\mathcal{I}$ satisfies $c_i(s)$ and (ii) for each $j \in \{1,\dots,m\}$ there are at least $(k+1)$ leaf nodes satisfying $c'_j(x)$. We are going to execute A at r (in $\mathcal{I}$) on various queues consisting of α-nodes.

By a *coordinate* $\bar{k}$ we mean a tuple $(k_1,\dots,k_m)$ such that $k_1,\dots,k_m \le (k+1)$. Let $\bar{k}$ be a coordinate. A *$\bar{k}$-queue* is a sequence of leaf nodes in $\mathcal{I}$ such that (i) the length of the sequence is $\sum_{j=1}^m k_j$ and (ii) for each $j \in \{1,\dots,m\}$ the sequence contains precisely k_j pairwise distinct nodes satisfying $c'_j(x)$. Let q be a $\bar{k}$-queue. We say that A at r *accepts* q if the first message sent by A at r on q is not empty (i.e., contains a node).

Verify that for any two $\bar{k}$-queues q and q', A at r accepts q iff A at r accepts q'. Hence, we can define an m-dimensional table T_i as follows: at coordinate $\bar{k}$, T_i contains "accept" if A at r accepts a(ny) $\bar{k}$-queue; otherwise it contains "reject". By D_i we denote the diagonal plane of T given by all coordinates satisfying $\sum_{j=1}^m k_j = (k+1)$. One can show that D_i has either only accept entries or only reject entries.

Next observe that the definition of T_i does not depend on the choice of $\mathcal{I}$. We obtain the same table for any flat $\mathcal{I}$ whose root node satisfies $c_i(s)$ and which contains enough leaf nodes satisfying $c'_j(x)$ (for each j). This shows that the decision of whether A at a c_i-colored source node is going to output or not is entirely determined by the entries on and below the diagonal plane D_i, that are, all entries at coordinates satisfying $\sum_j k_j \leq (k+1)$.

Suppose that D_i has only reject entries. Let S be the set of coordinates which satisfy $\sum_j k_j \leq k$ and where T_i has an accept entry. Define $\lambda_{\alpha,i}(s,x)$ to be

$$(|\alpha| \leq k) \wedge \bigvee_{\bar{k} \in S} \left(\gamma'_{\bar{k}} \wedge \gamma''_{\bar{k}}(s,x)\right) \qquad (2)$$

where $\gamma'_{\bar{k}}$ and $\gamma''_{\bar{k}}$ are as follows. If $c_i(x)$ does not occur among $c'_1(x), \ldots, c'_m(x)$, set $\gamma'_{\bar{k}} = \bigwedge_j (|c'_j| = k_j)$. Otherwise, suppose that $c_i(x) = c'_1(x)$. Set $\gamma'_{\bar{k}} = (|c'_1| = k_1 + 1) \wedge \bigwedge_{j>1}(|c'_j| = k_j)$. $\gamma''_{\bar{k}}$ specifies those (colorings of) nodes in an $\bar{k}$-queue which are output, and also whether s is output or not. This can be determined by testing A.

Now suppose that D_i has only accept entries. In that case, replace the first conjunction symbol in formula (2) with an implication symbol. □

A.5 Browser Stack Machines

Proposition A.20. *Q_{true} is BSM computable.*

Crux. A BSM M_{reach} computing Q_{true} uses its node stack to perform a depth-first search. In order to avoid running into an infinite loop, M_{reach} expands the top node of the stack only if that node does not occur elsewhere on the stack. M_{reach} can check this by storing the top node in one of its registers and letting the cursor scan all nodes currently on the stack. Whenever a node is expanded, it is also output. □

Proposition 7.1 now follows from an easy modification of M_{reach} in the above proof.

Proof of Theorem 7.2. Due to Theorem 5.5, it suffices to show that any depth-bounded BSM can be simulated by a logarithmic-space bounded Turing machine (with separate input and output tapes). Consider a d-bounded BSM M. We sketch a logarithmic-space bounded Turing machine T_M such that for every pair $(\mathcal{I}, s)$ appropriate for M, T_M on an encoding of $(\mathcal{I}, s)$ simulates M on $(\mathcal{I}, s)$.

Let C be a configuration of M on $(\mathcal{I}, s)$, say, $C = (st, c, \bar{a}, O)$. We call $(st, c, \bar{a})$ a *reduced configuration.* (Note that, because T_M has a separate output tape, it does not need to store the output set O.) We first observe that (a representation of) any reduced configuration can be stored in space logarithmic in the size of $\mathcal{I}$. This clearly holds for the contents $\bar{a}$ of the registers of M, for each node requires only logarithmic space.

Consider a stack st. By the *i-th segment* of st we mean the segment that starts with the node following the $(i-1)$-th separator and ends with the i-th separator. For example, the first segment of any stack consists of the source node and the first separator. To represent st we employ d logarithmic-space registers, called *stack registers.* Each stack register either holds a node or is undefined. For every $i \in \{1, \ldots, d\}$, the i-th stack register holds the last node of the i-th segment of st, i.e., the node before the i-th separator. This node was expanded when the $(i+1)$-th segment was placed on the node stack.

To represent the cursor c of M we employ a counter ranging in $\{1, \ldots, d+1\}$ and a logarithmic-space register, called *cursor register.* Intuitively, the cursor register holds the node currently read by the cursor. It is undefined iff the cursor is currently placed on a separator. The counter specifies in which segment the cursor is currently roaming. (Verify that, because no node occurs twice in the same segment, the counter and the cursor register together uniquely determine the position of the cursor.)

Given this encoding of (reduced) configurations of M, T_M can simulate a transition from one configuration to a successor configuration in logarithmic space. $\square$