SIAM J. COMPUT.
Vol. 21, No. 4, pp. 755–780, August 1992

AN OPTIMAL PARALLEL ALGORITHM FOR FORMULA EVALUATION*

S. BUSS[†], S. COOK[‡], A. GUPTA[§], AND V. RAMACHANDRAN[¶]

Abstract. A new approach to Buss's $\mathbf{NC}^1$ algorithm [*Proc. 19th ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, 1987, pp. 123–131] for evaluation of Boolean formulas is presented. This problem is shown to be complete for $\mathbf{NC}^1$ over $\mathbf{AC}^0$ reductions. This approach is then used to solve the more general problem of evaluating arithmetic formulas by using arithmetic circuits.

Key words. formula evaluation, Boolean formulas, parallel algorithms, circuit complexity, log-time hierarchy

AMS(MOS) subject classification. 68Q

1. Introduction. In this paper we consider the parallel complexity of the Boolean and arithmetic formula-value problems. The Boolean formula-value problem is that of determining the truth value of an infix Boolean formula (with connectives $\{\wedge, \vee, \neg\}$) given the truth assignments of the variables in the formula. Since it is easy to substitute values for the variables, we can reduce this problem to that of solving the Boolean sentence-value problem (BSVP), i.e., the Boolean formula-value problem restricted to the case in which the formula contains constants and operators, but no variables. The goal is to obtain a bounded fan-in Boolean circuit of small depth that solves the BSVP for all inputs of a given size. We assume that each gate takes unit time for its computation and that there is no propagation delay along wires. This is the standard circuit model (see, e.g., [sa76], [co85], [kr90]). In this model the time taken by a circuit to compute the values of its outputs when given values to its inputs is equal to the depth of the circuit. Hence, a circuit of small depth corresponds to a computation that can be performed quickly in parallel.

A natural extension to the Boolean formula-value problem is the problem of evaluating an arithmetic formula over a more general algebra. In this paper we consider this problem over semi-rings, rings, and fields. The problem is basically the same as BSVP—given an arithmetic formula over an algebra and the value of each variable in the formula, determine the value of the formula. We use the arithmetic-Boolean circuits of von zur Gathen [jg86] as our model, and we use the corresponding arithmetic complexity theory. Once again, the goal is to obtain a circuit of small depth that solves this problem for all inputs of a given depth. We assume that each arithmetic gate has unit delay, so that the time required by the circuit to perform a computation is equal to its depth.

An additional property that we desire in the family of circuits we construct is that it be uniform, i.e., that a description of the circuit for evaluating formulas of size n can be obtained easily when the value of n is known; the family is logspace uniform if the

*Received by the editors April 10, 1989; accepted for publication (in revised form) July 3, 1990.

[†]Department of Mathematics, University of California, San Diego, La Jolla, California 92093. This research was partially supported by National Science Foundation grants DMS85-11465 and DMS87-01828.

[‡]Department of Computer Science, University of Toronto, Toronto, Canada, M5S 1A4. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

[§]Department of Computer Science, University of Toronto, Toronto, Canada, M5S 1A4. Present address, School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada, V5A 1S6. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

[¶]Department of Computer Science, University of Texas at Austin, Austin, Texas 78712. This work was supported in part by National Science Foundation grant ECS 8404866, Semiconductor Research Corporation grant RSCH 84-06-049-6, and Joint Services Electronics Program grant N00014-84-C-0149 while the author was with the University of Illinois, Urbana.

description of the nth circuit can be provided by a deterministic Turing machine operating in space $O(\log n)$. The class $\mathbf{NC}^k$ for $k \geq 2$ is the class of problems that have a logspace-uniform family of circuits of depth $O(\log^k n)$ and polynomial size, where n is the size of the input; for $\mathbf{NC}^1$ a stronger notion of uniformity is usually used [ru81]. The class **NC** is the class of problems that have a logspace-uniform family of circuits of polylog depth and polynomial size; this class is generally considered to characterize the class of problems with feasible parallel algorithms. Let **P** be the class of problems solvable in sequential polynomial time. An important open question in parallel complexity theory is whether **NC** equals **P** or whether $\mathbf{NC}^1$ equals **P**. For more on parallel circuit complexity see, e.g., [co85], [kr90], [ru81].

Simple fan-in arguments show that any circuit for formula evaluation must have depth at least logarithmic in the size of the formula. Early work on BSVP was done by Spira [sp71], who showed that any sentence of size n can be restructured into a formula of depth $O(\log n)$ and size $O(n^2)$. Brent [br74] used a restructured circuit of logarithmic depth and linear size to evaluate a given arithmetic formula. These results gave hope of obtaining a logarithmic-depth circuit for formula evaluation by finding a logarithmic-depth circuit for performing the appropriate restructuring. However, direct implementation of these algorithms seems to require $\Omega(\log^2 n)$ depth for the restructuring. This result placed BSVP in $\mathbf{NC}^2$.

The BSVP can be shown to be in $\mathbf{NC}^2$ through the use of other techniques. Lynch [ly77] showed that parenthesised context-free languages can be recognized in deterministic log space (LOGSPACE). Since the set of true Boolean sentences is an instance of these languages, this immediately implied the same space bound for BSVP. The result of Borodin [bo77], that LOGSPACE $\subseteq \mathbf{NC}^2$, once again placed this problem in $\mathbf{NC}^2$. The logarithmic-time tree-contraction algorithm of Miller and Reif [mr85] for arithmetic expression evaluation on a PRAM again translates into an $\mathbf{NC}^2$ algorithm on arithmetic circuits.

The first sub-$\mathbf{NC}^2$ algorithm for BSVP was devised by Cook and Gupta [gu85] and independently by Ramachandran [ra86]. Their circuit family for the problem was log space uniform and had a depth of $O(\log n \log\log n)$, and this gave new hope that the problem had an $\mathbf{NC}^1$ algorithm. Cook and Gupta also showed that parenthesis context-free grammars can be recognized in depth $O(\log n \log\log n)$, while Ramachandran showed that arithmetic formulas over semi-rings can be evaluated within the same time bound.

Recently, Buss [bu87] devised an alternating log-time algorithm for both BSVP and the recognition problem for parenthesis context-free grammars. Since alternating log time is equivalent to $\mathbf{NC}^1$ [ru81] under a strong notion of uniformity, this finally settled the question of whether BSVP is in $\mathbf{NC}^1$. Buss's algorithm was based on converting the sentence into PLOF form (post-fix longer operand first) and then playing a two-person game on it. The game simulated the evaluation of the sentence and could be played in log-time on an alternating Turing machine. Buss also showed that his result is optimal in a very strong sense—he showed that BSVP is complete for alternating log-time under reductions from any level of the log-time hierarchy.

Dymond [dy88] extended Buss's result for parenthesis grammars to show that all input-driven languages can be recognized in $\mathbf{NC}^1$. His technique generalizes the game described by Buss.

Very recently, Muller and Preparata [mp88] devised log-depth circuits to solve formula evaluation for semi-rings. Their approach is based on using a universal evaluator to evaluate an infix formula where, for each operator, the longer operand occurs before the shorter.

There are a number of reasons why the formula-value problem is interesting. The BSVP is the analogue of the circuit-value problem for which each gate has fan-out 1. The circuit-value problem is log space complete for P and hence is not in **NC** unless **P** equals **NC**. The BSVP, on the other hand, is clearly in **NC** and is, therefore, a natural candidate for an $\mathbf{NC}^1$-complete problem. Also, propositional formulas are fundamental concepts in logic, and the complexity of evaluating them is of interest.

In this paper, we present a simple $\mathbf{NC}^1$ algorithm for BSVP that incorporates Buss's original ideas into a two-person pebbling game similar to that introduced by Dymond and Tompa [dt85]. This algorithm is designed to give insight into the mechanism of Buss's algorithm. We show that our result is optimal by proving that the problem is complete for $\mathbf{NC}^1$ under $\mathbf{AC}^0$ reductions. We then proceed to use our evaluation technique to place the general arithmetic-formula evaluation problem over rings, fields, and semi-rings in arithmetic $\mathbf{NC}^1$.

This paper is organized as follows. The relevant background is given in §2. In §3 we describe an $\mathbf{NC}^1$ algorithm that translates Boolean sentences into PLOF sentences. In §4 an $\mathbf{NC}^1$ algorithm for the PLOF sentence-value problem is given. This finishes the proof that BSVP is in $\mathbf{NC}^1$. Some completeness results for BSVP are given in §5. In §6 we generalize the technique of §4 to obtain an arithmetic $\mathbf{NC}^1$ algorithm for arithmetic-formula evaluation (over rings, fields, and semi-rings).

2. Background.

2.1. Boolean circuit complexity.

All unreferenced material in this section is from [co85], and we refer the reader to that paper for a more in-depth discussion of Boolean circuit complexity.

DEFINITION. A *Boolean circuit* α on n inputs and m outputs is a finite directed acyclic graph with each node labeled from $\{x_1, \cdots, x_n, 0, 1, \neg, \vee, \wedge\}$. Nodes labeled x_i are *input nodes* and have indegree 0. Nodes with indegree 1 are labeled $\neg$, and those with indegree 2 are labeled either $\vee$ or $\wedge$, where each edge into the node is associated with one argument of the function corresponding to the label. There is a sequence of $m \geq 1$ nodes in α designated as *output nodes*. In practice, the nodes of α are called *gates*.

DEFINITION. For a circuit α, the *complexity* of α, designated $c(\alpha)$, is the number of nodes in α. The *depth* of α, designated $d(\alpha)$, is the length of the longest path from some input node to some output node.

We also assign to each gate in our Boolean circuits a gate number. We assume that in a given circuit α, each gate has a unique gate number and all gate numbers are between 0 and $c(\alpha)^{O(1)}$ (i.e., their binary encoding is $O(\log c(\alpha))$). Furthermore, we assume that all gates in a Boolean circuit are on a path from some input to an output. When the inputs are assigned values from $\{0, 1\}$, each output takes on a unique value. A circuit α on n inputs and m outputs computes a function $f : \{0,1\}^n \to \{0,1\}^m$ in the obvious way. We are interested in computing more general functions, namely those of the form $f : \{0,1\}^* \to \{0,1\}^*$. We need circuit families for this.

DEFINITION. A *circuit family* $\langle \alpha_n \rangle$ is a sequence of Boolean circuits such that the nth circuit in the family has n inputs and $h(n)$ outputs, where $h(n) = n^{O(1)}$.

Notice that arbitrary circuit families are very powerful (they can even recognize nonrecursive languages). Therefore, we restrict ourselves to *uniform circuit families*. The strength of the Turing machines used to generate the circuits determines the uniformity condition on the circuit family. We note that all of our Turing machines are assumed to be multi-tape.

DEFINITION. Let α be a Boolean circuit. Let g be a gate in α and $\rho \in \{l, r\}^*$. Then $g(\rho)$ is the gate reached when ρ is followed (as a path) toward the inputs of α by starting at g. For example, $g(l)$ is g's left input. We make the convention that if g is an input gate, then $g(l) = g(r) = g$.

DEFINITION [ru81]. For a circuit family $\langle \alpha_n \rangle$, the *extended-connection language* L_{EC} consists of 4-tuples $\langle \bar{n}, g, \rho, y \rangle$, where $\bar{n} \in \{0,1\}^*$ ($\bar{n}$ is n in binary), $g \in \{0,1\}^*$ (g is a gate number), $y \in \{x_1, \cdots, x_n, 0, 1, \neg, \vee, \wedge\} \cup \{0,1\}^*$, and $|\rho| \leq \log c(\alpha_n)$ such that if $\rho = \epsilon$ then y is the label of the gate numbered g; otherwise, y is the gate number of $g(\rho)$. $\langle \alpha_n \rangle$ is U_E *uniform* if there is a deterministic linear-time Turing machine recognizing L_{EC}.

Here L_{EC} encodes local connection information in α_n, that is, connections that are within distance $\log c(\alpha_n)$.

Note. The time bound in Ruzzo's original definition of U_E uniformity is given as $O(\log c(\alpha_n))$. However, since the length of the input to the Turing machine is also $O(\log c(\alpha_n))$, our definition is equivalent—we prefer to use the size of the input to the machine in the definition.

DEFINITION. For all $k > 0$, define $\mathbf{NC}^k$ as the class of functions computable by a U_E-uniform circuit family $\langle \alpha_n \rangle$ such that $c(\alpha_n) = n^{O(1)}$ and $d(\alpha_n) = O(\log^k n)$. $\mathbf{NC} = \bigcup_{k>0} \mathbf{NC}^k$.

We use U_E uniformity in our definition of $\mathbf{NC}$ instead of the more common U_{E^*} uniformity. Ruzzo shows that $\mathbf{NC}^k$ ($k \geq 1$) is the same under both definitions. The advantage of using U_E uniformity is that the uniformity condition can be checked with the generally more familiar deterministic Turing machine (DTM) instead of an alternating Turing machine (ATM). The disadvantage is that ATMs are more powerful than DTMs and it may be easier to check the uniformity with an ATM.

Ruzzo [ru81] developed a Turing machine characterization of uniform Boolean circuits by showing that ATMs are basically uniform circuits.

PROPOSITION 2.1 [ru81]. *A problem is in* $\mathbf{NC}^k$ *if and only if it is solvable by an* ATM *in time* $O(\log^k n)$ *and space* $O(\log n)$.

Notice that in Proposition 2.1 the standard textbook definition of a Turing machine does not make sense because the time bound is sublinear (and, thus, not all of the input can be accessed on a single path of the computation). Therefore, we adopt the random-access multi-tape model described by Chandra, Kozen, and Stockmeyer [cks81]. This machine has a special index tape onto which the address of the input tape cell that needs to be accessed is written (in binary). The input head can then read the value of the input specified by this address. A further complication arises because circuit families are defined as computing multiple-valued functions (that is, the corresponding circuits may have more than one output gate), whereas Turing machines recognize sets of predicates. We make the convention that a Turing machine M is said to compute a function f if the predicate

$$A_f(c, i, z) \overset{\text{def}}{=} \text{the } i\text{th symbol of } f(z) \text{ is } c$$

is recognized by M.

Following Ruzzo [ru81], we also make a number of assumptions (without loss of generality) about ATMs. First, every configuration of an ATM has at most 2 successors. Second, all accesses to the ATM's input tape are performed at the end of the computation. This is easily accomplished by having the ATM guess the input and in parallel verify it (by looking at the input tape) and continue with the computation. Finally, we make the convention that deterministic configurations are considered to be existential with one successor.

$\mathbf{NC}^1$ is considered to be a very fast complexity class, and many problems have been shown to be in $\mathbf{NC}^1$. Sum and product of 2 n-bit integers, sum of n n-bit integers, and sorting of n n-bit integers are all in $\mathbf{NC}^1$ [sa76]. Because of their shallow depth, $\mathbf{NC}^1$ circuits can always be converted into equivalent circuits with fan-out 1, polynomial size, and $O(\log n)$ depth. In this form, they can be expressed as formulas.

COROLLARY 2.2. *$\mathbf{NC}^1$ is the class of languages recognized by uniform log-depth formula families. The nth member of the family recognizes all strings in the language of length n.*

A generalization of the uniform Boolean circuit families are the unbounded fan-in uniform circuit families [csv82]. These circuit families are allowed arbitrary fan-in at the $\wedge$ and $\vee$ gates. We need a new uniformity condition.

DEFINITION. The *direct connection language* for an unbounded fan-in family of circuits $\langle \alpha_n \rangle$ (denoted L_{DC}) is given by the set of 3-tuples $\langle \bar{n}, g, y \rangle$, where $\bar{n}, g \in \{0,1\}^*$, $y \in \{x_1, \cdots, x_n, 0, 1, \neg, \vee, \wedge\} \cup \{0,1\}^*$ such that if $y \in \{0,1\}^*$ then y is an input to g; otherwise, y is g's label. $\langle \alpha_n \rangle$ is U_{DL} *uniform* if L_{DC} can be recognized by a DTM in linear space (i.e., $O(\log c(\alpha_n))$ space).

We define a hierarchy of unbounded fan-in circuits by the following.

DEFINITION. For all $k > 0$, $\mathbf{AC}^k$ is the class of problems solvable by an unbounded fan-in U_{DL}-uniform circuit family $\langle \alpha_n \rangle$, where $c(\alpha_n) = n^{O(1)}$ and $d(\alpha_n) = O(\log^k n)$. $\mathbf{AC} = \bigcup_{k>0} \mathbf{AC}^k$.

Once again we can characterize this hierarchy by using alternating Turing machines:

PROPOSITION 2.3 [co85]. *For all $k > 0$, $\mathbf{AC}^k$ is the class of problem solvable by an* ATM *in space $O(\log n)$ and alternation depth $O(\log^k n)$.*

The definitions above suffice to define $\mathbf{AC}^k$ when $k > 0$. However, we are interested in $\mathbf{AC}^0$ because we wish to show that BSVP is complete for $\mathbf{NC}^1$ under $\mathbf{AC}^0$ reductions. The uniformity condition is too strong in the circuit definition. The ATM definition (Proposition 2.3) would have to place further resource restrictions on the machine since a straightforward extension of the proposition would imply $\mathbf{AC}^0 = \mathbf{NL}$ (nondeterministic log space).

Immerman [im89] proposed defining $\mathbf{AC}^k$ $(k \geq 0)$ in terms of a CRAM (a CRCW PRAM that is strengthened slightly to allow a processor to shift a word left or right by $\log n$ bits in unit time). This modification does not affect $\mathbf{AC}^k$ when $k > 0$. Immerman also gave a number of other characterizations of $\mathbf{AC}^0$, including first-order expressible properties and inductive definitions for which the depth of the induction is constant, and showed that all these characterizations are equivalent.

An alternate definition was proposed by Buss [bu87]. The log time hierarchy (denoted $\mathbf{LH}$) introduced by Sipser [si83] is the class of problems solvable by an ATM in log time and $O(1)$ alternations. Buss proposed $\mathbf{LH}$ as the definition of uniform $\mathbf{AC}^0$.

Recently Barrington, Immerman, and Straubing [bis88] showed that all 4 of the above characterizations give the same class, thus suggesting that these may be the appropriate definition. We would like a circuit definition of $\mathbf{AC}^0$. We begin by defining an appropriate uniformity condition.

DEFINITION [ru81]. $\langle \alpha_n \rangle$ is U_D *uniform* if its direct connection language, L_{DC}, can be recognized by a DTM in linear time (i.e., $O(\log c(\alpha_n))$ time).

Finally, $\mathbf{AC}^0$ is the class of problems solvable by an unbounded fan-in U_D-uniform circuit family $\langle \alpha_n \rangle$ that has constant depth and $n^{O(1)}$ size. This definition is consistent with the others due to the following.

THEOREM 2.4. *U_D-uniform* $\mathbf{AC}^0 = \mathbf{LH}$.

The proof of Theorem 2.4 requires the following lemma.

LEMMA 2.5. *If $L \in$ **LH**, then there is an* ATM *M that accepts L and there are $c, k \in \mathbb{N}$ so that for all n and all x in $\{0,1\}^*$ with $|x| = n$, and all configurations γ with $|\gamma| \leq c \log n$, every computation of M with input x starting in γ terminates within $c \log n$ steps and at most k alternations.*

To handle the time constraint, we incorporate a clock into the ATM that times the computation. When the clock runs out, the ATM automatically rejects. Because the clock can count in unary, this at most doubles the running time. For the bound on the alternations, we can use the finite state of the machine to count the number of alternations that have occurred.

Proof (*Theorem* 2.4). ($\subseteq$) An ATM M on input x guesses the output gate (say g) of $\alpha_{|x|}$. If g is "$\wedge$" ("$\vee$"), then M enters a universal (existential) state. M now guesses an input gate to g. If this gate is an input to the circuit, then M directly checks its corresponding input and accepts or rejects appropriately. Otherwise, it recursively applies this procedure to this new gate. All guesses about the circuit are verified by checking for the appropriate membership in L_{DC}. Since $\langle \alpha_n \rangle$ is constant depth and M uses at most 2 alternations to simulate each step of the circuit, M is in **LH**.

($\supseteq$) Let $L \in$ **LH**. Let M be a log time ATM recognizing L, and let $c, k \in \mathbb{N}$, as in Lemma 2.5. We define a U_D-uniform family of circuits $\langle \alpha_i \rangle$ that simulate M and have depth k.

The gates of α_n are labeled by a 3-tuple $\langle \gamma, t, \rho \rangle$, where

1. γ is a configuration of M (on inputs of length n).
2. $t \in \{\wedge, \vee, I, \overline{I}, 0, 1\}$, where
   (a) $t = \wedge$ if γ is an universal configuration.
   (b) $t = \vee$ if γ is an existential configuration.
   (c) $t = I$ if γ is an input configuration and M in configuration γ accepts if $x_i = 1$, where i is on the index tape.
   (d) $t = \overline{I}$ if γ is an input configuration and M in configuration γ accepts if $x_i = 0$, where i is on the index tape.
   (e) $t = 0$ if γ is a rejecting configuration.
   (f) $t = 1$ if γ is an accepting configuration.
3. $\rho \in \{l, r\}^*$, where $|\rho| \leq c \log n$.

There is exactly one gate for every possible triple $\langle \gamma, t, \rho \rangle$. The output gate is the gate labeled $\langle \gamma_0, t_0, \Lambda \rangle$, where γ_0 is the initial configuration, t_0 is the type of γ_0, and Λ is the empty string. The input gates are the triples $\langle \gamma, t, \rho \rangle$ of type t equal to I, $\overline{I}$, 0 or 1; these are identified with the inputs x_i, $\overline{x}_i$, 0 and 1 (respectively), where i is the value on the index tape of the configuration γ. If $g_1 = \langle \gamma_1, t_1, \rho_1 \rangle$ and $g_2 = \langle \gamma_2, t_2, \rho_2 \rangle$ are gates of α_n, then g_1 is an input to g_2 if $t_1 \neq t_2$ and the computation described by ρ_1 starting in configuration γ_2 ends at γ_1 such that all configurations in this computation except the last are of type t_2.

It is straightforward to show by induction that a gate $g = \langle \gamma, t, \rho \rangle$ is 1 if and only if γ is accepting with respect to the input x. Also, the depth of the circuit is clearly the number of alternations of M.

It remains to be proved that the circuit family $\langle \alpha_i \rangle$ is U_D uniform. However, this now follows directly, since for the given gates $g_1 = \langle \gamma_1, t_1, \rho_1 \rangle$ and $g_2 = \langle \gamma_2, t_2, \rho_2 \rangle$ we can simulate the computation specified by ρ_1 to determine if g_1 is an input to g_2. □

Note. If the U_D-uniformity condition is used in the definition of $\mathbf{AC}^k$ ($k > 0$), the class does not change. Also, the ATM definition of $\mathbf{AC}^k$ (Proposition 2.3) can be augmented to include $k = 0$ by adding the further restriction that the machine operate in time $O(\log^{k+1} n)$.

We are now ready to define $\mathbf{AC}^0$ reductions:

DEFINITION. Let A and B be sets. Then $A \leq_{\mathbf{AC}^0} B$ if there is a function f in $\mathbf{AC}^0$ such that for every x, $x \in A$ if and only if $f(x) \in B$.

THEOREM 2.6. *If* $A \leq_{\mathbf{AC}^0} B$ *and* $B \leq_{\mathbf{AC}^0} C$, *then* $A \leq_{\mathbf{AC}^0} C$.

Proof. Let f be an $\mathbf{AC}^0$ function such that $x \in A$ if and only if $f(x) \in B$. Let M be an ATM computing f. Let g be an $\mathbf{AC}^0$ function such that $x \in B$ if and only if $g(x) \in C$. Let N be an ATM computing g. We show that $g \circ f$ is computable in $\mathbf{AC}^0$ by an ATM T. Suppose the input to T is $\langle c, i, x\rangle$. T must accept if and only if the ith bit of $(g \circ f)(x)$ is c. T begins by simulating N on input $\langle c, i, x\rangle$ until N enters an input state. Suppose that when N does so, it has j written on its index tape and N would accept if x_j was $b \in \{0, 1\}$. T now finishes by simulating M on input $\langle b, j, x\rangle$. Clearly, T runs in log time, uses a constant number of alternations, and computes $g \circ f$. $\square$

THEOREM 2.7. *If* $B \in \mathbf{NC}^1$ *and* $A \leq_{\mathbf{NC}^1} B$, *then* $A \in \mathbf{NC}^1$.

Proof. Let f be a function such that for every x, $x \in A$ if and only if $f(x) \in B$ and f is in $\mathbf{NC}^1$. Let M be an ATM recognizing B in log time. Let N be an ATM computing f (that is, recognizing A_f) in log time. We describe an ATM T that recognizes A. T simulates M except where M enters an input configuration. Suppose T has simulated M up to an input configuration and has i written onto its index tape and c as the guess for the ith input bit. At this point T begins to simulate N with the input $\langle c, i, x\rangle$. It is easy to see that T accepts input x if and only if M accepts $f(x)$. Also, since M and N run in log time, so does T. Therefore, $A \in \mathbf{NC}^1$. $\square$

COROLLARY 2.8. *If* $B \in \mathbf{NC}^1$ *and* $A \leq_{\mathbf{AC}^0} B$, *then* $A \in \mathbf{NC}^1$.

2.2. Arithmetic circuit complexity.

Most of the material in this section can be found in [jg86].

DEFINITION. An *arithmetic circuit (straight–line program)* over an algebraic structure F is a directed acyclic graph for which each node has indegree 0, 1, or 2. Nodes with indegree 0 are labeled as either input nodes or elements of F. Nodes with indegree 1 and 2 are labeled with the unary and binary operators of F, respectively. For example if F is a field, then the unary operators are "$-$" (additive inverse) and "${}^{-1}$" (multiplicative inverse) and the binary operators are "$+$" and "$\times$". There is a sequence of $m \geq 1$ gates with outdegree 0 designated as output nodes.

As with Boolean circuits, we assume there are no superfluous nodes. For an arithmetic circuit α the *complexity* and *depth* of α are defined the same as for Boolean circuits.

Arithmetic circuits are not sufficiently powerful for our purpose. For example, there may be no way to describe and manipulate the formula within the particular algebraic structure.

DEFINITION. An *arithmetic-Boolean circuit* over an algebraic structure F is an arithmetic circuit (over F) augmented with a Boolean component and an interface between the two. The Boolean component is a Boolean circuit. The interface consists of two special gates—sign : $F \to \{0, 1\}$, defined by $\mathrm{sign}(a) = 0$ if and only if $a = 0$, and sel : $F \times F \times \{0, 1\} \to F$, defined by

$$\mathrm{sel}(a, b, c) = \begin{cases} a & \text{if } c = 0, \\ b & \text{if } c = 1. \end{cases}$$

The definitions of complexity and depth for arithmetic-Boolean circuits are extended from arithmetic circuits. Also, the definitions of arithmetic-Boolean circuit families, uniformity, and parallel complexity classes (i.e., the **NC** hierarchy) are analogous to those for Boolean circuits.

Inputs to an arithmetic-Boolean circuit consist of algebraic values to the arithmetic circuit and Boolean values to the Boolean circuit. In the case of arithmetic-formula evaluation, the Boolean inputs will describe the structure of the formula and the arithmetic inputs will specify the value of the variables in the formula.

2.3. Problem definitions.

DEFINITION. A *Boolean sentence* is defined inductively by

1. 0 and 1 are Boolean sentences.
2. If α and β are Boolean sentences, then so are $(\neg\alpha)$, $(\alpha \wedge \beta)$, and $(\alpha \vee \beta)$.

The definition of Boolean sentences above describes sentences in infix notation. However, our algorithm will work with sentences in postfix (reverse Polish) notation with the further provision that for any binary operator, the longer operand occurs first.

DEFINITION. A *postfix-longer-operand-first* (PLOF) sentence is defined by

1. 0 and 1 are PLOF sentences.
2. If α and β are PLOF sentences where $|\alpha| \geq |\beta|$, then $\alpha\neg$, $\alpha\beta\wedge$, and $\alpha\beta\vee$ are PLOF sentences.

We define the value of a Boolean or PLOF sentence in the usual way, where 0 and 1 represent False and True, respectively.

DEFINITION. The Boolean sentence-value problem (BSVP) is as follows: Given a Boolean sentence A, what is the value of A?

DEFINITION [js82]. A *semi-ring* $\mathbb{S}$ is a 5-tuple $(S, \oplus, \otimes, 0, 1)$, where $0, 1 \in S$ such that

1. $(S, \oplus, 0)$ is a commutative monoid.
2. $(S, \otimes, 1)$ is a monoid.
3. $\otimes$ distributes over $\oplus$.
4. For every $a \in S, a \otimes 0 = 0 = 0 \otimes a$.

For convenience, we will also assume a unary operator "$\odot$," where $\odot a = a$ for every $a \in S$. This will give us flexibility to increase the size of a formula over a semi-ring.

Some examples of semi-rings are $\mathbb{S} = (\{0,1\}, \vee, \wedge, 0, 1)$, $\mathbb{S} = (\mathbb{Z}, \min, \times, +\infty, 1)$, and any ring $\mathbb{S}$.

DEFINITION. Let $\mathbb{S}$ be a semi-ring (which may also be a ring or field). An *arithmetic formula* over $\mathbb{S}$ with indeterminates $X_1, X_2, \cdots, X_n$, is defined by

1. For $1 \leq i \leq n$, X_i is an arithmetic formula.
2. For every $c \in \mathbb{S}$, c is an arithmetic formula.
3. If α is an arithmetic formula and θ is a unary operator of $\mathbb{S}$, then $(\theta\ \alpha)$ is an arithmetic formula.
4. If α and β are arithmetic formulas and θ is a binary operator of $\mathbb{S}$, then $(\alpha\ \theta\ \beta)$ is an arithmetic formula.

An arithmetic formula A with indeterminates $X_1, X_2, \cdots, X_n$ is denoted by $A(X_1, \cdots, X_n)$.

The Boolean formula discussed earlier is clearly a special case of these new formulas. We define postfix arithmetic formula and PLOF (postfix-longer-operand-first) arithmetic formulas to be exact analogs of their Boolean counterparts. The length of an arithmetic formula A (denoted $|A|$) is the number of nonparenthesis symbols in A (where an indeterminate is one symbol).

DEFINITION. Let $\mathbb{S}$ be a ring, field, or semi-ring. The *arithmetic-formula evaluation problem* is as follows: Given an arithmetic formula $A(X_1, X_2, \cdots, X_n)$ over $\mathbb{S}$ and constants $c_1, c_2, \cdots, c_n \in \mathbb{S}$, what is $A(c_1, c_2, \cdots, c_n)$?

2.4. Other definitions. Consider a Boolean sentence A. Define the *depth* of atoms of A as the level of nesting of parentheses in the subsentence containing the atom. We can view A as a binary tree, namely its (unique) parse tree, defined inductively as follows: the root is the operator of A of minimum depth and its children are the roots of the trees of the operands of the operator. Notice that we do not need parentheses in the tree representation. In exposition we will use the tree representation interchangeably with the infix or PLOF representation. Therefore, we carry over tree notions such as *root, child, ancestor* and *descendent* to sentences.

DEFINITION. Let A be a Boolean sentence. The length of A, denoted $|A|$, is the number of nonparenthesis symbols in A.

This definition has the desirable property that sentences have the same length regardless of the representation used (either infix or PLOF).

DEFINITION. Let A be a postfix Boolean sentence, and suppose $1 \leq j \leq k \leq n$. Then $A[j,k]$ is the string $A[j]A[j+1]\cdots A[k]$. The *subsentences* of A are those strings of the form $A[j,k]$ that form sentences. For $1 \leq k \leq n$, A_k denotes the unique subsentence of A of the form $A[j,k]$ for some j. We call A_k the subsentence *rooted at position* k or, for short, *rooted at* $A[k]$. We use $j \trianglelefteq k$ to mean that $A[j]$ is in A_k and $j \triangleleft k$ to mean $j \trianglelefteq k$ and $j \neq k$.

Note that $j \trianglelefteq k$ if and only if A_j is a substring of A_k if and only if $A[k]$ is an ancestor of $A[j]$. Also, the relation $\trianglelefteq$ forms a partial order. The following fact is used often:

LEMMA 2.9. *Let A be a postfix Boolean sentence $|A| = n$. Let $a, b, c < n$ such that $c \trianglelefteq a$, $c \trianglelefteq b$, and $a \leq b$. Then $a \trianglelefteq b$.*

Proof. The subsentence A_b is of the form $A[j,b]$ $(j < b)$. $c \trianglelefteq b \Rightarrow j < c < b$ and $c \trianglelefteq a \Rightarrow c < a$. Therefore, $j < c < a < b \Rightarrow a \trianglelefteq b$. □

DEFINITION. Let A be a Boolean sentence. Consider the sentence obtained by removing a subsentence A_k and replacing it with some constant c (i.e., $c \in \{0,1\}$). The resulting sentence is denoted by $A(k,c)$. We say that $A(k,c)$ is A with a *scar* at k and that $A(k,c)$ is *scarred*.

All the definitions made here can be translated to arithmetic formulas in a natural way, and we will use these definitions when discussing arithmetic formulas.

3. Translation of Boolean sentences to PLOF sentences.

As a first step toward finding an $\mathbf{NC}^1$ algorithm for BSVP, we give an $\mathbf{NC}^1$ algorithm that translates Boolean sentences into PLOF sentences. It is well known [co85] that there is a uniform family of $\mathbf{NC}^1$ circuits in which the nth circuit computes the function

$$Count_n : \{0,1\}^n \to \{0,1\}^{\log n}$$

(i.e., given n Boolean values, output a binary string denoting their sum). Likewise, there is a uniform family of $\mathbf{NC}^1$ circuits where the nth circuit computes the summation of n n-bit numbers.

If A is an infix Boolean formula, let $A[i] \in \{\wedge, \vee, \neg, 0, 1\}$ be the ith nonparenthesis symbol in A. We describe an algorithm that outputs, for each $A[i]$, its position in the PLOF sentence.

DEFINITION. For A, an infix Boolean sentence, and $A[1]A[2]\cdots A[n]$, the enumeration of the nonparenthesis symbols of A, the *subsentence rooted at* $A[i]$ is the smallest subsentence of A containing $A[i]$. $A[k]$ is an *ancestor* of $A[i]$ if the subsentence rooted at $A[k]$ contains the subsentence rooted at $A[i]$.

LEMMA 3.1. *The following are computable in* $\mathbf{NC}^1$.

a. $Scope(A,i,j) \stackrel{\text{def}}{=}$ *$A[i]$ in the subsentence rooted at $A[j]$.*

b. *For each j, the size of the subsentence rooted at $A[j]$.*

Proof. Notice that by counting the nonparenthesis symbols in A, it is easy to locate $A[j]$ in A.

a. Define the depth of nonparenthesis symbols in A in the normal way (i.e., with respect to the parse tree for A). Using *Count*, it is easy to determine the depth of each nonparenthesis symbol of A. This is used to find the parentheses that delimit the subsentence rooted at $A[j]$. $A[i]$ is in this subsentence if it sits between these parentheses.

b. Count the number of i for which $Scope(A, i, j)$ holds. □

To determine the position of nonparenthesis symbol $A[j]$ in the PLOF translation of A, do the following:

1. Calculate the size of the subsentence rooted at $A[j]$.
2. For each ancestor $A[k]$ of $A[j]$, let L_k (R_k) be the size of the subsentence rooted at the left (right) child of $A[k]$. Define

$$S_k = \begin{cases} L_k & \text{if } L_k \geq R_k \text{ and } Scope(A, j, \text{right child of } A[k]), \\ R_k & \text{if } R_k > L_k \text{ and } Scope(A, j, \text{left child of } A[k]), \\ 0 & \text{otherwise.} \end{cases}$$

3. The position of $A[i]$ is $\sum S_k$.

Because summation is $\mathbf{NC}^1$ computable, it is now easy to see that the function mapping an infix formula A to its PLOF translation is $\mathbf{NC}^1$ computable.

4. The algorithm for the PLOF sentence-value problem. Because there is an $\mathbf{NC}^1$ algorithm that translates a Boolean sentence into an equivalent PLOF sentence, it suffices by Theorem 2.7 to prove the next theorem in order to prove that the Boolean sentence-value problem is in $\mathbf{NC}^1$.

THEOREM 4.1. *There is an* $\mathbf{NC}^1$ *algorithm for determining the truth value of a* PLOF *sentence.*

For the remainder of this section, we present a proof of Theorem 4.1. A good way to explain the algorithm is to use an interpreted version of the Dymond–Tompa 2-person pebbling game [dt85] (this is the standard simulation of a Boolean circuit by an ATM). This version can be used to determine the output of a Boolean circuit C with specified inputs, as follows. The game has two players, called the `Pebbler` and the `Challenger`. The `Pebbler` has a supply of pebbles, each labeled either 0 or 1. The `Pebbler` moves by placing a pebble on a node of C. (The node is either a gate or an input to C.) The label on the pebble represents the `Pebbler`'s guess as to the value of the node pebbled. The `Pebbler` moves first by placing a pebble labeled 1 on the output node, representing a guess that the output value of the circuit is 1. After each `Pebbler` move, the `Challenger` moves by challenging some pebbled node. The challenged node must either be the one just pebbled or the node last challenged by the `Challenger`. The game ends when all inputs to the challenged node have been pebbled (pebbles are never removed once placed by the `Pebbler`). The `Pebbler` wins if and only if the label on the pebble of the challenged node is consistent with the node type and the labels on its inputs.

For example, if the challenged node is an input node with value 1, then the `Pebbler` wins if and only if the pebble on that node has label 1. If, on the other hand, the challenged node is an OR gate with pebble label c and its inputs have pebble labels x and y, then the `Pebbler` wins if and only if c is the logical OR of x and y.

LEMMA 4.2. *In the above game, the* `Pebbler` *has a winning strategy if and only if the circuit has output* 1.

Proof. If the circuit has output 1, then the `Pebbler`'s strategy is, for each move, to pebble with the correct label all unpebbled inputs to the challenged node. If the circuit has output 0, then the `Challenger`'s strategy is to always challenge the node of minimum depth whose pebbled value is incorrect. □

The above game forms the basis for our $\mathbf{NC}^1$ algorithm for determining the value of a PLOF sentence. The input to the algorithm is a PLOF sentence A, which is a string of symbols $A = A[1] \cdots A[n]$ from the alphabet $\{\vee, \wedge, \equiv, \neg, 0, 1\}$. Hence, the length of A, $|A|$, is n.

Without loss of generality, we may assume that n is a power of 2. If n is not in fact a power of 2, the algorithm proceeds as if a string of $\neg$'s is tacked on to the right end, bringing the length of the input to the nearest larger power of 2. If the number of such $\neg$'s is odd, the normal output of the algorithm is negated.

To adapt the pebbling game from the circuit C to the PLOF sentence A, the `Pebbler` places a pebble on a position k of A instead of a node of C, and the label on the pebble is a guess as to the value of subsentence A_k. The maximal subsentences of A_k are the inputs to the node.

If the sentence has value 1, then the `Pebbler` can force a win in $O(\log n)$ moves by the strategy used by Tompa [to85] to efficiently pebble a tree. (The basic idea goes back to Lewis, Stearns, and Hartmanis [lsh65] in their proof that context-free languages can be recognized in space $O(\log^2 n)$.)

This strategy can be described as follows: Consider the challenged subsentence A_k to be scarred by replacing each of its maximal pebbled subsentences by 0 or 1 (the label on the pebble). Place the next pebble on the subsentence A_j of A_k that comes as close as possible to cutting the scarred A_k in half. That is, the scarred size of A_j should be as close as possible to the new scarred size of A_k (in fact, the size of A_j will be between $\frac{1}{3}$ and $\frac{2}{3}$ the size of A_k). In this way, whether the `Challenger` next challenges the same position or the new position, the scarred size of the challenged subsentence is at least $\frac{1}{3}$ less after each pair of moves.

A straightforward implementation of this strategy on an ATM requires time $O(\log^2 n)$, since each of the $O(\log n)$ steps requires time $O(\log n)$ to describe a pebble position. To reduce the ATM time, we present a variation of the strategy that includes a uniform method for choosing subsentences, so that each pebble move can be described in a constant number of bits.

Associated with each of the `Pebbler`'s moves is a substring $g = A[i, j]$ of the input sentence A whose length is a power of 2. This substring includes the currently challenged position k and all unpebbled positions in the scarred subsentence A_k. All future moves are made within g. To help determine these moves, we define distinguished positions $V(g)$, $V_1(g)$, and $V_2(g)$, in g which depend only on A and the end points (i and j) of g.

DEFINITION. Let $g = A[i, j]$. If g has even length, define

$$V(g) = \max\{k | k \leq j \text{ and } i \trianglelefteq k\}.$$

That is, $A_{V(g)}$ is the maximal subsentence of A containing $A[i]$ whose root is in g. Further, define

$$V_1(g) = V(A[i, (i+j-1)/2]) \quad \text{and} \quad V_2(g) = V(A[(i+j+1)/2, j]).$$

Here, $V_1(g)$ and $V_2(g)$ are just "V(first half of g)" and "V(second half of g)," respectively.

LEMMA 4.3. *Let $g = A[i, j]$ have even length. Then $V(g)$ is one of $V_1(g)$ and $V_2(g)$.*

Proof. If $V(g) \leq (i+j-1)/2$, then $V(g) = V_1(g)$; otherwise, it is $V_2(g)$. □

LEMMA 4.4. *Let $g = A[i,j]$ have even length. Then $V_1(g) + 1 \unlhd V_2(g)$.*

Proof. If $V_1(g) = (i+j-1)/2$ or $V_1(g) \unlhd V_2(g)$, the lemma is obvious. Therefore, assume that $V_1(g) < (i+j-1)/2$ and $V_1(g) \not\unlhd V_2(g)$. Let $W = V(A[V_1(g)+1, j])$. We show that $W = V_2(g)$. If $W \geq (i+j+1)/2$, then $W = V_2(g)$ since A_W and $A_{V_2(g)}$ are maximal and both contain $A[(i+j+1)/2]$. Now suppose that $W \leq (i+j-1)/2$ (see Fig. 4.1).

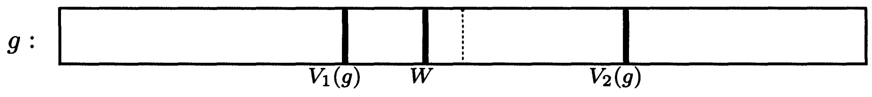

FIG. 4.1. $V_1(g) \not\unlhd V_2(g)$ *and* $W \leq (i+j-1)/2$.

Clearly, A_W is the left operand of its parent operator, and its parent occurs to the right of g (since otherwise we could extend W to include its parent). But A_W is at least as long as its sibling to its immediate right because the input is a PLOF sentence. Since $|A_W| < (i+j-1)/2$, the entire sibling as well as the parent, must be in g, a contradiction to the definition of W. □

DEFINITION. Let $g = A[i,j]$ be a substring of A whose length $|g| = j-i+1$ is divisible by 4. Then g_1, g_2, and g_3 denote the left, middle, and right halves of g, respectively. That is, $g_1 = A[i, i+|g|/2-1]$, $g_2 = A[i+|g|/4, i+3|g|/4-1]$, and $g_3 = A[i+|g|/2, j]$.

Our pebbling game will allow the Pebbler to place pebbles only at the V position corresponding to each quarter of g (see Fig. 4.2).

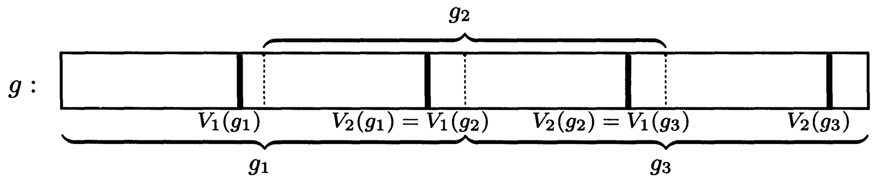

FIG. 4.2. *Pebble positions in g.*

It is easy to see

LEMMA 4.5. $V_1(g_1) < V_2(g_1) = V_1(g_2) < V_2(g_2) = V_1(g_3) < V_2(g_3)$.

Rules of the ATM game. The `Pebbler` may pebble up to 4 positions in one round of moves, as specified below. The `Challenger` challenges one of these 4. Associated with each round (except the first) is a substring g of A, whose length is a power of 2 and which contains the challenged position. We assume $|A| \geq 2$.

1. For the first round, the `Pebbler` places a pebble with label 1 on position n. The `Challenger` challenges n. The substring associated with the next round is $g = A[1,n]$. (Recall that n is a power of 2 by our earlier assumption.)
2. After the first round, the substring g contains the challenged position c. Assume $|g| \geq 4$. The pebbler considers $V_1(g_1), V_2(g_1), V_2(g_2), V_2(g_3)$, in that order, for pebbling. For each of these 4 candidates k, the `Pebbler` pebbles k if and only if $k \unlhd c$ and no *pebbled* l satisfies $k \unlhd l \lhd c$. The label on the pebble may be

either 0 or 1, except that if $k = c$ (the only case in which a position is allowed to be repebbled), the label must agree with the previous label. The `Challenger` challenges one such pebbled $V_i(g_j)$. Notice that by Lemma 4.3, this permits rechallenging c. The new substring is g_j.

3. Assume $|g| = 2$. Then g consists of the challenged position c and a second position k. The `Pebbler` pebbles k if and only if k is unpebbled and $k \lhd c$, and then repebbles c with the same label as before. The `Challenger` challenges one of the positions c' just pebbled. Lemma 4.6 below shows that all maximal subsentences of $A_{c'}$ have been pebbled. The `Pebbler` wins if and only if the pebble label on c' is consistent with the operator $A[c']$ and the pebble labels on the maximal subsentences of $A_{c'}$.

The following lemma justifies our assertion that if c is the challenged position, then the substring g contains all unpebbled positions in the scarred subsentence A_c. It also justifies the end condition in step 3 of the Rules.

LEMMA 4.6. *After each round in the* ATM *game, every position $k \lhd c$ with k to the left of g has some pebbled l such that $k \unlhd l \lhd c$.*

Proof. Induction on the round number.

Basis. Vacuous.

Induction. Suppose c' is the new challenged position and g_j is the new interval. By the Rules, $c' \unlhd c$. Let $k \lhd c'$, where k is to the left of g_j. Thus, $k \lhd c' \unlhd c$.

Case 1. k is to the left of g. By the induction hypothesis, there is a pebbled l with $k \unlhd l \lhd c$. By the *Rules*, $c' \not\!\unlhd l$. But subsentences $A_{c'}$ and A_l both contain k, so by Lemma 2.9 one subsentence contains the other. Hence, $k \unlhd l \lhd c'$.

Case 2. k is in g but to the left of g_j. Then k is in g_1. If $j = 2$, then by Lemma 4.4 $k \unlhd V_1(g_1)$ or $k \unlhd V_2(g_1)$ and by the *Rules* $c' = V_2(g_2)$. Similarly, if $j = 3$, then $k \unlhd V_1(g_1)$, $k \unlhd V_2(g_1)$, or $k \unlhd V_2(g_2)$ and $c' = V_2(g_3)$. In general, say that $k \unlhd V_a(g_b)$. Then clearly $V_a(g_b) < c'$, and by Lemmas 2.9 and 4.5 $k \unlhd V_a(g_b) \lhd c' \unlhd c$ (since the subsentences at both position $V_a(g_b)$ and c' contain k). If $V_a(g_b)$ is pebbled, then we are done, with $l = V_a(g_b)$. Otherwise, by the *Rules*, there is a pebbled l with $V_a(g_b) \unlhd l \lhd c$. Again by the *Rules*, $c' \not\!\unlhd l$. But subsentences $A_{c'}$ and A_l both contain $V_a(g_b)$, so one subsentence contains the other. Hence, $k \unlhd V_a(g_b) \unlhd l \lhd c'$. □

LEMMA 4.7. *In the* ATM *game, the* `Pebbler` *has a winning strategy if and only if the value of A is* 1.

Proof. The proof is similar to that of Lemma 4.2. The positions pebbled are completely determined by the *Rules* and the positions challenged. The only choice given to the `Pebbler` is the labels on the pebbles. If the value of A is 1, the `Pebbler`'s strategy is to choose each label equal to the value of the subsentence pebbled. If the value of A is 0, the `Challenger`'s strategy is to challenge the leftmost incorrectly labeled pebble in each round. □

It remains for us to show that the game can be implemented on an ATM in time $O(\log n)$.

LEMMA 4.8. *The following predicates are in* $\mathbf{NC}^1$.

a. $Subsentence(A, i, j) \overset{\text{def}}{=} A[i, j]$ *is a well-formed* PLOF *sentence.*

b. $Descendent(A, i, j) \overset{\text{def}}{=} i \unlhd j$.

c. $Child(A, i, j) \overset{\text{def}}{=} A_i$ *is a maximal proper subsentence of* A_j.

d. $V(A, i, j, k) \overset{\text{def}}{=} (k = V(A[i, j]))$.

Proof. Recall that the function *Count* is in $\mathbf{NC}^1$.

a. Check that the number of binary operators in $A[i,j]$ is one less than the number of constants and that for each binary operator in $A[i,j]$ (say $A[k]$) there are more constants in $A[i,k]$ than binary operators.

b. Find the unique k for which $Subsentence(A,k,j)$ holds, and check that $k \leq i \leq j$.

c. Assume that $A[j]$ is a binary operator and that $i \neq j-1$ (otherwise, it is trivial). Check that $A[i+1, j+1]$ is a sentence.

d. Since $V(g)$ is defined in terms of subsentences, descendents, and children, this is immediate from a.–c. □

LEMMA 4.9. *Let A be a* PLOF *Boolean sentence and $\langle p_1, \cdots, p_k\rangle$ ($p_i \in \{1, \cdots, 4\}$ for all p_i) be a sequence representing the challenged positions (from among $V_1(g_1)$, $V_2(g_1)$, $V_2(g_2)$, and $V_2(g_3)$) in the first k rounds of a 2-player game as described above. Then, the following are in* NC^1 :

a. *Determining if the sequence is valid.*

b. *Determining the position of p_k in A.*

c. *Determining the interval g after the kth round.*

Proof. Let $|A| = n = 2^r$. We prove the above in the reverse order.

c. For each i ($1 \leq i \leq k$) let

$$L_i = \begin{cases} 0 & \text{if } p_i = 1,2, \\ \frac{1}{2}\cdot\frac{2^r}{2^i} & \text{if } p_i = 3, \\ \frac{2^r}{2^i} & \text{if } p_i = 4, \end{cases}$$

$$R_i = \begin{cases} \frac{2^r}{2^i} & \text{if } p_i = 1,2, \\ \frac{1}{2}\cdot\frac{2^r}{2^i} & \text{if } p_i = 3, \\ 0 & \text{if } p_i = 4. \end{cases}$$

Here L_i and R_i represent the amount the left and right boundaries of g are moved at the ith round of the game. Then, the current g is given by the string $A[1 + \sum L_i, n - \sum R_i]$.

b. Let g' be the interval after the first $k-1$ moves (i.e., corresponding to plays $\langle p_1, \cdots,$ $p_{k-1}\rangle$). Then, we use the predicate V to determine the position of the p_kth pebble placement at move k in g'.

a. Denote the currently challenged position after the first i moves by $I(\langle p_1, \cdots, p_i\rangle)$. For every i ($1 \leq i < k$) check that $I(\langle p_1, \cdots, p_i, p_{i+1}\rangle) \trianglelefteq I(\langle p_1, \cdots, p_i\rangle)$, and for every j ($1 \leq j < p_i$) check that $I(\langle p_1, \cdots, p_{i+1}\rangle) \ntrianglelefteq I(\langle p_1, \cdots, p_{i-1}, j\rangle)$. All these checks can be performed in parallel using part b. to compute I. □

DEFINITION. Let A be a PLOF Boolean sentence. Then, a *k-round history* of A is a sequence $\nu = \langle \nu_1, \cdots, \nu_k\rangle$, where $\nu_i = \langle p_i, \tau_{i,1}, \cdots, \tau_{i,p_i-1}\rangle$, $p_i \in \{1, \cdots, 4\}$, and $\tau_{i,j} \in \{0,1\}$. A k-round history $\nu = \langle \nu_1, \cdots, \nu_k\rangle$ of A is *valid* if there is a play of the 2-person game outlined above such that each ν_i represents the ith round of this game.

By this we mean that for each round i ($1 \leq i \leq k$) p_i is the position (in the sequence $V_1(g_1), V_2(g_1), V_2(g_2), V_2(g_3)$) of the challenge node and $\tau_{i,j}$ is the value of the pebble placed by the `Pebbler` at the jth pebble position if this position was pebbled, and otherwise $\tau_{i,j}$ is arbitrary.

Using Lemmas 4.8 and 4.9, it is easy to prove the following.

LEMMA 4.10. *Let A be a* PLOF *Boolean sentence. Let ν be a k-round history of a game on A. Then, there is an* $\mathbf{NC}^1$ *algorithm to determine if ν is valid.*

Since $|g|$ is cut in half each round of the game, the number of rounds is at most $\log n + 1$. The ATM simulates the `Pebbler`'s moves by using existential states and the `Challenger`'s moves by using universal states. It records the history of the play in a string of 10 bits for each round. The 4 possible `Pebbler` moves are recorded by using a pair of bits each, telling (1) whether the position was pebbled and, if the position was pebbled, (2) the label. The `Challenger`'s move is recorded with 2 bits telling which of the potentially 4 moves is challenged. The finite-state control can ensure that the challenged position is one that was actually pebbled.

After the history of the play is recorded, the ATM checks whether 1) the `Pebbler`'s moves are legal and 2) whether the `Pebbler` won. The ATM accepts if and only if both conditions are true. To do 1), Lemma 4.10 is used. Condition 2) is checked by using the information in the history of the game and the *Child* predicate from Lemma 4.8. It is now easy to complete the proof of Theorem 4.1.

5. $\mathbf{NC}^1$ completeness of BSVP.

Theorem 4.1 showed that there is an $\mathbf{NC}^1$ algorithm for recognizing true PLOF sentences; hence, by the $\mathbf{NC}^1$ translation of Boolean sentences into PLOF sentences there is an $\mathbf{NC}^1$ algorithm for recognizing true Boolean sentences. In this section we prove that these results are optimal.

THEOREM 5.1. BSVP *is* $\mathbf{NC}^1$ *complete under many-one* $\mathbf{AC}^0$ *reductions.* BSVP *is also* $\mathbf{NC}^1$ *complete under many-one deterministic log-time reductions.*

Recall that BSVP is the set of true Boolean sentences; Theorem 5.1 also holds for the set of true PLOF sentences. By Theorem 2.4, a deterministic log-time reduction also is an $\mathbf{AC}^0$ reduction. So to prove Theorem 5.1 it suffices to prove completeness under deterministic log-time reductions.

Thus, it suffices to exhibit, for a log-time ATM M, a deterministic log-time function f such that, for any input x (with $|x| = n$), $f(x)$ is a Boolean sentence that has value *true* if and only if M accepts x. The sentence $f(x)$ will essentially be the execution tree of M on input x where the $\vee$'s, $\wedge$'s, 0's, and 1's in $f(x)$ correspond to the existential, universal, rejecting, and accepting configurations in the execution of M, respectively. We begin by building the framework for the proof of Theorem 5.1.

Recall the assumptions made in §2.1 about ATMs—every configuration has at most 2 successors, all accesses to the ATM's input tape are performed at the end of the computation, and deterministic configurations are considered to be existential.

Let M be a log-time ATM; without loss of generality, the input alphabet for M is $\{0, 1\}$ and the runtime of M is bounded by $t(n) = c \cdot \log n + c$ on inputs of length n for some constant c. Throughout the remainder of this proof we will be working with this fixed M. For each configuration s of M, we denote by $l(s)$ and $r(s)$ the successor configurations of s, where the degenerate cases are defined by $l(s) = r(s)$ when s has exactly 1 successor and $s = l(s) = r(s)$ when s has no successors (i.e., s is a halt state or a read state).

Suppose s is a configuration and $\rho \in \{l,r\}^*$. Define

$$\rho(s) = \begin{cases} s & \text{when } \rho = \epsilon \text{ (the empty string)},\\ \gamma(l(s)) & \text{when } \rho = l\gamma,\ (\gamma \in \{l,r\}^*),\\ \gamma(r(s)) & \text{when } \rho = r\gamma,\ (\gamma \in \{l,r\}^*). \end{cases}$$

Intuitively, ρ is a string of choices made by M, and $\rho(s)$ is the configuration of M reached from s by these choices.

We want to define a family of Boolean formulas $\langle F_n \rangle$ such that M on input x accepts if and only if $F_{|x|}(x)$ is a true Boolean sentence (here Boolean formulas are similar to the arithmetic formulas defined in §2.3 except they have $\wedge$ and $\vee$ as operators). Let I^M be the initial configuration of M. We define the nth Boolean formula F_n as follows: F_n has indeterminates $X_1, \cdots, X_n$. Let $\rho \in \{l,r\}^*$. First define the Boolean formulas $\beta_n(\rho)$ ($|\rho| \leq t(n)$) by

1. If $|\rho| = t(n)$, then $\rho(I^M)$ is a halting configuration and we define the following: If $\rho(I^M)$ is accepting (respectively, rejecting), then $\beta_n(\rho) = 1$ (respectively, $\beta_n(\rho) = 0$).
   If $\rho(I^M)$ is a read configuration with i on its index tape and M would accept (respectively, reject) if the ith bit of the input is 1, then $\beta_n(\rho) = X_i$ (respectively, $\beta_n(\rho) = \overline{X_i}$).
2. If $|\rho| < t(n)$, then let $\phi_l = \beta_n(\rho l)$ and $\phi_r = \beta_n(\rho r)$. If $\rho(I^M)$ is a universal configuration, then $\beta_n(\rho) = (\phi_l \wedge \phi_r)$ and otherwise $\beta_n = (\phi_l \vee \phi_r)$.

Now, $F_n = \beta_n(\epsilon)$. Clearly, $F_{|x|}(x)$ is *true* exactly when M accepts x.

If x is an input to M, then $x = x_1 \cdots x_n$ is a vector of 0's and 1's. We let $f(x)$ be the Boolean sentence obtained from $F_{|x|}$ by replacing each literal X_i by the binary digit x_i and each literal $\overline{X_i}$ by the binary digit $1 - x_i$. To prove Theorem 5.1, it will suffice to show that the function $f(x)$ is deterministic log-time computable. Recall that this means that there is a deterministic log-time Turing machine N that, on input $\langle x, i \rangle$, outputs the ith symbol of $f(x)$ in $O(\log n)$ time.

DEFINITION. Let ϕ be a Boolean formula. Let $|\phi|$ denote the number of symbols in ϕ, including parentheses. For each nonparenthesis symbol s in ϕ, the *height* of s is defined inductively by

1. If s is a 0 or 1 or for some i, X_i or $\overline{X_i}$, then the height of s is 0.
2. If s is an operator, then its height is 1 plus the maximum of the heights of its operands.

The height of ϕ is the maximum height over all the symbols of ϕ.

We notice that the formulas $\langle F_n \rangle$ are completely balanced (i.e., for every operator, both its operands have the same height). Also, F_n has height $t(n)$. It is easy to prove the following lemma by induction.

LEMMA 5.2. *Let ϕ be a completely balanced Boolean formula of height s with only binary connectives. Then $|\phi| = 2^{s+2} - 3$.*

Thus, $|F_n| = 2^{t(n)+2} - 3$, and for $\rho \in \{l,r\}^*$ ($|\rho| \leq t(n)$), $|\beta_n(\rho)| = 2^{s+2} - 3$, where $s = t(n) - |\rho|$. Our construction of the deterministic log-time algorithm is based on the following observations about F_n:

1. For each $i < 2^{t(n)+2} - 3$, there is a unique $\rho \in \{l,r\}^*$ such that the ith symbol of F_n is in $\beta_n(\rho)$ but not in $\beta_n(\rho l)$ or $\beta_n(\rho r)$. (We make the convention that the symbols of F_n and of $f(x)$ are numbered starting with 0.)
2. Given $\rho \in \{l,r\}^*$, there is a unique number N_ρ such that $\beta_n(\rho)$ occurs at positions $N_\rho, \cdots, N_\rho + |\beta_n(\rho)| - 1$ in F_n. Specifically, if $|\rho| < t(n)$,

(a) The leftmost "(" of $\beta_n(\rho)$ occurs at position N_ρ of F_n.
(b) The rightmost ")" of $\beta_n(\rho)$ occurs at position $N_\rho + |\beta_n(\rho)| - 1$.
(c) The root of $\beta_n(\rho)$ occurs at position $N_\rho + \frac{1}{2}(|\beta_n(\rho)| - 1)$.
(d) $\beta_n(\rho l)$ occurs at positions $N_\rho + 1, \cdots, N_\rho + \frac{1}{2}(|\beta_n(\rho)| - 3)$.
(e) $\beta_n(\rho r)$ occurs at positions $N_\rho + \frac{1}{2}(|\beta_n(\rho)| + 1), \cdots, N_\rho + (|\beta_n(\rho)| - 2)$.

3. Since $\beta_n(\rho)$ is completely balanced and has height $t(n) - |\rho|$, $|\beta_n(\rho)| = 2^{t(n)-|\rho|+2} - 3$. Hence, $N_{\rho l} = N_\rho + 1$ and $N_{\rho r} = N_\rho + 2^{t(n)-|\rho|+1} - 1$.

To compute the ith symbol of the Boolean sentence $f(x)$, we need to find a $\rho \in \{l, r\}^*$ such that $i = N_\rho$ or $i = N_\rho + 2^{t(n)-|\rho|+1} - 2$ or $i = N_\rho + 2^{t(n)-|\rho|+2} - 4$, which indicate that the ith symbol of F_n is the "(", the root, or the ")" of $\beta_n(\rho)$. It is then quite easy to simulate $M(x)$ to determine what the ith symbol of $f(x)$ is. We first give a naive algorithm for computing the ith symbol of $f(x)$; unfortunately, this naive algorithm does not execute in $O(\log n)$ time, so we shall later indicate how to improve its execution time.

```
  Input: $x, i$
 Output: The $i$th symbol of $f(x)$.
Step (1): Compute $n = |x|$.
Step (2): Compute $d = c \cdot \log n + c$. (This is easy because our logarithms are base
          two.)
Step (3): Check that $i < 2^{d+2} - 3 = |f(x)|$; if not, abort.
Step (4): Set $\rho = \epsilon$ (the empty string).
          Set $s = d$.
          Set $j = i$.
Step (5): (Loop while $s \geq 0$)
          Select one case (exactly one must hold):
            Case (5a): If $j = 0$, output "(" and halt.
            Case (5b): If $0 < j < 2^{s+1} - 2$, set $j = j - 1$ and set $\rho = \rho\ell$.
            Case (5c): If $j = 2^{s+1} - 2$, exit to step (6).
            Case (5d): If $2^{s+1} - 2 < j < 2^{s+2} - 4$, set $j = j - (2^{s+1} - 2)$ and set
                       $\rho = \rho r$.
            Case (5e): If $j = 2^{s+2} - 4$, output ")" and halt.
          Set $s = s - 1$.
          If $s \geq 0$, reiterate step (5); otherwise, exit to step (6).
Step (6): Simulate $M$ for $|\rho|$ steps to determine the configuration $\rho(I^M)$.
          If $|\rho| < d$ and $\rho(I^M)$ is a universal configuration, output "$\wedge$".
          Otherwise, if $|\rho| < d$, output "$\vee$".
          Otherwise, if $\rho(I^M)$ is an accepting configuration, output "1".
          Otherwise, if $\rho(I^M)$ is an rejecting configuration, output "0".
          Otherwise, $\rho(I^M)$ is an input configuration with some number $k$ written
          on the index tape. If the value of the $i$th symbol of $x$ would cause this
          configuration to accept, output "1". Otherwise, output "0".
```

It should be clear by inspection that this algorithm correctly computes the ith symbol of $f(x)$. In an iteration of the loop in step (5), s is equal to $t(n) - |\rho|$, and it has already been ascertained that the ith symbol of $F_{|x|}$ is the jth symbol of the subformula $\beta_{|x|}(\rho)$. The subformula $\beta_{|x|}(\rho)$ is of the form $\beta_{|x|}(\rho l) * \beta_{|x|}(\rho r)$, where $*$ is either $\vee$ or $\wedge$; the five cases correspond to the jth symbol being (a) the initial parenthesis, (b) in the subformula $\beta_{|x|}(\rho l)$, (c) the logical connective symbol, (d) in the subformula $\beta_{|x|}(\rho l)$, or (e) the final parenthesis.

To complete the proof of Theorem 5.1, we must prove that there is a log-time deterministic Turing machine N for computing the ith symbol of $f(x)$. In the above algorithm, each step other than step (5) takes $O(\log n)$ time. In particular, for step (6) the simulation of M is hard wired and N simulates each operation of M with only one operation. Step (1) can be executed by finding the least k such that $n < 2^k$ and then using a binary search to calculate n. Step (5), however, is more difficult: There are $O(\log n)$ iterations of the loop, and each iteration takes $O(\log n)$ time in our naive implementation—we need each iteration to take constant time.

The reason that each iteration takes $O(\log n)$ time is that in case (5d), for example, to subtract $2^{s+1} - 2$ from j, both the high- and low-order bits of j must be modified; but j has $O(\log n)$ bits, so it takes too much time just to move the tape head from one end of j to the other. Similar problems arise in comparing j to $2^{s+1} - 2$ and $2^{s+2} - 4$. Also, even when just decrementing j by 1 in case (5b), it may take $O(\log n)$ time to propagate a borrow.

Fortunately, all these problems can be avoided by a simple trick. Before starting step (5), N breaks j into two parts: the low-order $2 + \log d$ bits of j are stored on a tape in *unary* notation; the remaining high-order bits of j are kept on a different tape in binary notation. Thus, to decrement j by 1, N merely changes one tape square on the unary tape and moves that tape head one square. To subtract $2^{s+1} - 2$ from j, N need only change two squares on the unary tape and modify one square of the binary tape (since $j \leq 2^{s+2} - 4$). A complication arises when there is a carry or borrow out of the $(2 + \log d)$-th bit position of j. N handles this by allowing the unary tape to overflow (and cause a carry) or underflow (and cause a borrow). To do this the unary tape is initialized with a marker indicating where the overflow or underflow occurs; since the unary part of j is changed by -1 or $+2$ at most $d = c \cdot \log n + c$ times, at most one marker is needed. During the iterations of the loop in step (5) N remembers whether or not an underflow/overflow has occurred. N also initializes the binary tape with a marker that indicates how far the borrow or carry will propagate.

We can now summarize how N executes step (5) in $O(\log n)$ time. First j is split into binary high-order and unary low-order parts—these are stored on separate tapes along with borrow/carry information. Then the loop is executed for $s = d$ to $s = 1 + \log d$, maintaining the value of j in the split binary/unary form. After these iterations, the higher-order, binary portion of j is equal to zero. The unary portion of j is now converted back to binary notation, and the remaining iterations of the loop with $s = \log d$ to $s = 0$ are executed in the normal naive fashion with j in binary notation. This completes the proof of Theorem 5.1. □

The set of true PLOF sentences is also complete for $\mathbf{NC}^1$ under deterministic log-time reductions. This is proved similarly to the proof of Theorem 5.1: It must be shown that the ith symbol of $f(x)$ in postfix notation can be obtained in deterministic log time.

6. Log depth circuits for arithmetic formula evaluation. We begin by describing a 2-player game (similar to that in §4) for evaluating arithmetic formulas over commutative semi-rings. We then transform this game into a log depth arithmetic-Boolean circuit over the commutative semi-ring. Finally we show how the game can be modified to solve the problem for noncommutative semi-rings, rings, and fields.

Throughout this section let $\mathbb{S}$ be some fixed commutative semi-ring and A be an arithmetic formula over $\mathbb{S}$ of length n. Without loss of generality we can assume that n is a power of 2. If n is not a power of 2, we assume that A has a string of $\odot$ attached to the left-hand side, bringing the total length of A to the next power of 2. (Recall that "$\odot$" is the unary identity operator.) Let $A(j, X)$ be A with A_j (the subformula rooted at position j) replaced by the indeterminate X. Recall that this is equivalent to saying that

A is scarred at j. Then we can write

$$A(j, X) = B \cdot X + C \qquad (B, C \in \mathbb{S}).$$

Therefore, determining the value of A can be broken into 3 subproblems: Evaluate A_j for some appropriately chosen j, determine B, and determine C. This procedure can recursively be applied to evaluate A_j. However, if we now apply this procedure to $A(j, X)$, we end up with the formula $[A(j, X)](j', X')$, where $A[j']$ is not necessarily an ancestor of $A[j]$. That is, the new formula may have 2 scars. After $O(\log n)$ steps, the formula can end up with $O(\log n)$ scars, making the procedure useless.

Brent [br74] solved this problem by allowing only one scar in any formula. In his algorithm, A is initially scarred by a subformula of A (say A_j) of size approximately $|A|/2$. A_j is handled recursively. However, the next scar of A is chosen so that its root is an ancestor of j. Therefore, at any step in the algorithm the subformula being evaluated has at most 1 maximal scar. A straightforward implementation of this technique would require $O(\log^2 n)$ time, since finding successive j's takes $O(\log n)$ time and the algorithm takes $O(\log n)$ rounds.

We modify the pebbling game of §4 to maintain the condition of having only 1 scar. In this new game pebbles have no labels and pebbles can be removed as well as added. The game ends with a win for the pebbler when all inputs of the challenged node are pebbled. Suppose that the pebbler has a strategy such that after every challenge (after some pebbles are possibly removed), each pebbled subformula has at most 1 maximal scar unless both children of the subformula are pebbled. As before, in the first round the pebbler pebbles the root of the input formula and the challenger challenges this node; in each subsequent round the challenger challenges a node pebbled in the current round or rechallenges the node challenged in the previous round. If the pebbler has an r round winning strategy on any play on a given input formula, then the following theorem shows that there is a circuit of depth $2r$ that computes the value of the formula. The resulting circuit family may not be uniform, and later we describe a strategy that can be implemented on a uniform circuit family.

THEOREM 6.1. *Let A be an arithmetic formula, and let the pebbler in the above 2-person game have an r round winning strategy on all plays on A. Then there is a circuit of depth $2r$ that computes the value of A.*

Proof. We prove the more general result that if the pebbler has an r round winning strategy on all plays on a formula A with a scar X at position i, then a circuit of depth $2r$ suffices to compute values B, C such that

$$A(i, X) = B \cdot X + C.$$

The result required in the theorem is simply the special case in which A has no scar; in this case $B = 0$ and C is the value of A.

The proof is by induction on the number of pebble moves. The base case $r = 0$ is straightforward and is omitted. Assume inductively that any formula with a single maximal scar for which the pebbler has an $r - 1$ round winning strategy on all plays can be computed by a circuit of depth $2(r - 1)$. Let A be a formula with a single maximal scar X at position i, and let the pebbler have a winning r round strategy on all plays on this formula. Let $|A| = n$. We now show that there is a circuit of depth $2r$ that computes the value of $A(i, X)$.

In the first round of the game the position n is pebbled and challenged as required. Consider the next move by the pebbler. If this move does not provide a new scar for A, then the pebbler has an $r - 1$ round winning strategy on all plays of A. Hence, A can be evaluated by a circuit of depth $2(r-1)$ by the induction hypothesis, and we are done. If the move does provide a new scar Y at position j for A we distinguish two cases: 1) A has two maximal scars X and Y, and 2) Y is an ancestor of X and hence A continues to have one maximal scar.

Case 1. If X and Y are two distinct maximal scars of A, then by assumption X and Y are the two children of A. Since it is possible for position j to be challenged in the current round, the pebbler has an $r-1$ round winning strategy for any play on the subformula rooted at position j. Hence, by the induction hypothesis, there is a circuit of depth $2(r-1)$ that computes the value of Y. But $A(i, X) = B \cdot X + C$ with $B = 1$ and $C = Y$ if the root of A is an addition node, and $B = Y$ and $C = 0$ if the root of A is a multiplication node. Hence, $A(i, X)$ can be computed by a circuit of depth $2(r-1)$ in this case.

Case 2. The new scar Y is an ancestor of the old scar X (see Fig. 6.1).

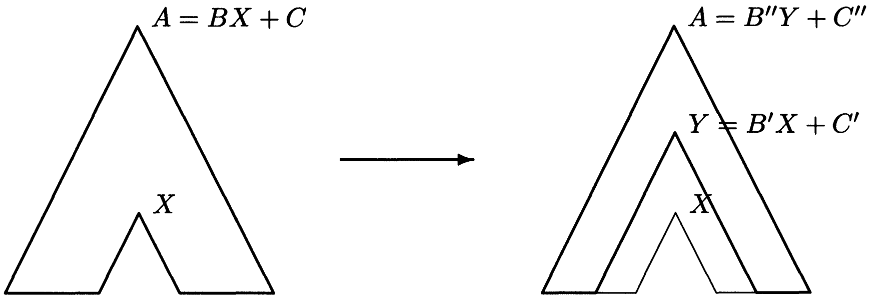

FIG. 6.1. *New scar Y is an ancestor of old scar X.*

As in Case 1, it is possible for position j to be challenged in the current round, hence the pebbler has an $r-1$ round winning strategy for the formula Y with a single maximal scar X. Hence, we have circuits to compute B' and C', each of depth $2(r-1)$, such that $Y(i', X) = B' \cdot X + C'$, where i' is the new position of i in the formula Y. Similarly, it is possible for position n to be rechallenged in the current round, so the pebbler has an $r-1$ round winning strategy for the formula A with a single maximal scar Y. Hence, we have circuits of depth at most $2(r-1)$ to compute B'' and C'' such that $A(j, Y) = B'' \cdot Y + C''$. But $A(i, X) = B''(B' \cdot X + C') + C'' = B \cdot X + C$, giving

$$B = B'' \cdot B' \quad \text{and} \quad C = B'' \cdot C' + C''.$$

Since a circuit of depth 2 computes B and C in terms of B', C', B'', and C'', a depth $2r$ circuit suffices to compute $A(i, X)$.

This completes the induction step, and the theorem is proved. □

We now show how to make the game uniform.

DEFINITION. Let A be a PLOF formula, $|A| = n$. For $i, j < n$, the *least common ancestor* of i and j (denoted *lca*(i, j)) is the common ancestor of i and j with minimum depth. Furthermore, *right*(i) denotes the right child of node i.

In our new game, we add 5 pebbling points to the 4 used in the Boolean game of §4. The object will be to ensure that the challenged formula either is contained in the new interval or has a leftmost scar in the interval. In the original game, the pebbled positions in an interval g were $V_1(g_1)$, $V_2(g_1)$, $V_2(g_2)$, and $V_2(g_3)$. We augment these with

1. $lca(V_1(g_1), V_2(g_1))$ denoted by $Lca_1(g)$.
2. $right(Lca_1(g))$ denoted by $R_1(g)$. This is a pebble position only if $Lca_1(g) \neq V_2(g_1)$.
3. $lca(V_2(g_1), V_2(g_2))$ denoted by $Lca_2(g)$.
4. $right(Lca_2(g))$ denoted by $R_2(g)$. Again, this is a pebble position only if $Lca_2 \neq V_2(g_2)$.
5. The node challenged in the previous round denoted by $last(g)$.

In this game we explicitly include the challenged node from the previous round since it may not be one of the other 8. Figure 6.2 shows one possible placement of pebbles.

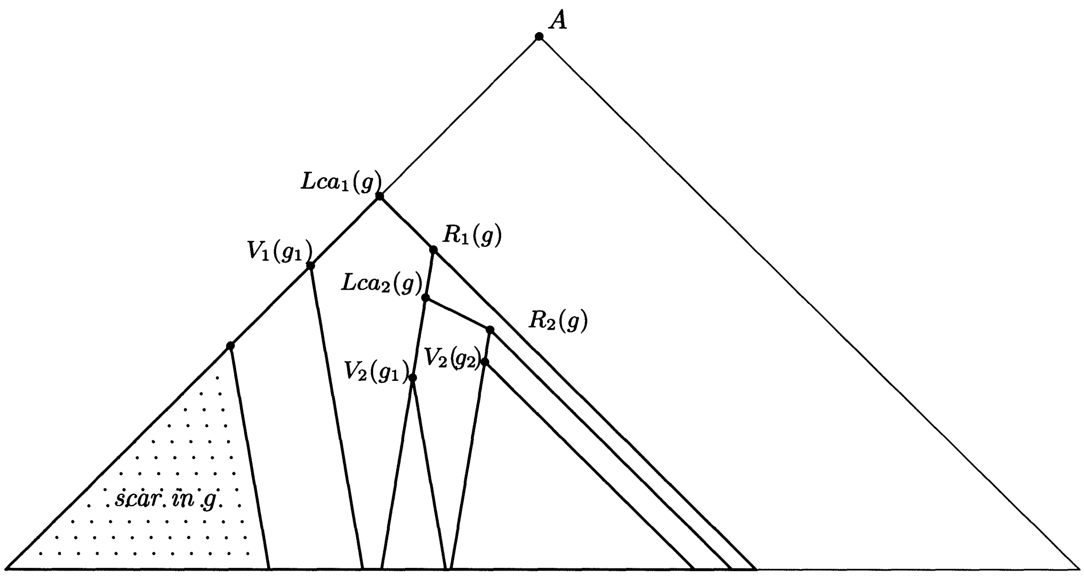

FIG. 6.2. *One possible pebble placement.*

DEFINITION. Let A be a PLOF formula. A_k is a *leftmost* subformula of A if it is a subformula of A such that when A is viewed as a tree, A_k occurs along the leftmost branch. A *leftmost scar* of A is a maximal pebbled leftmost subformula.

Notice that if F is a subformula of a PLOF A and F_k is a leftmost subformula of F, then F_k is an initial segment of F. The following rules ensure that every challenged formula has a single leftmost scar.

Rules of the algebraic game. Let A be an arithmetic formula, $|A| = n$ a power of 2 ($n \geq 2$).

1. In the first round, the `Pebbler` places a pebble on n, and the `Challenger` challenges it. In all subsequent rounds there will be an interval g whose length is a power of 2 and a challenged position c within g. For the next round, $g = A[1, n]$ and $c = n$.
2. For c, the challenged position in in g, if at least one child of c is not pebbled, then let $\nu_1, \cdots, \nu_9$ be the 9 pebble positions defined earlier such that $\nu_1 \leq \cdots \leq \nu_9$. We consider these for pebbling in this order. For each of these 9 candidates ν_i,

the `Pebbler` pebbles ν_i if and only if $\nu_i \trianglelefteq c$ and there is no pebbled l satisfying $\nu_i \trianglelefteq l \triangleleft c$. The `Challenger` challenges one of these new pebble positions. Notice that this allows rechallenging c. After the round ends all newly placed pebbles except the challenged node and any leftmost scar are removed, unless both children of the challenged node are pebbled. The new substring is the leftmost g_j containing the new challenged position.

3. If both children of c are pebbled, then the `Pebbler` wins.

LEMMA 6.2. *Let g be the current interval, and let c be the challenged node. Then either 1) A_c is contained in g or 2) A_c has a leftmost scar in g.*

Proof. We proceed by induction on the round number. Notice that to establish condition 2, it suffices to show that some leftmost subformula of A_c rooted in g is pebbled.

Basis. At round 1, $A = A_c$ satisfies condition 1.

Induction. In general, suppose the lemma holds for an interval g and challenged node c. If the new challenged node c' is in g_1, then g_1 is the new interval and either condition 1 holds for c' and g_1 or condition 2 holds with the same scar as for A_c.

Now suppose c' is in the second half of g_2, so that g_2 is the new interval, and suppose that $A_{c'}$ is not contained in g_2. Assume that the leftmost scar of A_c does not lie in g_2 (since otherwise we are done). Therefore, it lies in the first half of g_1. There are two subcases, depending on whether $A_{c'}$ includes an initial segment of g. If it does, then the leftmost scar of A_c is a leftmost subformula of $A_{c'}$, and $Lca_1(g)$ is either a leftmost scar of $A_{c'}$ in g_2 (so condition 2 holds for c' and g_2) or $Lca_1(g)$ is c', in which case the two children of c' are $V_1(g_1)$ and $R_1(g)$, so the game ends. The second subcase is that $A_{c'}$ is not an initial segment of g (see Fig. 6.3).

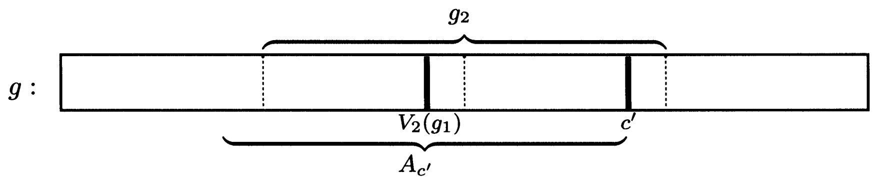

FIG. 6.3. *Subcase in which $A_{c'}$ is not an initial segment of g.*

In that case, $V_2(g_1)$ is a leftmost subformula of $A_{c'}$ because of the PLOF property.

Finally, suppose c' is in the second half of g_3, so that g_3 is the new interval. Furthermore, assume that $A_{c'}$ does not lie entirely in g_3. If the left child of c' is to the left of g_3, then it is in g_1 (since A_c is in g or has a leftmost scar in g), so either $c' = Lca_1(g)$ or $c' = Lca_2(g)$. In either case, both children of c' are pebbled, so the game ends. If the left child of c' is in the left half of g_3, it is $V_2(g_2)$ and condition 2 holds for c' and g_3. The final case occurs when the left child of c' is in the right half of g_3. Then one of $\{Lca_1(g), Lca_2(g), V_2(g_2)$, the leftmost scar of $c\}$ provides a leftmost scar of c' in g_3. □

Lemma 4.6 can easily be adapted to show that in any round of the game, every position $k \triangleleft c$ with k to the left of g has some pebbled l such that $k \trianglelefteq l \triangleleft c$.

LEMMA 6.3. *In the game, the indicated strategy for the* `Pebbler` *wins in $O(\log n)$ rounds.*

Proof. In every round of the game, the interval g is cut in half. □

We must now show that the game can be converted into a uniform log-depth arithmetic-Boolean circuit family.

In the game above, we did not know where any Lca_i, R_i, or $last$ was in the interval g. If one of these positions was challenged, a constant depth circuit could not determine the new interval. We modify the game slightly so that the `Pebbler` must specify an interval when a pebble is placed (corresponding to the interval the pebble is in). This interval can be placed as a label on the pebble. This gives a total of at most 14 different pebble points and labels (since many of the pebble points cannot be in every interval).

We must augment Lemma 4.8 to show that the new pebbling points can be determined in Boolean $\mathbf{NC}^1$. However, although the least common ancestor (and its right child) can be determined once the interval is known, the entire history of the game may be necessary to determine the last challenged position. Let $\nu = \langle p_1, \cdots, p_k \rangle$ ($p_i \in \{1, \cdots, 14\}$) be a sequence that describes the first k moves of the game. Here, p_i denotes the pebble challenged by the `Challenger` in the ith round.

LEMMA 6.4. *The following predicates are in Boolean* $\mathbf{NC}^1$:

a. $Lca_1(A, i, j, k) \overset{\text{def}}{=} (k = Lca_1(A[i, j]))$.

b. $Lca_2(A, i, j, k) \overset{\text{def}}{=} (k = Lca_2(A[i, j]))$.

c. $R_1(A, i, j, k) \overset{\text{def}}{=} (k = R_1(A[i, j]))$.

d. $R_2(A, i, j, k) \overset{\text{def}}{=} (k = R_2(A[i, j]))$.

e. $Last(A, i, j, k, \nu) \overset{\text{def}}{=}$ *k is the challenged node in the previous round of the game.*

Proof.

a., b. By Lemma 4.8, we can determine $V_1(A[i, j])$ and $V_2([A(i, j)])$ in $\mathbf{NC}^1$. The *Descendent* predicate in Lemma 4.8 can be used to check that one of the V_i occurs as a descendent of the left child of k and the other as a right child.

c., d. Determining the right child of a least common ancestor is easy once we can determine the least common ancestor.

e. Let $\nu = \langle p_1, \cdots, p_k \rangle$. In ν, find the largest i such that p_i is not $last$ (say p_l). Find the indicated pebble position in round l. This will be the challenged position in the current round. □

We can use a slightly modified Lemma 4.9 to determine if $\nu = \langle p_1, \cdots, p_k \rangle$ codes a valid sequence of challenges.

Let A^ν denote the scarred subformula challenged after the k indicated challenges, and let $I(\nu)$ be the position of the root of A^ν. Let Ω_ν be the circuit that computes the value of A^ν and α_ν be the value computed by the circuit Ω_ν. To obtain the desired log-depth arithmetic-Boolean circuit over the commutative semi-ring $\mathbb{S}$ we must compute α_ν with a constant-depth circuit using the values $\alpha_{\nu^\frown\langle q \rangle}$ ($q \leq 14$). We break Ω_ν into subcircuits $\Omega_{\nu^\frown\langle q \rangle}$. α_ν will be either a single value in $\mathbb{S}$ or a tuple, depending upon which of the cases in Lemma 6.2 holds. We will describe the circuit for the case where A^ν has a leftmost scar in g. The other case is an easy modification of this. Figure 6.4 is a simplified circuit to compute the value of A^ν for the pebble placements in Fig. 6.2.

Suppose that A^ν has $A^{\nu'}$ as a leftmost scar. Denote by σ_q the value $\alpha_{\nu^\frown\langle q \rangle}$, assuming the correct values (or tuples) at all pebble positions in this round of the game that came before the qth are computed. Now our algorithm is as follows.

Algorithm: *Compute α_ν where A^ν satisfies case* 2 *of Lemma* 6.2

```
$q_{\min}$ := smallest $q$ satisfying $I(\nu') < I(\nu^\frown\langle q \rangle) < I(\nu)$
$q_{\max}$ := largest $q$ satisfying $I(\nu') < I(\nu^\frown\langle q \rangle) \leq I(\nu)$
In parallel, for each $q \in q_{\min}, \cdots, q_{\max}$ compute:  $\alpha_{\nu^\frown\langle q \rangle}$
```

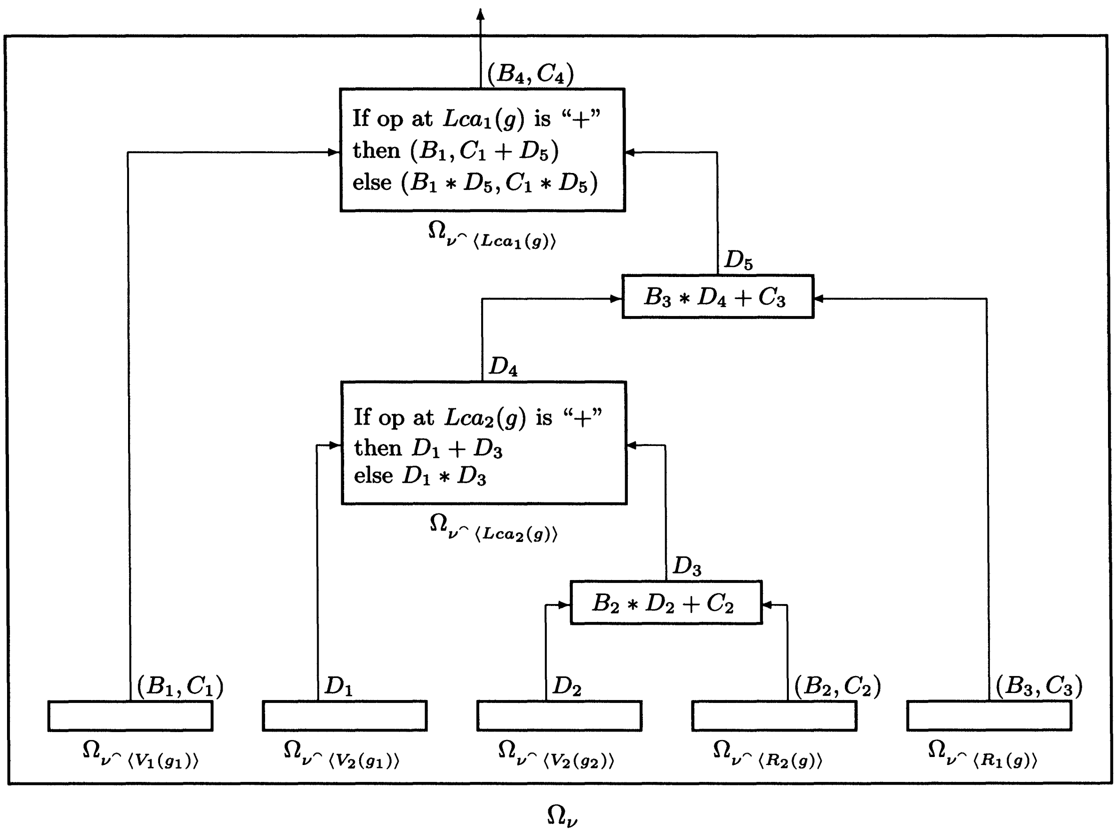
(B_4, C_4)
If op at $Lca_1(g)$ is "+" then $(B_1, C_1 + D_5)$ else $(B_1 * D_5, C_1 * D_5)$
$\Omega_{\nu^\frown\langle Lca_1(g)\rangle}$
D_5
$B_3 * D_4 + C_3$
D_4
If op at $Lca_2(g)$ is "+" then $D_1 + D_3$ else $D_1 * D_3$
$\Omega_{\nu^\frown\langle Lca_2(g)\rangle}$
D_3
$B_2 * D_2 + C_2$
(B_1, C_1)
D_1
D_2
(B_2, C_2)
(B_3, C_3)
$\Omega_{\nu^\frown\langle V_1(g_1)\rangle}$
$\Omega_{\nu^\frown\langle V_2(g_1)\rangle}$
$\Omega_{\nu^\frown\langle V_2(g_2)\rangle}$
$\Omega_{\nu^\frown\langle R_2(g)\rangle}$
$\Omega_{\nu^\frown\langle R_1(g)\rangle}$
Ω_ν

FIG. 6.4. *Circuit for computing A^v for pebble placements in Fig. 6.2.*

```
$\sigma_{\min} := \alpha_{\nu^\frown\langle q_{\min}\rangle}$
For $q = q_{\min} + 1$ to $q_{\max}$
    Let $\mu = \nu^\frown\langle q\rangle$
    If $A^\mu$ satisfies condition 1 of Lemma 6.2 then $\sigma_q := \alpha_\mu$
    else if $A^\mu$ satisfies condition 2 of Lemma 6.2 then
        Let $(B, C) = \alpha_\mu$
        Let $q_1$ be the position of the maximal leftmost pebbled
              subformula of $A^\mu$
        If $\sigma_{q_1}$ is an element of $\mathbb{S}$ then $\sigma_q := B * \sigma_{q_1} + C$
        else Let $(B', C') = \sigma_{q_1}$
            $\sigma_q := (B * B', B * C' + C)$
    else Let $q_1$ and $q_2$ be the pebble placements of the left
              and right operands of $A^\mu$
        Let $\theta$ be the operator at $A^\mu$
        If $\sigma_{q_1}$ is an element of $\mathbb{S}$ then $\sigma_q := \sigma_{q_1}\theta\sigma_{g_2}$
        else Let $(B, C) = \sigma_{q_1}$
                 If $\theta = *$ then $\sigma_q := (B * \sigma_{q_2}, C * \sigma_{q_2})$
                 else $\sigma_q := (B, C + \sigma_{q_2})$
    Endfor
```

The only change that must be made to the algorithm for it to work for the other condition of Lemma 6.2 above is the value of $q_{\min}$.

The technique described above can easily be generalized to solve the problem for fields and (noncommutative) semi-rings. We show the field case first. Suppose $\mathbb{F}$ is a field and A is an arithmetic formula over $\mathbb{F}$. It is easy to show that a scarred formula $A(j, X)$ can be written as a rational affine function:

$$A(j, X) = \frac{B \cdot X + C}{D \cdot X + E} \qquad (B, C, D, E \in \mathbb{F}).$$

It is also easy to verify that these functions are closed under composition. Therefore, the same algorithm as above is used except the value α_v of a subformula with a leftmost scar is represented by a 4-tuple (B, C, D, E).

For the (noncommutative) semi-ring case, we must first convert to PLOF form. However, we have problems if $*$ is not commutative. Therefore, we augment the language to include a "reverse multiplication," denoted $*'$, where $a * b = b *' a$. Any formula can be put in equivalent PLOF form in this augmented language. Now, a scarred formula $A(j, X)$ can be written as

$$A(j, X) = B \cdot X \cdot C + D \qquad (B, C, D \in \mathbb{S}).$$

A subformula $A(j, X)$ is represented by a 3-tuple (B, C, D). Again, composition is easy to do. Therefore, the semi-ring algorithm can be used except that a little care is necessary in keeping the left and right multipliers separate.

Finally, we present a simpler method for solving the evaluation problem when the algebra is a ring. Suppose we wish to evaluate the scarred formula $A(j, X) = B \cdot X + C$. Then, $A(j, 0) = C$ and $A(j, 1) = B + C$. From this system of equations we can easily determine both B and C. Therefore, the problem of determining A is broken into three subproblems: Evaluate the formula rooted at X, evaluate $A(j, 0)$, and evaluate $A(j, 1)$. These problems can be recursively solved.

Acknowledgments. We thank Martin Tompa for helping with the exposition in §4. We also thank the referees for their careful reading of the manuscript and many useful suggestions.

REFERENCES

[bis88] D. Barrington, N. Immerman, and H. Straubing, *On uniformity within* NC^1, in IEEE Structures in Complexity Theory, IEEE Computer Society, Washington, DC, 1988, pp. 47–59. Revised version to appear in J. Comput. System Sci.

[bo77] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[br74] R. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[bu87] S. Buss, *The Boolean formula value problem is in* ALOGTIME, in Proc. 19th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 123–131.

[cks81] A. Chandra, D. Kozen, and L. Stockmeyer, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[co85] S. Cook, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985) pp. 2–22.

[csv82] A. Chandra, L. Stockmeyer, and U. Vishkin, *Complexity theory for unbounded fan-in parallelism*, in Proc. 23rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1982, pp. 1–13.

[dy88] P. Dymond, *Input-driven languages are in* log n *depth*, Information Process. Lett., 26 (1988), pp. 247–250.

[dt85] P. Dymond and M. Tompa, *Speedups of deterministic machines by synchronous parallel machines*, J. Comput. System Sci., 30 (1985), pp. 149–161.

[gu85] A. Gupta, *A fast parallel algorithm for recognition of parenthesis languages*, Master's Thesis, University of Toronto, Canada, 1985.

[im89] N. Immerman, *Expressibility and parallel complexity*, SIAM J. Comput., 8 (1989), pp. 625–638.

[js82] M. Jerrum and M. Snir, *Some exact complexity results for straight-line computations over semirings*, J. Assoc. Comput. Mach., 29 (1982), pp. 874–897.

[kr90] R. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, Elsevier, New York, 1990, pp. 869–941.

[lsh65] P. Lewis, R. Stearns, and J. Hartmanis, *Memory bounds for recognition of context-free and context-sensitive languages*, in Proc. 6th Annual IEEE Symposium on Switching Circuit Theory and Logical Design, IEEE Computer Society, Washington, DC, 1965, pp. 191–202.

[ly77] N. Lynch, *Log space recognition and translation of parenthesis languages*, J. Assoc. Comput. Mach., 24 (1977), pp. 583–590.

[mr85] G. Miller and J. Reif, *Parallel tree contraction and its application*, in Proc. 26th Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 478–489.

[mp88] D. E. Muller and F. Preparata, *Parallel restructuring and evaluation of expressions*, Tech. Report UILU-ENG-88-2253 ACT-101, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1988.

[ra86] V. Ramachandran, *Restructuring formula trees*, unpublished manuscript, May 1986.

[ru81] W. Ruzzo, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[sa76] J. Savage, *The Complexity of Computing*, Wiley–Interscience, Toronto, 1976.

[si83] M. Sipser, *Borel sets and circuit complexity*, in Proc. 15th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 61–69.

[sp71] P. Spira, *On time hardware complexity tradeoffs for Boolean functions*, in Proc. 4th Hawaii International Symposium on System Sciences, 1971, pp. 525–527.

[to85] M. Tompa, *A pebble game that models alternation*, unpublished manuscript, 1985.

[jg86] J. von zur Gathen, *Parallel arithmetic computations: a survey*, in Proc. 12th International Symposium on Mathematical Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986.