Information Processing Letters 26 (1987/88) 247–250
North-Holland

11 January 1988

INPUT-DRIVEN LANGUAGES ARE IN LOG n DEPTH

Patrick W. DYMOND *

Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093, U.S.A.

Communicated by E.C.R. Hehner
Received 22 April 1987

Keywords: Parallel computation, NC^1, log depth circuit, formula value problem, input-driven language, context-free language

A context-free language $\mathcal{L}$ is *input-driven* if the stack motion of a pda for $\mathcal{L}$ (i.e., wether it pushes of pops) can be determined from the current input symbol alone and is independent of the current state and stack symbol. Input-driven languages (idl's) have been studied by Von Braunmühl and Verbeek [11], who showed that every idl can be recognized in log n space. Here, their result is refined by showing that each idl has a recognition algorithm which can be implemented on a Boolean circuit of *depth* log n. [1] Because our circuits have bounded fan-in, this is an optimal result up to constant factors. In fact, the result can be generalized to a larger class of languages than idl's. We define an f(n)-*predictable language* to be any context-free language which can be recognized on a pda without ϵ-moves whose stack height after processing the *ith* input symbol can be computed as a function of the input and i by a circuit family of depth f(n). (In the case of nondeterministic pda's, it suffices for the function to produce height values corresponding consistently to any valid (accepting) computation.) It follows from the proof given here that any log n-predictable language is in log n depth. Straightforward examples of log n-predictable languages which are not idl's are

$$\{a^i b^j c^k \mid i = j \text{ or } j = k\}$$

and

$$\{ww^r \mid w \in \{a, b\}^*\}.$$

The log depth idl recognition algorithm presented here hinges essentially on the technique of Buss [1], who improved Lynch's log space algorithm for Boolean formula evaluation [6] to log depth. Buss also describes a log n depth recognition algorithm for parenthesis languages; this paper generalizes both results, since parenthesis languages are a proper subclass of idl's and the set of fully parenthesized true Boolean formulas is a parenthesis language. Previous to Buss [1], the best depth for parenthesis language recognition was log n log log n due to Cook and Gupta [4] and the best space bound was log n due to Mehlhorn [7].

In extending his result to parenthesis languages, Buss generalized his Boolean formula evaluation technique to expressions involving k-ary functions over a finite domain. To make use of this input-driven languages, we follow the approach taken by Rytter [9] and earlier Mehlhorn [7] in transforming recognition to the problem of efficiently

* Research was sponsored by the National Science Foundation under Grant No. DCR-8604031.

[1] We say that a language can be recognized in log n depth if there is a (uniform) family of Boolean circuits recognizing the language such that the circuit for inputs of length n has depth O(log n). The multiplicative constant implied by the "O" notation depends on the language.

evaluating an expression whose values are binary relations on a finite set, and whose operations are functional compositions and certain unary operations determined by the inputs. However, to use Buss's evaluation procedure, the expression must be in postfix form with longer operand on the left, the transformation from the input word to the expression must be computable in log n depth, and the operations should be unary or binary (whereas the construction in [9] uses functions of unbounded arity). The purpose of this paper is to note that Rytter's and Buss's constructions can be combined to obtain both the idl result and the extension to log n-predictable languages mentioned above.

In describing the expression to be evaluated it is useful at first to simplify the class of languages under consideration. Let $\mathscr{L} \subseteq \Sigma^*$ be a deterministic input-driven cfl, and let $\mathscr{P}$ be a deterministic pda recognizing $\mathscr{L}$ with the following properties:

- There are no ϵ-transitions, i.e., each transition consumes an input symbol.
- The state set is Q, the stack alphabet is Γ, the start state is q_0, the starting pushdown symbol is #, and $\mathscr{P}$ accepts by reading all the input and ending in a configuration with a single symbol on the stack in state q_a.
- $\mathscr{P}$ is *input-driven*: the input alphabet Σ can be partitioned into sets Σ_0, Σ_1, Σ_2 such that δ: $Q \times \Gamma \times \Sigma_i \mapsto Q \times \Gamma^i$, i.e., when making a transition depending on a symbol in Σ_i, $\mathscr{P}$ pops the topmost stack symbol and replaces it with a string from Γ^i, causing the pushdown stack height either to decrease or increase by one or to stay the same.
- There is a function *height*(i, w) which for any input w and any $0 \leqslant i \leqslant |w|$ gives the height of $\mathscr{P}$'s pushdown store after processing the first i symbols of w. Furthermore, this function can be computed in log n depth. The existence of such a function is implied by the previous assumption that $\mathscr{P}$ is input-driven (and this is the only time that assumption is used).

A *reduced configuration* (r.c.) of $\mathscr{P}$ is a pair $(q, \gamma) \in Q \times \Gamma$. Relative to a particular input $w \in \Sigma^*$, $|w| = n$, we define the (i, j)-*relations* $\Rightarrow^{i,j}$ for $1 \leqslant i \leqslant j \leqslant n$ as binary relations on r.c.'s as follows:

- If *height*(i − 1, w) ≠ *height*(j, w) or, for some ℓ, $i \leqslant \ell < j$, *height*(ℓ, w) < *height*(j, w), then $\Rightarrow^{i,j}$ is empty.
- Otherwise, $(q_1, \gamma_1) \Rightarrow^{i,j} (q_2, \gamma_2)$ if $\mathscr{P}$, when run on input $w_i \ldots w_j$ starting in state q_1 with γ_1 as a single element stack, reaches state q_2 with γ_2 as the only stack element after reading all of $w_i \ldots w_j$.

The recognition of w is reduced to an evaluation of an expression whose subexpressions evaluate to (i,j)-relations, and whose value is $\Rightarrow^{1,n}$ (since $w \in \mathscr{L}$ iff $(q_0, \#) \Rightarrow^{1,n} (q_a, \gamma)$ for some γ). There are two basic kinds of operations on the relations. Relational composition can be applied in the case of three indices $i \leqslant j \leqslant k$ provided *height*(i − 1) = *height*(j − 1) = *height*(k) (and no intermediate heights are smaller than k): in this case, $\Rightarrow^{i,k}$ can be expressed as $\Rightarrow^{i,j-1} \circ \Rightarrow^{j,k}$. The second kind of operation on relations is a unary operation determined by two input symbols. For i < j, the pair of symbols from the input alphabet at positions i and j can be applied to $\Rightarrow^{i+1,j-1}$ to compute $\Rightarrow^{i,j}$, provided that w_i causes a push, w_j causes a pop, *height*(i − 1, w) = *height*(j, w), and the intervening stack heights are always greater than *height*(j, w) is.[2] The result of the operation should of course be the relation $\Rightarrow^{i,j}$, which satisfies $(q_1, \gamma_1) \Rightarrow^{i,j} (q_2, \gamma_2)$ iff (q_1, γ_1) pushes $\gamma_3\gamma_2$, entering some state q′ where $(q', \gamma_3) \Rightarrow^{i+1,j-1} (q'', \gamma_4)$, and $\mathscr{P}$ in state q″ with γ_4 on the stack and w_j in the input pops its stack and enters state q_2. The situation i = j is a special case, specifying a particular relation determined by the effect of processing a single symbol of Σ and not changing the stack height. We represent this constant relation with the notation $\Rightarrow^{\sigma}$ where $\sigma = w_i$. In the more general case (j > i), we call an operation of this kind a w_i, w_j-operation: note that every pair of symbols from the input alphabet determines a separate unary operation on relations. (By this we mean to emphasize that the operation itself is over a finite domain, and does not depend on the input except insofar as deciding wether or not it can be legally applied at a particu-

[2] If j = i + 1, then $\Rightarrow^{i+1,j-1}$ should be interpreted as the identity relation.

lar position.) We can represent application of the w_i, w_j-operation described above using infix notation: $\Rightarrow^{i,j}$ is the value of the expression ($w_i \Rightarrow^{i+1,j-1} w_j$). If $i + 1 = j$, the middle symbol representing the identity relation will be omitted by convention. Expressions formed from constant relations, compositions and w_i, w_j-operations are called *relational expressions.*

It is now easy to see how the input determines a relational expression whose value is $\Rightarrow^{1,n}$. We describe a series of reductions from the input to a relational expression achieving this. First, the w_i, w_j-operations can be identified and inserted. A symbol of the input word w at position i should be preceded by an opening parenthesis if $height(i, w) > height(i - 1, w)$, followed by a closing parenthesis if $height(i, w) < height(i - 1, w)$, and replaced by $(\Rightarrow^{\sigma})$ if $\sigma = w_i$ and the two heights are equal. Next, composition is handled by insering $\circ$ wherever an opening parenthesis follows a closing one.

The resulting expression is not fully parenthesized, due to the possibility of an unbounded number of composition operators at the same level in a subexpression. It is necessary to insert extra parentheses so that (say) left-to-right associativity of $\circ$ is achieved. The associativity of functional composition permits us to group the subexpressions this way.

Buss's algorithm makes use of the fact that his formulas are written using postfix notation in which the longer of the two operands appears first. We achieve this for relational expressions as follows: For obtaining postfix format there is no difficulty with the unary operations (w_i can be moved and stored as a composite symbol with its corresponding w_j). The binary composition operation can be converted to postfix form in a manner similar to that of Buss [1]. However, his operations were commutative while composition is not, and so we must specify what to do with a composition where the left operand is shorter than the right one. In this case we reverse the operands, and replace the composition operator $\circ$ with a *commuted composition* operator $\bullet$, defined such that $g \bullet f = f \circ g$. This new operation is still a well defined operator on a finite domain.

Below we describe how the above reduction can be carried out in $O(\log n)$ depth, after which Buss's algorithm can be applied in an additional $O(\log n)$ depth. A useful alternative characterization of $\log n$ depth has been provided by Ruzzo [8], who showed that any language accepted in $O(\log n)$ time on an alternating Turing machine [2] can be recognized in $\log n$ depth. Similarly, if f is a function such that the *ith* bit of f's output on inputs of length n can be computed in $O(\log n)$ time on an alternating Turing machine (in this case we say the ATM computes f in log time), then f can be computed by log depth Boolean circuits.

Alternation can be viewed as a two-person game, and Buss's algorithm can be viewed as implementing a kind of two-person pebbling game [3] and [10] played on a formula. One player, the *pebbler*, asserts values for certain subexpressions of the formula. The other player, the *challenger*, tries to catch the pebbler in an incorrect assertion, or failing that, to delay the end of the game as long as possible. The game ends when a value challenged in the previous round can immediately be determined from previously pebbled values and the input. The pebbler wins if the value he asserted is consistent. Buss found ways to add further structure to such a game so that the winner could be determined in log depth. We do not explain his algorithm here. Instead we note that, for our purposes, his algorithm can be modified so that the assertions made are of finite relations rather than Boolean values, and are verified in accordance with the combinational rules determined by the pda as described above, rather than Boolean ones.

It remains to describe how a log time alternating Turing machine (and, hence, a log depth circuit) given the input word w can compute the expression described above. Buss describes a subroutine *count* which can be used by a log time ATM to count the number of positions of the input with a distinguished symbol. This can be generalized to a log time subroutine counting the number of positions satisfying any predicate which can itself be computed in log time on an ATM.

Using a log depth circuit, the stack height can be computed for every position of the input. To find matching positions for w_i, w_j-operations, the height values determine whether the match is to the right or left and the matching character can be found by looking in that direction for the next position of matching height. Then, the second of the matching symbols can be replaced by the composite symbol mentioned above, and the first replaced by a blank. (Blanks are easily eliminated using *count*.)

It is easy to compute the atomic values for $\Rightarrow^{w_i}$ where needed, and insert the composition operators. To fully parenthesize the expression, we can use the fact that the number of composition operators to the right of a given position which will require the introduction of an opening parenthesis to the left of that position can be counted using the predicate "*this composition operator combines relations for subcomputations with stacks beginning and ending at the same or lower height, and there has been no intervening composition operator of still lower height*". After the correct number of opening parentheses are inserted, adding a right parenthesis before every composition operator fully parenthesizes the expression. Redundant parentheses and blanks are easily removed via another log depth reduction. At this point, the expression can be converted to postfix format with longest operand first, replacing composition with commuted composition where necessary. The final step is the application of Buss's algorithm, modified as noted above.

Once the above construction is understood, it should not be difficult to see how the generalizations to nondeterministic pda's and to log n-predictable languages can be obtained, using the same technique. This result shows that a very large class of cfl's is in log depth, and it is natural to inquire about the possibility that all deterministic cfl's, which are already known to be in log space, might also be in log depth. One dcfl in log space which is not currently known to be recognizable in log depth is the two-sided Dyck language on four letters [5].

Acknowledgment

My thanks go to S.A. Cook and A. Gupta for helpful explanations of Buss's algorithm in terms of two-person pebbling games, and to N. Rhodes for useful discussions about input-driven languages. This work was done while visiting the Department of Computer Science, University of Toronto, Canada.

References

[1] S.R. Buss, The boolean formula value problem is in ALOGTIME, in: Proc. 19th Ann. ACM Symp. on Theory of Computation, 1987, to appear.

[2] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer, Alternation, J. ACM 28 (1981) 114–133.

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

[4] A. Gupta, A Fast Parallel Algorithm for Recognition of Parenthesis Languages, Tech. Rept. 182/85, Dept. of Computer Science, Univ. of Toronto, January 1985.

[5] R.J. Lipton and Y. Zalcstein, Word problems solvable in log space, J. ACM 24 (1977) 522–526.

[6] N. Lynch, Log space recognition and translation of parenthesis languages, J. ACM 24 (1977) 583–590.

[7] K. Mehlhorn, Bracket languages are recognizable in log n space, Inform. Process. Lett. 5 (1976) 169–170.

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

[9] W. Rytter, An application of Mehlhorn's algorithm for bracket languages to log n space recognition of input-driven languages, Inform. Process. Lett. 23 (1986) 81–84.

[10] H. Venkateshwaran and M. Tompa, A new pebble game that characterizes parallel complexity classes, in: Proc. 27th Ann. IEEE Symp. on Foundations of Computer Science (1986) 348–360.

[11] B. Von Braunmühl and R. Verbeek, Input-driven languages are recognized in log n space, in: Proc. FCT Conf., Lecture Notes in Computer Science, Vol. 158 (Springer, Berlin, 1983) 40–51.