THE MEMBERSHIP QUESTION FOR ETOL-LANGUAGES IS POLYNOMIALLY COMPLETE*

Jan Van LEEUWEN
Mathematisch Centrum, 2e Boerhaave Straat 49, Amsterdam 1005, the Netherlands

Received 22 August 1974
Revised version received 24 April 1975

complexity of computations ETOL languages

1. Introduction

Cook [3] showed four years ago that problems with an easy, polynomial-time solution on a non-deterministic Turing-machine may be extremely hard to solve in polynomial time on a deterministic device. Karp [7] gave a long list of combinatorial problems which are all "complete" in the sense that when one of them is do-able in deterministic polynomial time then all of them are and P = NP.

Shamir has looked for NP-complete problems which are membership questions for as narrow a language family F as possible. Shamir and Beeri [14] proved that when F is equal to the family of 1-way chacking-stack automaton languages (a proper subset of the indexed languages) or equal to the λ-free context-free programmed languages then NP-completeness can already be embedded in F.

In this paper we solve a question of Shamir posed at the 2nd Colloquium on Automata, Languages and Programming (Saarbrücken, 1974) how one can embed NP-completeness in families of simple developmental languages, and in particular whether or not one can do it in ETOL-languages. We shall answer the question affirmatively and obtain therefore a language family quite "low" in the Chomsky-hierarchy whose membership question is NP-complete.

Since we shall also prove that all ETOL-languages (possibly up to the empty word) can be generated by a λ-free unconditional transfer contect-free programmed grammar, it immediately follows that we can strenghthen some results of Shamir and Beeri [14] and conclude that NP-completeness can already be embedded into the λ-free unconditional transfer context-free programmed languages.

* This work has been partly supported by NSF grant GJ 998 and by NATO grant 574.

2. Preliminaries

Most notations and terminology will be as in standard texts on language theory like Aho and Ullman [1], or Salomaa [13]. It was only recently that parallel rewriting systems enjoyed increased attention and we shall have to mention some of the pertinent definitions. The area where we shall draw some results from is nicely surveyed in Herman and Rozenberg [5] and we shall therefore only present the concepts that are strictly needed here.

The idea behind ETOL-grammars is to have a finite set of tables with production-rules (or: substitutions if you like) which are iterated in some order or another on an initial string and applied as parallel replacement operations. Only the words over a designated terminal alphabet which we can derive are counted and collected into a language.

Definition. An ETOL-grammar is a 4-tuple $G = \langle V, \Sigma, \{\delta_1, ..., \delta_k\}, S\rangle$, where V, Σ, and S are as usual and $\delta_1, ..., \delta_k$ are finite substitutions from V into V^*. When all δ_i's are λ-free then G is called λ-free.

Definition. A language L is called an ETOL-language iff there is some ETOL-grammar $G = \langle V, \Sigma, \{\delta_1, ..., \delta_k\}, S\rangle$ such that $L = \bigcup_n \delta_{i_n} ... \delta_{i_1}(S) \cap \Sigma^*$, the union taken over all sequences $\delta_{i_n} ... \delta_{i_1}$ of substitutions from G.

The family of ETOL-languages(called ETOL hereafter) has nice properties, for instance one can always effectively find a λ-free ETOL-grammar for each ETOL-language. One can easily show that ETOL properly contains the context-free languages [5] but is itself no more powerful than the indexed languages [4]. Using rather involved arguments it was recently even shown that ETOL is strictly included in the family of indexed languages [6].

One can easily simulate ETOL-grammars with context-free programmed grammars but the result can be made stronger. Recall that in cfpg-grammars all productions are uniquely identifiable by some label r and look like

$$(r)A \to w \quad S(D_1)\ F(D_1),$$

where D_1 and D_2 are the "success" and "failure" domains. If in all productions D_1 and D_2 are equal, then the cfpg-grammar is called "unconditional transfer" ([8], p. 125), and such grammars define a quite restricted subclass of the cfpg-languages.

Theorem 2.1. All ETOL-languages are λ-free unconditional transfer context-free programmed languages.

Proof. (We should in fact be more precise and always exclude the empty word.) Let L be an ETOL-language and G a λ-free ETOL-grammar generating L. Let $V = \{\sigma_1, ..., \sigma_n\}$, and introduce new symbols σ' and σ'' for each $\sigma \in V$. We construct an utcfpg-grammar whose terminal alphabet is still Σ but which uses all other symbols as variables and in addition requires a new symbol T for starting. Each table δ shall be written out line by line in two versions:

δ', in which we make all productions $a \to w$ in δ into $a' \to w''$

δ'', in which we make all productions $a \to w$ in δ into $a'' \to w'$.

Also, there will be blocks CHECK′ for ′–symbols and CHECK″ for ″–symbols, and similar blocks TERM′ and TERM″.

When we list the name of a block in a success or failure field we actually mean all names of the productions in that block. $\not\zeta$ will be used as a standard nonterminal symbol not equal to any of before, and is written when we need to escape a wrong production-sequence.

Here is how the utcfpg-grammar looks like. The idea is that we alternately have the intermediate string in ′– and ″–symbols, simulate an arbitrarily selected table by rewriting ′– (or ″–) symbols into ″– (or ′–) symbols and always check at each step that we really converted ALL ′– (or ″–) symbols. Instead of selecting a next table one may choose to print-out the result and hope that it is built from only terminals.

– (1) $T \to S'\ S(\delta'_1, ..., \delta'_k, \text{TERM}')\ F(\delta'_1, ..., \delta'_k, \text{TERM}')$

– for each δ in G there is a block δ' that looks like

$$\left[\begin{array}{l} \vdots \\ (\ )\ \sigma' \to w''\ S(\delta', \text{CHECK}'1)\ F(\delta', \text{CHECK}'1), \text{ for all } \sigma \to w \in \delta \\ \vdots \end{array}\right.$$

– blocks δ'' are defined similarly with ′ replacing ″ and vice versa

– in the block CHECK′ productions are numbered 1, ..., n and are defined as follows

$$\left[\begin{array}{l} \vdots \\ (i)\ \sigma'_i \to \not\zeta\ S(\text{CHECK}'i+1)\ F(\text{CHECK}'i+1) \\ \vdots \\ (n)\ \sigma'_n \to \not\zeta\ S(\delta''_1, ..., \delta''_k, \text{TERM}'')\ F(\delta''_1, ..., \delta''_k, \text{TERM}'') \end{array}\right.$$

– the block CHECK″ is as CHECK′ with obvious modifications

– the block TERM′ looks as

$$\left[\begin{array}{l} \vdots \\ (\ )\ \sigma'_i \to \sigma_i\ S(\text{TERM}')\ F(\text{TERM}') \\ \vdots \end{array}\right.$$

– and finally, TERM″ is similar to TERM′, this time reading out ″–symbols.

The terminal words which this λ-free grammar generates are exactly all the words from L, possibly excluding the empty word. □

One can show that ETOL is in fact properly included in the λ-free unconditional transfer context-free programmed languages [12].

Although the membership question for ETOL-languages is easily seen to be algorithmically solvable, the fundamental problem we address is how hard it is in terms of time complexity. In particular, we want to know about procedures for this job which are possible polynomial-time bounded since that, as is nowadays widely accepted, would be a good standard for feasibility [2]. We shall essentially prove that there

is no hope for such an algorithm by relating the problem to the P = NP question.

Recall that we always work with some encoding of a problem that can be written on a Turing-machine tape. L is *p*-reducible to M iff there exists a polynomial-time computable transformation f such that $x \in L \Leftrightarrow f(x) \in M$. In somewhat informal way we formulate

Theorem 2.2. (ref. [3]). All NP-problems are *p*-reducible to the satisfiability problem for conjunctive normal form propositional formulae with at most 3 literals per clause.

There is a coding problem since Cook [3] used binary notation in the reduction-procedure while we need a unary version of the satisfiability problem. As Shamir and Beeri [14] point out, the mere fact of polynomial boundedness makes however that one can equally give a *p*-reduction to the satisfiability problem in unary code and Cook's theorem is representation independent. Let SAT denote the satisfiability-problem referred to in 2.2.

Definition. A problem (or language) L is called NP-complete if and only if $L \in$ NP and SAT is *p*-reducible to L.

Practically from the definition follows that P = NP iff there is an NP-complete problem that can be done in deterministic polynomial time.

We shall finally call a problem (which is not necessarily in NP) NP-hard if SAT is *p*-reducible to it. (See ref. [8] for a documented discussion on the terminology).

3. The membership question for ET0L-languages

We shall prove that the P = NP question is equivalent to the membership problem for languages generated by such a simple device as the iterated context independent substitution imminent to ET0L-grammars.

Theorem 3.1. The membership-question for λ-free ET0L-languages is NP-complete.

Proof. First we argue that membership for ET0L-languages is an NP-problem. All ET0L-languages have an essentially λ-free generating ET0L-grammar. In the derivation $S = w_0 \Rightarrow \cdots \Rightarrow w_n = w$ of a word one can distinguish stationary levels (when $|w_i| = |w_{i-1}|$) and splitting levels (when $|w_i| > |w_{i-1}|$). By a periodicity-argument (as in [15]) there must be a constant c such that no sequence of consecutive stationary levels need be longer than c. Hence all words have a derivation linearly bounded in their length. An NP-algorithm for deciding membership would first guess the sequence of tables and then evaluate it starting from S: when the finally produced string matches the input we are done, otherwise we reject. (Evaluating the substitutions is essentially an $O(n^2)$ task.)

Now the harder part, we have to embed SAT in an ET0L-language. Like Shamir and Beeri [14] we use unary notation but it is better that we set it up from the beginning. The formulas we consider look like

$$E_1 \wedge \ldots \wedge E_k$$

where each E_i is of the form $(A_{i_1} \vee A_{i_2} \vee A_{i_3})$. The A_{i_j} are either some A or some $\neg A$, and all A's like that occurring in the formula are referred to as literals†).

Each literal will be uniquely encoded into a sequence of ones, but we leave the other symbols (logical connectives and brackets) as they are. As an example

$$(A \vee B \vee C) \wedge (\neg A \vee B \vee \neg C) \wedge \ldots$$

will be encoded as

$$(1 \vee 11 \vee 111) \wedge (\neg 1 \vee 11 \vee \neg 111) \wedge \ldots$$

an encoding which, by the way, is clearly do-able in polynomial time.

We shall make an ET0L-language precisely containing all formulae that are satisfiable, but with one further degree of freedom. Instead of an "initial segment" code (as in the given example), the language shall in fact contain ALL possible, unary encoded representations, but then certainly the "decent" ones.

The idea behind the ET0L-grammar that we give is to build the logical evaluation-trees upside down, that is, we first build the regular set of well-formed and satisfiable formulae with each literal replaced by the explicit truth assignment that makes the formula "true" and then write in code for real literals in conflict-free manner.

Let the axiom be a symbol Π ("true"), and use T ("true") and F ("false") for assigned values.

† It is easy to modify 2.2. so as to require that in the expressions in SAT there indeed occur precisely three (not necessarily distinct) literals.

The first table has productions:

$$\begin{cases} \Pi \to (T \vee T \vee T) \wedge \Pi \\ \Pi \to (T \vee T \vee T) \\ \Pi \to (T \vee T \vee F) \wedge \Pi \\ \Pi \to (T \vee T \vee F) \\ \Pi \to (T \vee F \vee T) \wedge \Pi \\ \Pi \to (T \vee F \vee T) \\ \Pi \to (T \vee F \vee F) \wedge \Pi \\ \Pi \to (T \vee F \vee F) \\ \Pi \to (F \vee T \vee T) \wedge \Pi \\ \Pi \to (F \vee T \vee T) \\ \Pi \to (F \vee F \vee T) \wedge \Pi \\ \Pi \to (F \vee F \vee T) \\ \Pi \to (F \vee T \vee F) \wedge \Pi \\ \Pi \to (F \vee T \vee F) \\ \sigma \to \sigma \text{ for all other symbols.} \end{cases}$$

In all further tables we include a production $\Pi \to \not\in$ (the only one for Π, with $\not\in$ a non-terminal which can never lead to a terminal), so we are forced to iterate this table until we got rid of the Π. We necessarily end up with a well-formed, "true" assignment and prepare for replacing T's and F's by occurrences of (negations of) literals.

We shall have to distinguish quite specifically between variables which occur as "A" and get the value "T" (or "F") and which occur as "$\neg A$" and get the value "T" (or "F"), because that turns out to be a main source of conflicts. We shall use symbols [., true], [., false], [$\neg$, true], and [$\neg$, false] to keep track of it respectively.

The next table then will be

$$\begin{cases} T \to [., \text{true}] \\ T \to \neg[\neg, \text{true}] \\ F \to [., \text{false}] \\ F \to \neg[\neg, \text{false}] \\ \Pi \to \not\in \\ \sigma \to \sigma \text{ for all other symbols.} \end{cases}$$

Since T and F then disappear entirely we can list identity productions for them in all subsequent tables.

We can now expand the unary representations with the table

$$\begin{cases} [., \text{true}] \to 1[., \text{true}] \\ [\neg, \text{true}] \to 1[\neg, \text{true}] \\ [., \text{false}] \to 1[., \text{false}] \\ [\neg, \text{false}] \to 1[\neg, \text{false}] \\ \Pi \to \not\in \\ \sigma \to \sigma \text{ for all other symbols} \end{cases}$$

or decide to terminate some of them. In that case we may terminate both a "normal" variable and its negation (if it occurs) and we need two tables to avoid conflicts.

$$\begin{cases} [., \text{true}] \to 1 \\ [\neg, \text{false}] \to 1 \\ [., \text{true}] \to 1[., \text{true}] \\ [\neg, \text{true}] \to 1[\neg, \text{true}] \\ [., \text{false}] \to 1[., \text{false}] \\ [\neg, \text{false}] \to 1[\neg, \text{false}] \\ \Pi \to \not\in \\ \sigma \to \sigma \text{ for all other symbols} \end{cases}$$

and

$$\begin{cases} [\neg, \text{true}] \to 1 \\ [., \text{false}] \to 1 \\ [., \text{true}] \to 1[., \text{true}] \\ [\neg, \text{true}] \to 1[\neg, \text{true}] \\ [., \text{false}] \to 1[., \text{false}] \\ [\neg, \text{false}] \to 1[\neg, \text{false}] \\ \Pi \to \xi \\ \sigma \to \sigma \text{ for all other symbols .} \end{cases}$$

Note that now the third table is superfluous and may be omitted since the effect it has can always be attained with the fourth or the fifth table, and we leave it as an exercise to the reader to condense the number of tables needed even further down to 2.

If we let $\Sigma = \{-, \vee, \wedge,), (, 1\}$, then all terminal strings generated by the gammar are clearly just all the unary encoded instances of SAT and the theorem follows. □

Considering the given proof, we observe that there never are sentential forms which in the coding that we used represent non-satisfiable formulae (they are either of the wrong form or over the desired terminal alphabet). We may then actually conclude NP-completeness for an even narrower class, the T0L-languages (see [11] or [5]) which are exactly the sentential form languages of ET0L-grammars.

Corollary 3.2. The membership-question for (λ-free) T0L-languages is NP-complete.

We can make further conclusions and relate our results to recent observations of Shamir and Beeri [14].

Corollary 3.3. The membership question for λ-free unconditional transfer context-free programmed languages is NP-hard.

By a further argument one can prove that the membership question for λ-free unconditional transfer context-free programmed grammars is solvable in nondeterministic polynomial time by showing that words need never have a derivation longer than polynomial in their length [15], and it follows that in the λ-free case one may improve 3.3. to NP-completeness.

Acknowledgement

I wish to thank Arto Salomaa, Eli Shamir and Amir Pnueli, and Jaroslav Opatrny for useful comments.

References

[1] A.V. Aho and J.D. Ullman, The theory of parsing, translating and compiling, vol. I (Prentice Hall, Englewood Cliffs, N.J., 1972).
[2] A. Cobham, The intrinsic computational difficulty of functions, in: Y. Bar-hillel (ed.): Proc. 1964 Intern. Congress on Logic, Methodology, and Philosophy of Science (North-Holland Publ. Comp., Amsterdam. 1965), 24–30.
[3] S.A. Cook, The complexity of theorem-proving procedures, Proc. 3rd Annual ACM Symp. on Theory of Computing (1971) 151–158.
[4] K. Culik II, On some families of languages related to developmental systems, Techn. Rep. CS-73-12, Dept. of Appl. Analysis and Computer Science, Waterloo, Canada (1973).
[5] G.T. Herman and G. Rozenberg, Developmental systems and languages (North-Holland Publ. Comp., Amsterdam, 1975).
[6] A. Ehrenfeucht, G. Rozenberg and S. Skyum, A relationship between ET0L and EDT0L languages, Theor. Computer Science (to appear).
[7] R.M. Karp, Reducibility among combinatorial problems, in: R.E. Miller and J.W. Thatcher (ed.), Complexity of Computer Computations (Plenum Press, New York, 1973), 85–104.
[8] D.E. Knuth, A terminological proposal, SIGACT NEWS 6, January 1974, 12–18 (see also SIGACT NEWS 6, April 1974, 15–16).
[9] D.J. Rosenkrantz, Programmed grammars and classes of formal languages, JACM 16 (1969) 107–131.
[10] W.C. Rounds, Complexity of intermediate level languages, Conf. Record 14th Ann. Symp. on Switching and Automata Theory, Iowa (1973) 145–158.
[11] G. Rozenberg, T0L systems and languages, Inform. and Control 23 (1973) 357–381.
[12] G. Rozenberg, Personal communication (1974).
[13] A. Salomaa, Formal Languages, Acad. Press. New York (1973).

[14] E. Shamir and C. Beeri, Checking stacks and context-free programmed grammars accept P-complete languages, in: J. Loeckx (ed.), Proc. 2nd Colloquium on Automata, Languages, and Programming, Springer Lecture Notes in Computer Science 14 (1974) 27–33.

[15] J. van Leeuwen, Extremal properties of non-deterministic time-complexity classes, Proc. Intern. Computing Symposium 1975, Antibes-Juan les Pins (North-Holland Publ. Comp.) to appear.