On Exponential-Time Completeness of the Circularity Problem for Attribute Grammars

PEI-CHI WU
National Penghu Institute of Technology

Attribute grammars (AGs) are a formal technique for defining semantics of programming languages. Existing complexity proofs on the circularity problem of AGs are based on automata theory, such as writing pushdown acceptor and alternating Turing machines. They reduced the acceptance problems of above automata, which are exponential-time (EXPTIME) complete, to the AG circularity problem. These proofs thus show that the circularity problem is EXPTIME-*hard*, at least as hard as the most difficult problems in EXPTIME. However, none has shown that the problem is EXPTIME-complete. This paper presents an alternating Turing machine for the circularity problem. The alternating Turing machine requires polynomial space. Thus, the circularity problem is in EXPTIME and is then EXPTIME-complete.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.4 [**Programming Languages**]: Processors—*Compilers, translator writing systems and compiler generators*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Decision problems*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Attribute grammars, alternating Turing machines, circularity problem, EXPTIME-complete.

1. INTRODUCTION

Attribute grammars (AGs) [Knuth 1968] are a formal technique for defining semantics of programming languages. There are many AG systems (e.g., Eli [Gray et al. 1992] and Synthesizer Generator [Reps and Teitelbaum 1989]) developed to assist the development of "language processors," such as compilers, semantic checkers, language-based editors, etc. These AG systems process language specifications written in AGs and generate corresponding language processors.

An AG is *circular* if there is a cycle in (attribute) dependency graphs. A circular dependency is thought to be a specification error and should be detected by AG systems. Knuth [1968] presented an exponential-space algorithm for the

This research was partly supported by the National Science Council, Taiwan, R.O.C., under Contract No. NSC 89-2213-E-346-002.
Author's address: Department of Computer Science and Information Engineering, National Penghu Institute of Technology, Penghu, Taiwan, R.O.C.; email: pcwu@npit.edu.tw.

circularity problem. The algorithm collects a set of synthesized dependencies $sd(X)$ for each symbol X. Let $D(p)$ be the local dependency graph in production p: $X_0 \rightarrow X_1 \ldots X_k$. For any combination of $[d_1, \ldots, d_k]$, $d_i \in sd(X_i)$, $i > 0$, the algorithm then pastes together $[d_1, \ldots, d_k]$ with $D(p)$. The AG is circular if a cycle is found in the resulting dependency graph. The only exponential factor in the algorithm is the number of graphs in $sd(X)$.

The intrinsically exponential complexity of the circularity problem for AGs was first proved by Jazayeri et al. [1975], who reduced the acceptance problem of writing pushdown acceptors to the circularity problem. Jones [1980] proposed a simpler proof by reducing the acceptance problem of exponential-time-bounded deterministic Turing machines to the circularity problem of AGs. Jazayeri [1981] (and the correction by Dill [1989]) also tried to provide a simpler construction of AGs by reducing the acceptance problem of space-bounded alternating Turing machines [Chandra et al. 1981] to the circularity problem. The acceptance problem of (polynomial) space-bounded writing pushdown acceptors and alternating Turing machines are exponential-time- (EXPTIME-) complete. Both reductions show that the circularity problem for AGs is EXPTIME-*hard*, at least as hard as the most difficult problems in EXPTIME. However, none has shown that the problem is EXPTIME-complete.

To show the circularity problem for AGs to be EXPTIME-complete, we need an algorithm that requires exponential time but not actually exponential space. The existing circularity test algorithms [Deransart et al. 1984; Jazayeri et al. 1975; Knuth 1968; Räihä and Saarinen 1982] all require exponential space in the worst cases. Jazayeri et al. [1975, p. 704] also provided an upper bound on the circularity problem, which takes $O(2^{dn})$ time and space, where n is the grammar size. In fact, it is almost unlikely to find an algorithm for the circularity problem that takes only polynomial space. If there is one, the circularity problem is in polynomial space (PSPACE) and PSPACE $\supseteq$ EXPTIME. Since PSPACE $\subseteq$ EXPTIME, we would get PSPACE $=$ EXPTIME, which is still unknown in complexity theory (cf., e.g., Chandra et al. [1981]).

This paper presents an alternating Turing machine for the circularity problem. The alternating Turing machine requires polynomial space. Thus, the circularity problem is in EXPTIME and is then EXPTIME-complete.

2. THE ALTERNATING ALGORITHM

In an alternating Turing machine [Chandra et al. 1981; Jazayeri 1981], there are two main kinds of states: *universal* and *existential*. An existential state leads to an acceptance of the input, if *one* of the alternatives (possible next moves) leads to an accepting state. A universal state leads to an acceptance of the input, if *all* of the next moves lead to an accepting state.

Algorithm *CircularityTest* is an EXPTIME algorithm represented using an alternating Turing machine. Since complete development of the algorithm using "primitive" operations, for example, moving heads and writing tape symbols, in Turing machines is very complex, the algorithm is sketched using higher-level operations, borrowed from programming languages. Typical sequential code fragments are written directly in Pascal-style pseudocodes, which can easily be converted into alternating Turing machine operations. The "guess" actions in states q_1 and q_3 can be programmed using a series of existential states, each of which randomly chooses or does not choose an arc in a dependency subgraph. To explore the parallelism in alternating Turing machines, we use the ALL and SOME statements to create a number of parallel tasks (possible next moves). The ALL statement succeeds if all of the child tasks succeed; the SOME statement succeeds if some of the child tasks succeed. Each state is universal or existential depending on whether the ALL or SOME statement is used, respectively. Each task is given its next state and a number of input parameters, which are placed on the working tape. A task ends with either success or failure. Alternating Turing machines directly support task returns and do not need run-time stacks.

Algorithm *CircularityTest.*

Input. An AG *ag*. P denotes its production rules, N denotes its nonterminal symbols, $Inh(X)$ ($Syn(X)$) denotes the inherited (synthesized) attributes of symbol X, and $D(p)$ denotes the dependency graph of a production p.
Output. Whether *ag* is circular or not: *ag* is circular if *success*, noncircular if *failure*.
Method.

```
State $q_0$:
  SOME $q_1(p)$, $p \in P$;
State $q_1(p)$:
  IF $p : X_0 \to \varepsilon$
  THEN failure; {*There is no cycle in the dependency subgraph of $X_0$*}
  ELSE
    Let $p : X_0 \to X_1 \ldots X_k$;
    Guess an array of dependency subgraphs $[d_1, \ldots, d_k]$ for $X_i$,

                where $d_i \subseteq Inh(X_i) \times Syn(X_i)$, $i > 0$;

    Compose dependency subgraphs in $[d_1, \ldots, d_k]$ with $D(p)$;
    IF a (potential) cycle is found in the resulting dependency graph
    THEN ALL $q_2(X_i, d_i)$, $i > 0$; {*verify each dependency subgraph $d_i$ of $X_i$*}
    ELSE failure;
State $q_2(X; d)$:
  SOME $q_3(p, d)$, $\forall p \in P$, $p : X \to \ldots$; {* succeeds if production p has dependency
    graph d *}
State $q_3(p; d)$:
  IF $p : X_0 \to \varepsilon$ and $d = D(p)$
  THEN success;
  ELSE
    Let $p : X_0 \to X_1 \ldots X_k$;
    Guess an array of dependency subgraphs $[d_1, \ldots, d_k]$ for $X_i$,

                where $d_i \subseteq Inh(X_i) \times Syn(X_i)$, $i > 0$;

    Compose dependency subgraphs in $[d_1, \ldots, d_k]$ with $D(p)$;
    IF $d$ equals the resulting dependency subgraph induced at $X_0$
    THEN ALL $q_2(X_i, d_i)$, $i > 0$; {*verify each dependency subgraph $d_i$ of $X_i$*}
    ELSE failure;
```

End of Algorithm.

The initial state is q_0. The *success* statement enters an accepting state, and the *failure* statement enters a rejecting state. States q_1 and q_3 are universal.

They call state q_2 to verify that all dependency subgraphs d_i that make a cycle can be generated. The space needed in the algorithm is proportional to the size of the input n, that is, $O(n)$. The working tape contains the area for placing parameters, which space is limited by the size of largest parameters. Operations such as composing dependency graphs and detecting cycles can be programmed using $O(n)$ space.

The correctness of the algorithm is verified in brief as follows. The algorithm is a reverse version of the original circularity test algorithm [Knuth 1968]: first, guess an array of possible dependency subgraphs. Second, if there exists a cycle in the composition of these subgraphs, make sure that all the dependency subgraphs can be generated. The second step repeatedly calls itself (the calling sequence q_2-q_3-q_2-q_3-$\cdots$). This is similar to the step that collects dependency graphs repeatedly in the original algorithm. An infinite sequence of transitions may arise. Note that an algorithm can be in alternating PSPACE (APSPACE), even if it does not terminate. In alternating Turing machines, a task can return to its parent and has enough information to determine its result, although some of its child tasks have not returned [Chandra et al. 1981, p. 118].

The presented alternating Turing machine needs only polynomial space; however, it cannot be converted into a space-efficient algorithm in computers (or deterministic Turing machines), because alternating Turing machines are very powerful and have almost unlimited parallelism to create and execute new parallel tasks. A straightforward simulation of the presented alternating Turing machine needs exponential time and space, which is not better than the complexity of the existing circularity test algorithms. In addition, this algorithm may be even slower than the original algorithms, because this algorithm minimizes the space needed for each task but may perform redundant work by the numerous parallel tasks.

3. CONCLUSIONS

We have presented an alternating Turing machine for the circularity problem. The alternating Turing machine requires polynomial space. The circularity problem is in EXPTIME and is then EXPTIME-complete.

ACKNOWLEDGMENTS

The author would like to thank the referees, whose comments helped to improve the overall presentation.

REFERENCES

Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. 1981. Alternation. *J. ACM 28*, 1 (Jan.), 114–133.

Deransart, P., Jourdan, M., and Lorho, B. 1984. Speeding up circularity tests for attribute grammars. *Acta Informatica 21*, 375–391.

Dill, J. M. 1989. A counterexample for 'A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars.' *J. ACM 36*, 1 (Jan.), 92–96.

Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. W., and Waite, W. M. 1992. Eli: A complete, flexible compiler construction system. *Commun. ACM 35*, 2 (Feb.), 121–131.

Jazayeri, M., Ogden, W. F., and Rounds, W. C. 1975. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Commun. ACM 18*, 12 (Dec.), 697–706.

Jazayeri, M. 1981. A simpler construction for showing the intrinsically exponential complexity of the circularity problem for attribute grammars. *J. ACM 28*, 4 (Oct.), 715–720.

Jones, N. D. 1980. Circularity testing of attribute grammars requires exponential time: A simpler proof. Tech. rep. DAIMI PB-107. Computer Science Department, Aarhus University, Aarhus, Denmark.

Knuth, D. E. 1968. Semantics of context-free languages. *Math. Syst. Theory 2*, 2, 127–145. Correction: Knuth, D. E. 1971. *Math. Syst, Theory 5*, 1, 95–96.

Räihä, K.-J., and Saarinen, M. 1982. Testing attribute grammars for circularity. *Acta Informatica 17*, 185–192.

Reps, T., and Teitelbaum, T. 1989. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, NY.

Received December 2001; revised May 2002; accepted January 2003