Contents

Chapter 11

Decision problems for grammars

11.1 Computation histories of a Turing machine

11.1.1 Computation histories of Turing machines

There exists an important general method for proving undecidability results for various kinds of formal grammars. This method, discovered by Hartmanis [7], is based on representing *the language of computation histories of a Turing machine* by a grammar. Improved by Baker and Book [1] to use linear grammars.

Consider a Turing machine that accepts by halting, defined over some input alphabet Σ. At every moment of its computation, the machine's configuration is comprised of all symbols on the tape, the position of the head and the current internal state. These data can be written as a string over a suitable alphabet. The *computation history* of the machine on a given input is a string obtained roughly by concatenating string representations of the machine's configurations at each step of its computation on this input. If the machine eventually halts, this is a finite string, and the set of all such strings representing halting computations is *the language of valid accepting computations*, commonly denoted by VALC(T), where T is a Turing machine.

Let $T = (\Sigma, \Gamma, Q, q_0, \delta, q_{acc})$ be a deterministic Turing machine, where $\Gamma \supset \Sigma$ be the tape alphabet of T, which contains a blank symbol $\text{␣} \in \Gamma \setminus \Sigma$, and Q be the set of states, disjoint with Γ. Assume that T operates on a one-sided infinite tape and never attempts to move beyond its leftmost symbol. Initially, the tape contains the input string followed by infinitely many squares filled with blank symbols, the machine scans its leftmost symbol and is in the initial state $q_0 \in Q$. The machine accepts by entering the state q_{acc}.

Let $\Omega = \Gamma \cup Q \cup \{\#, \$\}$ be the alphabet used for representing computation histories. Whenever T is in a state $q \in Q$ scanning a symbol $a \in \Gamma$, and its tape contains a string $u \in \Gamma^*$ to the left of the head and a string $v \in \Gamma^*$ to the right (excluding the blank symbols that have not yet been visited), this configuration can be encoded by the string $uqav \in \Gamma^* Q \Gamma^+$. For every input string $w \in \Sigma^*$, denote the machine's configuration after i steps of computation by

$$C_i = C_i(T, w) = uqav.$$

If T halts on w after n steps, then its computation history is

$$C_T(w) = w\#C_0\#C_1\#C_2\#\ldots\#C_{n-1}\$C_n\#C_n^R\#C_{n-1}^R\#\ldots\#C_2^R\#C_1^R\#C_0^R.$$

The language of computation histories of T is

$$\text{VALC}(T) = \{\, C_T(w) \mid w \in L(T)\}$$

Lemma 11.1. *For every Turing machine T there exists and can be effectively constructed such LL(1) linear grammars G_1 and G_2, that $L(G_1) \cap L(G_2) = \text{VALC}(T)$.*

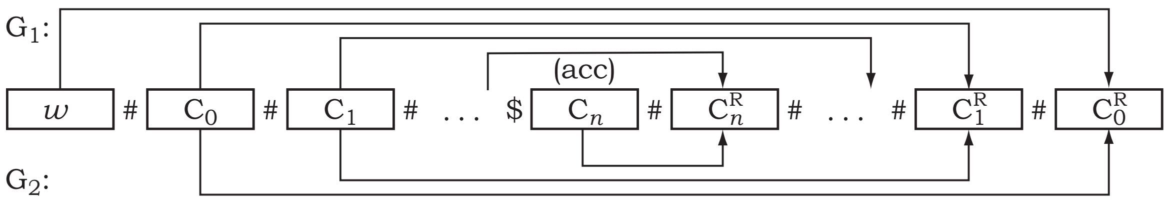

Figure 11.1: Representing VALC(T) $= \{\, C_T(w) \mid w \in L(T)\}$ as $L(G_1) \cap L(G_2)$, for LL(1) linear grammars G_1 and G_2.

Proof. The form of computation histories and their representation as an intersection of two languages is illustrated in Figure 11.1. Each configuration is listed in the computation history twice, and the first grammar compares the first reversed copy of each i-th configuration C_i^R to the second copy of the next configuration C_{i+1}. The last configuration C_n^R is preceded by a special symbol \$, which instructs the grammar that instead of comparing the next configuration to another, it should ensure that it is accepting. The second grammar only verifies that both copies of each configuration are identical.

The first grammar begins by comparing the input string w written in the beginning of the computation history to the reversed first configuration $C_0^R = wq_0$ in the very end.

$$\begin{aligned} S_1 &\to Bq_0 \\ B &\to aBa && (a \in \Sigma) \\ B &\to \#A\# \end{aligned}$$

Next, the nonterminal A is used to compare each configuration C_i on the left to the reversed next configuration C_{i+1}^R on the right, in order to ensure that these are indeed consecutive configurations of T. The grammar first matches the symbols in the unchanged portion of the tape.

$$A \to cAc \qquad (c \in \Gamma)$$

When the head and the state of the Turing machine are encountered in the configuration C_i on the left, as a substring of the form bqa, with $b, a \in \Gamma$ and $q \in Q$, the grammar matches them to the corresponding symbols in the next configurations C_{i+1}, which are determined by the machine's transition function. Let $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ be the transition function of T, where $\delta(q, a)$ determines the behaviour of T in a state $q \in Q$ when observing a symbol $a \in \Sigma$: the machine enters a given new state, overwrites a with a given new symbol and moves the head in a given direction.

Consider first the case of transition to the left, and let $\delta(q, a) = (q', a', -1)$. Then, a configuration $C_i = ubqav$ is followed by $C_{i+1} = uq'ba'v$, which is written backwards as $C_{i+1}^R = v^R a' b q' u^R$. The corresponding rule of the grammar puts the substrings bqa and $a'bq'$ at the two sides of the string.

$$A \to bqaAa'bq' \qquad (\delta(q, a) = (q', a', -1),\ b \in \Gamma)$$

If the machine uses a transition $\delta(q, a) = (q', a', +1)$ to change from a configuration $C_i = uqacv$ to $C_{i+1} = ua'q'cv$ (written as $C_{i+1}^R = v^R c q' a' u^R$), the grammar puts aq and $a'q'$ around the string.

$$A \to qaAq'a' \qquad (\delta(q, a) = (q', a', +1)).$$

The rest of the symbols in C_i are compared to their counterparts in C_{i+1}^R by the rule $A \to cAc$. Once all symbols in these configurations are generated, the grammar proceeds either to the next pair of configurations or to an accepting configuration.

$$\begin{aligned} A &\to \#A\# \\ A &\to \$C\# \end{aligned}$$

A special case of a transition between two configurations is when the Turing machine moves beyond the rightmost symbol of the tape. In this case, $C_i = uq$, and a missing symbol under q is treated as a blank symbol.

$$\begin{aligned} A &\to bq\#A\#a'bq' && (\delta(q, \text{␣}) = (q', a', -1),\ b \in \Gamma) \\ A &\to bq\$C\#a'bq' && (\delta(q, \text{␣}) = (q', a', -1),\ b \in \Gamma) \\ A &\to q\#A\#q'a' && (\delta(q, \text{␣}) = (q', a', +1)) \\ A &\to q\$C\#q'a' && (\delta(q, \text{␣}) = (q', a', +1)) \end{aligned}$$

Finally, the grammar has to check that the last configuration C_n^R is accepting. This is done in the symbol C.

$$\begin{aligned} C &\to cC && (c \in \Gamma) \\ C &\to q_{\text{Acc}}C \\ C &\to \varepsilon. \end{aligned}$$

The grammar, as defined above, is LL(3). Indeed, each rule of the form $B \to bqaBa'bq'$ is completely determined by the transition of the Turing machine in the state q by the symbol a, and in order to distinguish between several such rules, a parser needs to see both q and a. This requires a three-symbol look-ahead.

An LL(1) grammar is obtained by reconstructing the rules for A, so that the required three symbols of look-ahead are accummulated in A's subscripts. Consider new nonterminal symbols A_b, with $b \in \Gamma$, $A_{b,q}$, with $b \in \Gamma \cup \{\varepsilon\}$ and $q \in Q$, and $A_{q,a}$, with $q \in Q$ and $a \in \Sigma$. The intention is to have $L(A_b) = \{\, u \mid bu \in L(A)\}$, $L(A_{b,q}) = \{\, u \mid bqu \in L(A)\}$ and $L(A_{q,a}) = \{\, u \mid qau \in L(A)\}$. This is achieved by the rules below that replace all rules for A.

$$\begin{aligned} A &\to qA_{\varepsilon,q} && (q \in Q) \\ A &\to bA_b && (b \in \Gamma) \\ A_b &\to qA_{b,q} && (b \in Q,\ q \in Q) \\ A_{b'} &\to b'A_bb && (b, b' \in \Gamma) \\ A_{b,q} &\to aAa'bq' && (\delta(q, a) = (q', a', -1),\ b \in \Gamma) \\ A_{b,q} &\to \#A\#a'bq' && (\delta(q, \text{␣}) = (q', a', -1)) \\ A_{b,q} &\to \$C\#a'bq' && (\delta(q, \text{␣}) = (q', a', -1)) \\ A_{b,q} &\to aAq'a'b && (\delta(q, a) = (q', a', +1)) \\ A_{b,q} &\to \#A\#q'a'b && (\delta(q, \text{␣}) = (q', a', +1)) \\ A_{b,q} &\to \$C\#q'a'b && (\delta(q, \text{␣}) = (q', a', +1)) \\ A_b &\to \#A\#b && (b \in \Gamma) \\ A &\to \#A\# \end{aligned}$$

The second grammar simply verifies that for every configuration C_i on the left, the corresponding string on the left is indeed the reversal C_i^R of this configuration. This is done using the following rules.

$$\begin{aligned}
S_2 &\to aS_2 && (a \in \Sigma)\\
S_2 &\to D\\
D &\to sEs && (s \in \Gamma \cup Q)\\
D &\to \#D\# \mid \$E\#\\
E &\to sEs && (s \in \Gamma \cup Q)\\
E &\to \#
\end{aligned}$$

□

Undecidable to test whether VALC(T) is empty.

Theorem 11.1. *Given two LL(1) linear grammars, G_1 and G_2, it is undecidable whether the intersection $L(G_1) \cap L(G_2)$ is empty.*

Theorem 11.2. *It is undecidable whether a given linear grammar is unambiguous.*

Proof. Given a Turing machine, construct the grammars $G_1 = (\Sigma, N_1, R_1, S_1)$ and $G_2 = (\Sigma, N_2, R_2, S_2)$, as in Lemma 11.1. Let $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, R_1 \cup R_2 \cup R, S)$ be a new grammar, which combines G_1 and G_2 by the following new rules.

$$S \to S_1 \mid S_2$$

As the grammars G_1 and G_2 are unambiguous, the only possible source of ambiguity in G is the choice between the two new rules. If VALC(T) $= \varnothing$, then $L(G_1) \cap L(G_2) = \varnothing$, and the choice is unambiguous. And if there is a string $w \in$ VALC(T), then the choice is ambiguous on this string w. □

11.1.2 The complement of the language of computation histories

Language of computation histories VALC(T) $\subseteq \Omega^*$. Representable as VALC(T) $= L_1 \cap L_2$.
Its complement: $\overline{\text{VALC}(T)} = \overline{L_1} \cup \overline{L_2}$.

Lemma 11.2. *For every Turing machine T, the complements of each of two languages defined in Lemma 11.1 are described by LR(1) linear grammars G_1' and G_2'.*

Proof. The grammars G_1 and G_2 are LL(1), and therefore LR(1). Then, LR(1) languages are closed under complementation. □

Undecidable to test whether $\overline{\text{VALC}(T)}$ is Σ^*.
Therefore, equivalence undecidable already for linear grammars.

Theorem 11.3. *Testing whether a given linear grammar generates the set of all strings over its alphabet is undecidable.*

Proof. Given a Turing machine, let $G_1' = (\Sigma, N_1, R_1, S_1)$ and $G_2' = (\Sigma, N_2, R_2, S_2)$, be the grammars defined in Lemma 11.2. Construct a new grammar $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, R_1 \cup R_2 \cup R, S)$, which combines G_1' and G_2' by the following new rules.

$$S \to S_1 \mid S_2$$

The new grammar is still linear, and it describes the language Σ^* if and only if VALC(T) $= \varnothing$. □

Corollary 11.1. *Given two linear grammars, it is undecidable whether they describe the same language*

Proof. Let the second grammar generate Σ^*. □

11.2 Equivalence problem for deterministic grammars

11.2.1 Equivalence problem for LL(k) grammars

Theorem 11.4 (Rosenkrantz and Stearns [10]). *Given two LL(k) ordinary grammars, it is decidable whether these grammars describe the same language.*

Sketch of a proof. Two grammars, $G_1 = (\Sigma, N_1, R_1, S_1)$ and $G_2 = (\Sigma, N_2, R_2, S_2)$. Assume that both grammars are in the Greibach normal form, that is, with all rules of the form $A \to a\alpha$, with $a \in \Sigma$ and $\alpha \in (\Sigma \cup N_i)^*$. For the sake of the uniformity of notation, also assume that $N_1 \cap N_2 = \varnothing$ and denote $N = N_1 \cup N_2$ and $R = R_1 \cup R_2$.

General plan: construct a DPDA that recognizes the symmetric difference of $L(G_1)$ and $L(G_2)$, that is, accepts those strings that are generated only by one of the two grammars. Then test this DPDA for emptiness.

This requires simulating both LL parsers together. If their stacks contained the same number of symbols at every step, this would be easy. However, their stacks may contain a different number of symbols even if the grammars describe the same language.

Example 11.1. *Consider the following two different LL(k) grammars, both describing the same language* $\{\, a^n b^{6n} \mid n \geqslant 0\}$

$$\begin{aligned} S_1 &\to aS_1BB \mid \varepsilon \qquad & S_2 &\to aS_2CCC \mid \varepsilon \\ B &\to bbb & C &\to bb \end{aligned}$$

On an input string $a^n b^{6n}$, after reading a^n, the first parser has $2n$ symbols in its stack (B^{2n}), whereas the second parser has $3n$ symbols (C^{3n}).

Define another notion of similarity of stack contents, so that, as long as two grammars describe the same language, their LL parsers' stack contents will be similar to each other at every step of their computations.

Thickness of a stack symbol $X \in \Sigma \cup N$, denoted by $\tau(X)$, is defined as the length of the shortest string representable as X. Then, $\tau(a) = 1$ for $a \in \Sigma$ and $\tau(A) = \min_{w \in L(A)} |w|$, This notion is extended to the whole stack contents as $\tau(X_1 \ldots X_\ell) = \tau(X_1) + \ldots + \tau(X_\ell)$.

Assume that the two grammars describe the same language, and that their parsers read the same prefix u of the same string uv, reaching configurations (α_1, v) and (α_2, v), respectively. Then, the sets of strings v accepted from these configurations must be the same, and, in particular, the shortest accepted strings have the same length.

Thickness of a symbol wrt look-ahead $x \in \Sigma^{\leqslant k}$. $\tau_x(A) = \min_{w \in L(A), \mathrm{First}_k(w) = x} |w|$. Can be larger than $\tau(A)$.

Example 11.2. *Consider the grammar $S \to aS \mid bbS \mid \varepsilon$ and its LL(1) parser. Then, $\tau(S) = 0$, because $\varepsilon \in L(S)$, and $\tau_\varepsilon(S) = 0$, $\tau_a(S) = 1$ and $\tau_b(S) = 2$.*

Let $m = \max_{A \to \alpha \in R} \tau(\alpha)$ be the largest thickness of a right-hand side of any rule.

Claim: $\tau_x(\alpha) - \tau(\alpha) \leqslant k(m-1)$, where $t = \max_{A \to \alpha \in R} \tau(\alpha)$.

Proof of the claim: Let $\alpha = X_1 \ldots X_\ell$. If $w \in L(\alpha)$ and $\mathrm{First}_k(w) = x$, then applying at most k rules to the first symbols of α leads to $w \in L(x\beta X_{k+1} \ldots X_\ell)$, for some $\beta \in \Sigma \cup N$. Each of the rules applied has thickness at most m, and it replaces a nonterminal symbol of thickness at least 1. Therefore, the thickness of $x\beta$ is at most $k(m-1)$, leading to the desired upper bound.

$$\tau_x(X_1 \ldots X_\ell) \leqslant \tau(x\beta X_{k+1} \ldots X_\ell) \leqslant k(m-1) + \tau(X_{k+1} \ldots X_\ell) \leqslant k(m-1) + \tau(X_1 \ldots X_\ell)$$

If on uv, the first parser reads u and enters configuration (α_1, v), whereas the second parser enters the configuration (α_2, v). Then, $\tau_{\mathrm{First}_k(v)}(\alpha_1) = \tau_{\mathrm{First}_k(v)}(\alpha_2)$. This does not guarantee that $\tau(\alpha_1) = \tau(\alpha_2)$.

Claim: $|\tau(\alpha_1) - \tau(\alpha_2)| \leqslant k(m-1)$.

(follows from the previous claim)

(2) Construct a DPDA that, given an input string $w \in \Sigma^*$, simulates both parsers on w at once, assuming the similarity of their stack contents, and if this similarity is ever violated, the DPDA skips the rest of w and accepts; otherwise, the simulation continues, and if one of the simulated parsers accepts and the other rejects, the DPDA accepts, and rejects otherwise. Two-track simulation of both parsers, each symbol used by an LL parsr occupies a number of stack symbols corresponding to its thickness. The $k(m-1)$ top symbols are kept in the internal state. □

Korenjak and Hopcroft [8] gave a different algorithm for testing equivalence of LL(1) grammars in Greibach normal form, which does not use any intermediate problems. Olshansky and Pnueli [9] extended the method of Korenjak and Hopcroft to LL(k) grammars.

The inclusion problem is undecidable already for linear LL(1) grammars.

Theorem 11.5 (Friedman [5])**.** *Given two linear LL(1) grammars G_1 and G_2 in Greibach normal form, it is undecidable to determine whether $L(G_1)$ is a subset of $L(G_2)$.*

11.2.2 LR grammars

Equivalence for LR grammars was proved decidable by Sénizergues [11].

Later, alternative solutions were presented by Stirling and by Jančar.

Bibliography

[1] B. S. Baker, R. V. Book, “Reversal-bounded multipushdown machines”, *Journal of Computer and System Sciences*, 8 (1974), 315–332.

[2] Y. Bar-Hillel, M. Perles, E. Shamir, “On formal properties of simple phrase-structure grammars”, *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14 (1961), 143–177.

[3] D. J. Cantor, “On the ambiguity problem of Backus systems”, *Journal of the ACM*, 9:4 (1962), 477–479.

[4] N. Chomsky, M. P. Schützenberger, “The algebraic theory of context-free languages”, in: Braffort, Hirschberg (Eds.), *Computer Programming and Formal Systems*, North-Holland, 1963, 118–161.

[5] E. P. Friedman, “The inclusion problem for simple languages”, *Theoretical Computer Science*, 1:4 (1976), 297–316.

[6] S. A. Greibach, “The undecidability of the ambiguity problem for minimal linear grammars”, *Information and Control*, 6:2 (1963), 119–125.

[7] J. Hartmanis, “Context-free languages and Turing machine computations”, *Proceedings of Symposia in Applied Mathematics*, Vol. 19, AMS, 1967, 42–51.

[8] A. J. Korenjak, J. E. Hopcroft, “Simple deterministic languages”, *7th Annual Symposium on Switching and Automata Theory* (SWAT 1966, Berkeley, California, USA, 23–25 October 1966), IEEE Computer Society, 36–46.

[9] T. Olshansky, A. Pnueli, “A direct algorithm for checking equivalence of LL(k) grammars”, *Theoretical Computer Science*, 4:3 (1977), 321–349.

[10] D. J. Rosenkrantz, R. E. Stearns, “Properties of deterministic top-down grammars”, *Information and Control*, 17 (1970), 226–256.

[11] G. Sénizergues, $L(A) = L(B)$? decidability results from complete formal systems”, *Theoretical Computer Science*, 251:1–2 (2001), 1–166.

[12] R. E. Stearns, H. B. Hunt III, “On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata”, *SIAM Journal on Computing*, 14 (1985), 598–611.

Index