Equivalence of simple functions

Cédric Bastien\textsuperscript{a}, Jurek Czyzowicz\textsuperscript{a}, Wojciech Fraczak\textsuperscript{b,a,*}, Wojciech Rytter\textsuperscript{c}

\textsuperscript{a}Dépt d’informatique, Université du Québec en Outaouais, Gatineau PQ, Canada
\textsuperscript{b}IDT Canada Inc., Ottawa ON, Canada
\textsuperscript{c}Inst. of Informatics, Warsaw University, Warsaw, Poland

Abstract

A partial function $F : \Sigma^* \rightarrow \Omega^*$ is called a \textit{simple} function if $F(w) \in \Omega^*$ is the output produced in the leftmost derivation of a word $w \in \Sigma^*$ from a nonterminal of a simple context free grammar $G$ with output alphabet $\Omega$. In this paper we present an efficient algorithm for testing the equivalence of simple functions. Such functions correspond also to one-state deterministic pushdown transducers. Our algorithm works in time polynomial with respect to $|G| + v(G)$, where $|G|$ is the size of the textual description of $G$, and $v(G)$ is the maximum of the shortest lengths of words generated by nonterminals of $G$.

Keywords: Formal language; Context-free grammar; Push-down transducer; Equivalence problem; Simple grammar; Simple function

1. Introduction

The decidability problem of the equivalence for functions defined by different classes of deterministic push-down automata and pushdown transducers (dpdt) was studied extensively, see for example [9,12], leading eventually to a proof of the decidability of the equivalence problem for deterministic pushdown transducers. The main issue was decidability, and little was said about the effective algorithms for the equivalence of pushdown transducers.

In this paper we present an efficient and easy to implement algorithm for deciding the equivalence of simple functions, i.e., functions defined by one-state dpdts. The algorithm we propose in this paper is a nontrivial extension of the simple languages equivalence algorithms from [3,8,2] to the case of simple functions. Simple functions can be seen as a proper extension of sequential functions (functions realized by deterministic finite transducers) and simple languages. Simple languages were introduced in [10] as languages recognized by a dpda with a single state, also called \textit{simple pda}, or, equivalently, as languages generated by \textit{simple grammars}. We extend the definition of simple grammars to functions defined by grammars with output.

Simple functions were initially introduced in [6]. The interest in simple functions was originally triggered by an application in the domain of packet classification at IDT Canada (Integrated Device Technology Inc.), which specializes in producing integrated circuits responsible for performing network packet classification at wire speed. The

---

\textsuperscript{*} The research of the first three authors was supported by NSERC and the research of the fourth author was supported by the grant KBN 4T11C04425.

\textsuperscript{*} Corresponding author at: IDT Canada Inc., Ottawa, ON, Canada. Tel.: +1 613 287 5125; fax: +1 613 287 5190.

E-mail address: fraczak@gmail.com (W. Fraczak).

0304-3975/ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2007.01.011
The process of classification is accomplished with the aid of a specialized push-down transducer called the *Concatenation State Machine*, which reads the input packet and produces as its output the result of the classification. Concatenation State Machines may be considered as hardware implementations of single-state dpdts. The classification policies, usually defined by a human, are described with the aid of a subclass of context free grammars, representing simple functions. Any such user-defined grammar description is then automatically converted into a corresponding Concatenation State Machine.

In some cases, e.g., when the classification purpose is routing, the classification policies may evolve continuously, usually in an incremental way, keeping large portions of the policies intact. Indeed, packet routing may be altered dynamically following network adjustments (i.e., node or link failures, traffic congestion, etc.). On the other hand, many user defined classification policies may share large fragments of the policy description. It then often happens that parts of such policy descriptions are semantically equivalent. In order to manage large sets of classification policies in memory it is useful to represent them in a structured way so that the same descriptions are not kept uselessly in many equivalent versions. Consequently, it is necessary to recognize whether two simple functions are equivalent. The simple functions equivalence algorithm must be reasonably efficient, at least in the average case. The algorithm presented in this paper is the solution to the above problem. It should be noted as well that the presented algorithm turns out to be relatively easy to implement.

A *simple function grammar* is formally described by a 4-tuple:

\[ G = (\Sigma, \Omega, N, P), \]

where \( \Sigma, \Omega, N \) are finite and mutually disjoint sets of *input symbols*, *output symbols*, and *nonterminals*, and \( P \subset N \times \Sigma \times (N \cup \Omega)^* \) is a finite set of *production rules* with output. Moreover, we require that for given \( A \in N \) and \( a \in \Sigma \) there is at most one \( \alpha \in (N \cup \Omega)^* \), such that \((A, a, \alpha) \in P \).

Each production can be written as \( A \rightarrow aa \), where \( a \in \Sigma \) and \( \alpha \in (N \cup \Omega)^* \). We also write \( A \overset{a}{\rightarrow} \alpha \). The relation \( \overset{a}{\rightarrow} \) is extended in the following way.

We write \( \alpha_1 \overset{a}{\rightarrow} \alpha_2 \), iff \( \alpha_1 = \beta_1 \alpha_2 \beta_1, \beta_1 \in \Omega^*, \beta_2 \in (N \cup \Omega)^* \), \( A \in N, \alpha_2 = \beta_1 \gamma \beta_2, \) and \( A \rightarrow \alpha \gamma \) is a production.

Intuitively, relation \( \alpha_1 \overset{a}{\rightarrow} \alpha_2 \) corresponds to a single-step leftmost derivation.

For \( w = a_1 a_2 \ldots a_n \) and \( \alpha_i \in (N \cup \Omega)^* \) we write \( \alpha_0 \overset{w}{\rightarrow} \alpha_n \) iff

\[ \alpha_0 \overset{a_1}{\rightarrow} \alpha_1, \alpha_1 \overset{a_2}{\rightarrow} \alpha_2, \alpha_2 \overset{a_3}{\rightarrow} \alpha_3, \ldots \alpha_{n-1} \overset{a_n}{\rightarrow} \alpha_n. \]

For \( w \in \Sigma^* \) we write:

\[ \beta = \text{Derived}(\alpha, w) \Leftrightarrow \alpha \overset{w}{\rightarrow} \beta, \text{where } \beta \in (N \cup \Omega)^*. \]

If there is no derivation \( \alpha \overset{w}{\rightarrow} u \) for any \( u \) then we write \( \text{Derived}(\alpha, w) = \perp \). The input-output relation corresponding to a sequence \( \alpha \in (N \cup \Omega)^* \) is defined in the following way:

\[ F_G(\alpha) \overset{\text{def}}{=} \{(w, u) \in \Sigma^* \times \Omega^* \mid u = \text{Derived}(\alpha, w), u \neq \perp \}. \]

A relation \( F_G(\alpha) \), for any given \( \alpha \in (\Omega \cup N)^* \), over input and output strings, which can be defined by a simple function grammar, is called a *simple function*. We use also function terminology, i.e., \( F_G(\alpha)(w) = u \) iff \((w, u) \in F_G(\alpha)\).

The domain of a simple function is a simple language, i.e., if \( \Omega \) is empty then \( G \) is just a simple grammar.

We define the *simple function equivalence problem* as follows.

**Input:** a simple function grammar \( G \) and two nonterminals \( A, B \in N \);

**Output:** SUCCESS if \( F_G(A) = F_G(B) \), and FAILURE otherwise.

**Example 1.** Let us consider the simple function grammar

\[ G = ((0, 1), \{a, b\}, \{S_1, S_2, A_1, A_2\}, P), \]

where \( P \) is given by rules:

\[ S_1 \rightarrow 0aS_1A_1b, \quad S_1 \rightarrow 1, \quad A_1 \rightarrow 1, \quad S_2 \rightarrow 0aS_2A_2, \quad S_2 \rightarrow 1, \quad A_2 \rightarrow 1b \]

and consider the equivalence problem \( F_G(S_1) = F_G(S_2) ? \).
We have SUCCESS since $F_G(S_1) = F_G(S_2)$.
For $w$ which are not of the form $0^n1^{n+1}$, $F_G(S_1)(w)$ and $F_G(S_2)(w)$ are both undefined. Otherwise, we have:

$$F_G(S_1)(0^n1^{n+1}) = F_G(S_2)(0^n1^{n+1}) = a^n b^n.$$

Let $\alpha \in (N \cup \Omega)^*$. By $||\alpha||$ we denote the shortest-word complexity of $\alpha$ defined as the length of a shortest $w \in \Sigma^*$ such that $F_G(\alpha)(w)$ is defined. The shortest-word complexity of grammar $G$ is defined as $\nu(G) \defeq \max(||A|| \mid A \in N). |G|$ denotes the size of the textual description of $G$.

Our main result is the constructive proof of the following theorem.

**Theorem 2.** Assume $A$, $B$ are two nonterminals of a simple grammar $G$ with output. Then we can test if $F_G(A) = F_G(B)$ in time polynomial with respect to $|G| + \nu(G)$.

2. Free group over $\Omega$ and properties of simple functions

In the course of the algorithm we consider, as intermediate data, output sequences which are to be compensated for later. For example we could know that the output for $A$ is the same as for $B$, except for a prefix $u$ that must be cut off from every output for $B$. Then, we formally write $uA = B$, or equivalently $A = u^{-1}B$.

This motivates the introduction of the free group $\Omega^\circ$ over the output alphabet $\Omega$. The concept of this group and the operation Derived of taking a syntactic remainder are among our basic tools. First we introduce some basic properties and definitions related to the free group over $\Omega$.

By $\varepsilon$ we denote an empty sequence. Simple functions together with concatenation defined by $fg \defeq \{(x_1 y_2, y_1 y_2) \mid (x_1, y_1) \in f, (x_2, y_2) \in g\}$, constitute a monoid with $\{(\varepsilon, \varepsilon)\}$ acting as unit and with $\{(), ()\}$ acting as zero. More details about simple functions seen as a monoid can be found in [6]. We will write $u$ and $v$ instead of $\{(w, v)\}$ and $\{(v, u)\}$ respectively. In particular, the unit function $\{(\varepsilon, \varepsilon)\}$ will be denoted by $\varepsilon$.

Let $f, g$ be simple functions. By $g^{-1} f$ we will denote the unique, if it exists, simple function $h$ such that $f = gh$.

As mentioned above, for technical reasons we extend the image of simple functions to the free group generated by $\Omega$, denoted by $\Omega^\circ$, so $w^{-1}f$ would be such that $w(w^{-1}f) = f$, for all $w \in \Omega^\circ$. More precisely, $\Omega^\circ \defeq (\Omega \cup \Omega^\circ)/(a\Omega = \Omega a = \varepsilon\mid a \in \Omega)$, where $\Omega$ is a copy of $\Omega$ with bijection $\varepsilon : \Omega \mapsto \Omega$ playing the role of the inverse. Therefore, apart from monoid properties, i.e., $x(yz) = (xy)z$, $x\varepsilon = \varepsilon x = x$, we have $\overline{a\overline{a}} = \overline{\varepsilon a} = \varepsilon$, for every $a \in \Omega$. For example, $(\overline{ab\overline{c}})^{-1} = \overline{cb\overline{a}}$, or $bc(\overline{abc})^{-1} = \overline{a}$.

Given two strings $u, v$ over $\Omega \cup \Omega^\circ$, we write $u = v$ to say that they are equivalent in $\Omega^\circ$. When we want to underline that $u$ and $v$ are identical as strings, we write $u \equiv v$. A usual way of representing an element of $\Omega^\circ$, i.e., an equivalence class over $(\Omega \cup \Omega^\circ)^*$, is to choose the shortest string from the class (such a word does not contain subwords $\overline{a}\overline{a}$ or $\overline{a}a$, for any $a \in \Omega$). Given a string $u \in (\Omega \cup \Omega^\circ)^*$, by reduce$(u)$ we denote the shortest string over $\Omega \cup \Omega^\circ$ such that reduce$(u) = u$.

If reduce$(u) \equiv u$ then $u$ is called reduced. The reduced form can be easily computed in linear time with respect to $|u|$. We say that $u \in \Omega^\circ$ is not primitive if $u \neq \varepsilon$ and there is an $x \in \Omega^\circ$ and $k > 1$ such that $u = x^k$; otherwise $u$ is primitive. For every $u \neq \varepsilon$ there exists a unique primitive $x \in \Omega^\circ$, denoted root$(u)$, and a $k > 0$, denoted power$(u)$, such that $u = x^k$.

**Lemma 3.** Let $u \in (\Omega \cup \Omega^\circ)^*$. There is an algorithm for calculating power$(u)$ and root$(u)$ running in $O(n)$, where $n = |u|$.

**Proof.** We assume that $u$ is given in the reduced form. Let $u$ be written as $u_1 u_2 u_1^{-1}$, where $u_1$ is the maximal length prefix of $u$ such that its inverse is a suffix of $u$. The value of $u_1$ can be easily computed in linear time. Note that $u$ is a power of a primitive word $x$ iff $u_2$ is a power of a primitive word $y$ such that $x = u_1 y u_1^{-1}$. By the choice of $u_2$, the first and the last symbols of $y$ are not inverse of each other, therefore reduce$(y^k) \equiv y^k$ for all $k \geq 1$ and $u_2$ can be treated as a word in a free monoid generated by the alphabet $(\Omega \cup \Omega^\circ)$. In this context, computing power$(u_2)$ and root$(u_2)$ can be done by finding the occurrences of $u_2$ in the word $u_2 u_2$ using any linear time pattern matching algorithm (see [5] for details), from which we can deduce the values of power$(u)$ and root$(u)$. □
Lemma 4. Let $X \subseteq \Omega^\circ$ such that $|X| \geq 2$, $r_1, r_2 \in \Omega^\circ$ be both primitive, and $r_1 w = w r_2$ for all $w \in X$. For every $u, v \in \Omega^\circ$ with $u, v \neq \varepsilon$, $uw = vw$ for all $w \in X$ if $\text{power}(u) = \text{power}(v)$ and $(\text{root}(u), \text{root}(v)) \in \{(r_1, r_2), (r_1^{-1}, r_2^{-1})\}$.

Proof. Firstly, we observe that in $\Omega^\circ$, $uw = vw$ iff $u = st, v = ts$, and $w = (st)^k s$, for some $s, t \in \Omega^\circ$ and $k \in \mathbb{Z}$ (Observation 1a). If in addition $uw' = w'u$ then $w' = (st)^k s$, for the same $s, t$ and some $k' \in \mathbb{Z}$ (Observation 1b). Moreover, if $uw = vw$ then $u = wk_1, v = wk_2$, for some $w \in \Omega^\circ$ and $k_1, k_2 \in \mathbb{Z}$ (Observation 2).

It is straightforward to prove if the part of the lemma, from Observation 1a.

To prove the only if part, we use Observations 1a and 1b to find that for any two words $w_1, w_2 \in X$ there exist $s, t \in \Omega^\circ$ and $k_1, k_2 \in \mathbb{Z}$ such that $r_1 = st, r_2 = ts$, $w_1 = (st)^{k_1} s$, and $w_2 = (st)^{k_2} s$. Therefore, $u(st)^{k_1} s = (st)^{k_1} s u$ and $u(st)^{k_2} s = (st)^{k_2} s u$. From these two equations, by eliminating $u$ we obtain $(ts)^{k_1 - k_2} v = v(ts)^{k_1 - k_2}$. Using Observation 2 and since $ts$ is primitive and $k_1 - k_2 \neq 0$, there exists $k_3 \in \mathbb{Z}$ such that $v = (ts)^{k_3}$ and $u = (st)^{k_3}$. This implies $\text{power}(u) = \text{power}(v)$ and $(\text{root}(u), \text{root}(v)) \in \{(r_1, r_2), (r_1^{-1}, r_2^{-1})\}$. □

Example 5. Note that, if $|X| = 1$ then the lemma does not apply. E.g., for $X = \{\varepsilon\}$, $u \varepsilon = \varepsilon u$ for all $u \in \Omega^\circ$.

From this point on, we extend the definition of the output alphabet to $\Omega \cup \overline{\Omega}$ and we will assume $G = (\Sigma, \Omega \cup \overline{\Omega}, N, \text{P})$ is a simple grammar with output.

We distinguish two types of sequences $\alpha$ over $N \cup \Omega \cup \overline{\Omega}$: $\alpha$ is of output type if $\alpha \in (\Omega \cup \overline{\Omega})^*$; and $\alpha$ is of general type, when $\alpha$ is of form $u A \alpha'$, for some $u \in (\Omega \cup \overline{\Omega})^*$, $A \in N$, and $\alpha' \in (N \cup \Omega \cup \overline{\Omega})^*$. In the case of general type of $\alpha$, we refer to $u, A, \text{and } \alpha'$ by OutPref($\alpha$), First($\alpha$), and Tail($\alpha$), respectively.

Example 6. Let $\Omega = \{a, b\}$, $N = \{A, B, C\}$, and $\alpha = baabaAbBaCaAb$. First($\alpha$) = $A$, OutPref($\alpha$) = $baaba$, and Tail($\alpha$) = $bBaCaAb$.

For every simple function $F_G(A)$, denoted by $\min F_G(A)$ the unique element $(w, u) \in F_G(A)$ such that $w \in \Sigma^*$ is the shortest and lexicographically smallest input word generated by $G$ from $A$. For every nonterminal $A \in N$ we can precompute $(w_A, u_A) \doteq \min F_G(A)$ in time polynomial with respect to $|A|$.

Moreover, we compute the set $\text{SingleOut}(G) \subseteq N$ of all non-terminals, each of them producing only one output string, i.e.,

$$\text{SingleOut}(G) \doteq \{A \in N \mid \forall (x, y), (x', y') \in F_G(A), y = y'\}.$$

Lemma 7. We can calculate $\text{SingleOut}(G)$ in time $O(|G| + v(G))$.

Proof. We assume that each nonterminal in $G$ generates at least one word. Associate to each $A \in N$ a word $w_A$, initially set to nil, and a boolean flag manyOutput$_A$, initially set to false, specifying whether or not we have found that $A$ produces more than one different output word. Associate also to each $A \in N$ a list called reverseRef$_A$ containing the references to all the rules $B \rightarrow \alpha A A \in P$ such that $\alpha = a_1 A a_2$, for some $a_1, a_2$. Next, associate to each rule a boolean flag which can take either the value marked or unmarked, initially set to unmarked. Associate also to each rule $A \rightarrow aa \in P$ an integer value $n_{(A \rightarrow aa)}$, initialized to the number of occurrences of different nonterminals in $\alpha$. The value $n_{(A \rightarrow aa)}$ keep tracks of the number of distinct nonterminals $B$ in $\alpha$ such that $w_B = \text{nil}$. This information can be precomputed in $O(|G|)$.

We can find which nonterminals generate more than one output word by iterating the following procedure:

- Consider each unmarked rule $A \rightarrow aa \in P$ such that $n_{(A \rightarrow aa)} = 0$ and manyOutput$_A = \text{false}$. If there is no such rule, terminate. Otherwise, set the rule as marked. Then, since $n_{(A \rightarrow aa)} = 0$, we know that $w_B \neq \text{nil}$ for every nonterminal $B$ in $\alpha$, and we can compute a word $w'$ for $A$.
- If $A$ already has an output word $w_A \neq w'$, then set manyOutput$_A := \text{true}$ and apply the following recursive procedure to reverseRef$_A$:
  
  For each rule $(B \rightarrow b \beta)$ in reverseRef$_A$: if manyOutput$_B = \text{false}$, set manyOutput$_B := \text{true}$ and apply recursively this procedure to reverseRef$_B$.
- Otherwise, if $w_A = \text{nil}$, set $w_A := w'$ and, for each $B \rightarrow b \beta$ in reverseRef$_A$, set $n_{(B \rightarrow b \beta)} := n_{(B \rightarrow b \beta)} - 1$.

This procedure takes time $O(|G| + v(G))$ since we consider an occurrence of a nonterminal in a rule a constant number of times. □
Proposition 8. Let \( G = (\Sigma, \Omega \cup \overline{\Omega}, N, P) \) be a simple function grammar, \( \alpha, \alpha', \beta, \beta' \in (N \cup \Omega \cup \overline{\Omega})^* \) such that \( ||\alpha|| \leq ||\beta|| \), \( A \in N \), and \( u, v \in (\Omega \cup \overline{\Omega})^* \).

1. \( F_G(\alpha) = F_G(\beta) \) iff \( \forall a \in \Sigma \ F_G(\text{Derived}(\alpha, a)) = F_G(\text{Derived}(\beta, a)) \) or \( (\alpha, \beta \in (\Omega \cup \overline{\Omega})^* \) and \( \alpha = \beta \).
2. \( F_G(\alpha \gamma) = F_G(\beta \gamma) \iff F_G(\alpha) = F_G(\beta) \) and \( F_G(\alpha') = F_G(\gamma \beta') \), where \( (w_a, u_a) = \min F_G(\alpha) \) and \( \gamma = u_a^{-1} \text{Derived}(\beta, w_u) \).

Proof. The first “if and only if” statement is obvious. The second statement follows from the fact that the monoid of simple functions is cancellative. The following cases are possible:

1. \( \gamma \) is such that \( F_G(\alpha \gamma) = F_G(\beta) \), i.e., \( F_G(\gamma) = (F_G(\alpha))^{-1} F_G(\beta) \). In this case the “if and only if” statement is straightforward.
2. If \( F_G(\alpha \gamma) \neq F_G(\beta) \) then \( (F_G(\alpha))^{-1} F_G(\beta) \) is not defined, and thus, assuming \( ||\alpha|| \leq ||\beta|| \), \( F_G(\alpha \gamma) \) cannot be equal to \( F_G(\beta \gamma) \). \( \square \)

Corollary 9. Let \( G \) be a simple function grammar, \( \alpha, \beta \in (N \cup \Omega \cup \overline{\Omega})^* \), \( A \in N \), and \( u, v \in (\Omega \cup \overline{\Omega})^* \).

\[ F_G(uA\alpha) = F_G(vA\beta) \iff F_G(\alpha) = F_G(\gamma \beta) \) and \( F_G(uA\gamma) = F_G(vA), \]

where \( (w_A, u_A) = \min F_G(\alpha) \) and \( \gamma = (u_A)^{-1} \text{Derived}(vA, w_A) = u_A^{-1} u_A^{-1} \text{Derived}(vA, w_A) \). Notice that \( A \in \text{SingleOut}(G) \) implies \( F_G(uA\gamma) = F_G(vA) \).

3. Equivalence algorithm

The algorithm \text{EQUIVALENCE} which checks for the equality of \( A \) and \( B \), consists of constructing a relation \( \mathcal{R} \subset (N \cup \Omega \cup \overline{\Omega})^+ \times (N \cup \Omega \cup \overline{\Omega})^+ \), which implies \( F_G(A) = F_G(B) \). In the terminology of [4], \( \mathcal{R} \) would be called a self-proving relation. In our case \( \mathcal{R} \) will consist of two relations \( \mathcal{D} \), called a decomposition relation, and \( \mathcal{C} \), called a conjugation relation.

Let \( \leq \) be a total order over nonterminals verifying \( A \leq B \Rightarrow ||A|| \leq ||B|| \).

\text{Decomposition relation and unfolding.}

The decomposition relation is a partial mapping \( \mathcal{D} : N \rightarrow (N \cup \Omega \cup \overline{\Omega})^+ \) such that \( \mathcal{D}(A) \in \{(X \in N \mid X < A) \cup \Omega \cup \overline{\Omega})^+ \), i.e., \( \mathcal{D}(A) \) contains only nonterminals smaller than \( A \). By \( \mathcal{D}^*(\beta) \) we denote the complete unfolding of \( \beta \). This means that if \( (A, \alpha) \in \mathcal{D} \) then \( A \) is replaced in \( \beta \) by \( \alpha \), such an operation is iterated until the resulting string \( \beta \) stabilizes. E.g., if \( \mathcal{D} = \{(A, BcB), (B, Cb)\} \) with \( A, B, C \in N \) and \( a, b, c \in (\Omega \cup \overline{\Omega}) \), then \( \mathcal{D}^*(aAA) = acbcbCbcBcB \).

\text{Conjugation relation.}

The relation \( \mathcal{C} \) contains conjugation equations of the form \( r_1 A = Ar_2 \), where \( A \in N \) and \( r_1, r_2 \in \Omega^\circ \). In \( \mathcal{C} \) we will keep only reduced non-trivial conjugation equations, i.e., we will assume that the nonterminal \( A \) present in the equation generates at least two different elements, \( A \notin \text{SingleOut}(G) \), and that \( r_1 \) and \( r_2 \) are primitive and reduced.

By Lemma 4, it is enough to keep in \( \mathcal{C} \) only one such conjugation equation per \( A \). Hence, the size of the conjugation relation \( |\mathcal{C}| \) is bounded by \( |N| \).

\text{Description of the algorithm.}

The algorithm is presented in Fig. 1. Intuitively, the algorithm constructs \( \mathcal{R} = \mathcal{C} \cup \mathcal{D} \), maintaining a list \( Q \) of equations on sequences over \( N \cup \Omega \cup \overline{\Omega} \), called targets. The targets are processed within a while-loop one by one until the set \( Q \) becomes empty, which is equivalent to a proof that the initial equation is true, or until a counter-example disproving the equivalence is found and \text{FAILURE} is reported. The processing of a target \( (\alpha_1, \alpha_2) \) from \( Q \) is as follows. Firstly, the target is normalized through the unfolding by \( \mathcal{D} \) and the removal of the common prefix from \( \mathcal{D}^*(\alpha_1) \) and \( \mathcal{D}^*(\alpha_2) \). If the normalized target is trivially true or false it is immediately treated as such. Otherwise, the target, which can be written \( (u_1 A \alpha_1', u_2 B \alpha_2') \), is split right after the first nonterminals creating new targets \( (u_1 A \gamma, u_2 B) \) and \( (\alpha_1', \gamma A \alpha_2') \), assuming \( A \leq B \). The second target is put back into \( Q \) for a later processing. The former target \( (u_1 A \gamma, u_2 B) \) is considered immediately, and is processed according to its format: if \( A = B \), it is added to \( \mathcal{C} \) unless it is already present, in which case it is compared to the existing value; if \( A \neq B \), it is added to \( \mathcal{D} \). If the target is added to either \( \mathcal{C} \) or \( \mathcal{D} \), the target is also derived by all terminal symbols and the resulting targets are added to \( Q \).
Algorithm EQUIVALENCE($A$, $B$):

--- Comment: $A$, $B \in N$, are two nonterminals of a simple function grammar $G = (\Sigma, \Omega, N, P)$. The algorithm returns SUCCESS iff $F_G(A) = F_G(B)$.

$Q := \{(A, B)\}; C := \emptyset; D := \emptyset; \$

while $Q$ is not empty do:

$(\alpha_1, \alpha_2) := \text{delete}(Q);$

if $\alpha_1 = \bot$ and $\alpha_2 = \bot$ then start the next iteration.

if $\alpha_1 = \bot$ or $\alpha_2 = \bot$ then return FAILURE.

$\alpha_1 \leftarrow D^*(\alpha_1), \alpha_2 \leftarrow D^*(\alpha_2)$ — Unfolding $\alpha_1$ and $\alpha_2$ by $D$.

Simplify $(\alpha_1, \alpha_2)$ by eliminating the common prefix.

if $\alpha_1 = \alpha_2 = \epsilon$ then start the next iteration.

if $\alpha_1$ or $\alpha_2$ is of output type then return FAILURE.

--- Comment: At this stage $\alpha_1$ and $\alpha_2$ are of general type, i.e., $\alpha_1 = u_1 A\alpha_1'$ and $\alpha_2 = u_2 B\alpha_2'$, and they differ syntactically on the first (nonterminal or output) symbol.

$u_1 \leftarrow \text{OutPref}(\alpha_1), A \leftarrow \text{First}(\alpha_1), \alpha_1' \leftarrow \text{Tail}(\alpha_1)$

$u_2 \leftarrow \text{OutPref}(\alpha_2), B \leftarrow \text{First}(\alpha_2), \alpha_2' \leftarrow \text{Tail}(\alpha_2)$

--- Comment: Without loss of generality, assume $A \leq B$.

$(w_A, u_A) \leftarrow \min F_G(A); \gamma \leftarrow u_A^{-1} u_1^{-1} \text{Derived}(u_2 B, w_A)$

if $\gamma = \bot$ then return FAILURE.

Add $(\alpha_1', \gamma u_2')$ to $Q$.

if $A = B$ then:

if $A \in \text{SingleOut}(G)$ then start the next iteration.

if $\text{power}(u_1^{-1} u_2) \neq \text{power}(\gamma)$ then return FAILURE.

$x \leftarrow \text{root}(u_1^{-1} u_2), y \leftarrow \text{root}(\gamma)$

--- Comment: Equation $u_1 A\gamma = u_2 A$ corresponds to conjugation $u_1^{-1} u_2 A = A\gamma$, and, by Lemma 4, to $xA = A\gamma$.

if $(r_1 A, Ar_2)$ is in $C$ then

if $(x = r_1$ and $y = r_2$) or $(x = r_1^{-1}$ and $y = r_2^{-1})$ then start the next iteration else return FAILURE.

Add $(xA, A\gamma)$ to $C$

For each $a \in \Sigma$ do:

$\beta_1 \leftarrow \text{Derived}(xA, a), \beta_2 \leftarrow \text{Derived}(A\gamma, a), \text{Add} (\beta_1, \beta_2)$ to $Q$

else --- Comment: $A < B$

Add $(B, u_2^{-1} u_1 A\gamma)$ to $D$

For each $a \in \Sigma$ do:

$\beta_1 \leftarrow \text{Derived}(B, a), \beta_2 \leftarrow \text{Derived}(u_2^{-1} u_1 A\gamma, a), \text{Add} (\beta_1, \beta_2)$ to $Q$

end [of while]

return SUCCESS.

---

3.1. Trace history of the algorithm

Let $G = ([0, 1], \{a, b\}, \{S, T, X, Y, Z\}, P)$ be a simple grammar with output, where $P$ is given by productions:

$S \rightarrow 1bZbaab, \ T \rightarrow 1Ybaay, \ X \rightarrow 0abba, \ X \rightarrow 1aYba,$

$Y \rightarrow 0bbaXab, \ Y \rightarrow 1bZbaab, \ Z \rightarrow 0baXaY, \ Z \rightarrow 1Zbaay.$

We show how EQUIVALENCE($S$, $T$) is computed.

We start by precomputing min $F_G(A)$, for $A \in \{S, T, X, Y, Z\}$:

$(w_S, u_S) = (w_T, u_T) = (10000, bbaabbaabbaabbaabbaab)$,

$(w_X, u_X) = (0, abba), (w_Y, u_Y) = (00, bbaabbaab), \text{and}

(w_Z, u_Z) = (0000, baabbaabbaabbaab).
We set $X < Y < Z < S < T$ since $||X|| = 1$, $||Y|| = 2$, $||Z|| = 4$, $||S|| = ||T|| = 5$. We have to precompute $\text{SingleOut}(G)$, which in our case is empty.

The initialization step sets $Q = \{(S, T), \emptyset\}$, $C = \{}$, and $D = \{}$.

The first iteration of the main while loop begins by retrieving the target $(\alpha_1, \alpha_2) = (S, T)$ from $Q$. The target is first simplified using $D$ which does not change its state since $D$ is empty. The longest common prefix of $\alpha_1$ and $\alpha_2$ is then removed, which again does not modify the target since $S$ and $T$ have no common prefix. Both $\alpha_1$ and $\alpha_2$ are of general type, therefore they are decomposed as $u_1.A.\alpha'_1 = \varepsilon.S.\varepsilon$ and $u_2.B.\alpha'_2 = \varepsilon.T.\varepsilon$. Then, since

$$(w_S, u_S) = (10000, bbaabbaabbaabbaabbaab)$$

we compute

$$\gamma = (bbaabbaabbaabbaabbaab)^{-1}\text{Derived}(T, 10000) = \varepsilon$$

and add $(\gamma, \varepsilon)$ to $Q$.

Finally, since the first nonterminal of $\alpha_1$ and $\alpha_2$ are different (i.e. $S \neq T$) and $T > S$, we add $(T, S)$ to $D$. We also compute $\text{Derived}(T, 0) = \bot$, $\text{Derived}(S, 0) = \bot$, $\text{Derived}(T, 1) = YbaaY$ and $\text{Derived}(S, 1) = bZbaab$, from which we set $Q = Q \cup \{(\bot, \bot), (YbaaY, bZbaab)\}$. This completes the iteration.

The full trace of the execution of the algorithm has been summarized in Fig. 2. The underlined elements in column $Q$ are the targets $(\alpha_1, \alpha_2)$ considered by the algorithm during the iteration. Column “simplified $(\alpha_1, \alpha_2)$” corresponds to the result of the simplification by $D$ followed by the removal of the longest common prefix; the result is written in the form $u_1.A.\alpha'_1, u_2.B.\alpha'_2$. Column $C$ contains the conjugation equations added (black ones) or checked for (gray ones) in the iteration. The last column, $D$, contains the decomposition added in the iteration.

### 3.2. Correctness of the EQUIVALENCE algorithm

In order to demonstrate that the algorithm is correct we will show that:

1. The algorithm always terminates.
2. The validity of the set of equations corresponding to $Q \cup D \cup C$ is an invariant at every iteration of the while loop.
3. The value FAILURE is reported only if the chosen target $(\alpha_1, \alpha_2) \in Q$ is such that $F_G(\alpha_1) \neq F_G(\alpha_2)$.
4. If $F_G(\alpha_1) \neq F_G(\alpha_2)$ for some $(\alpha_1, \alpha_2) \in Q$ then FAILURE is reported.

Let $||Q||$ denote the shortest-word complexity of $Q$, i.e.,

$$||Q|| \overset{\text{def}}{=} \sum ||\alpha|| + ||\beta|| \mid (\alpha, \beta) \in Q.$$

At every iteration which does not add anything to $D$ nor to $C$, the value $||Q||$ strictly decreases. The algorithm terminates since the number of insertions into $D$ and $C$ is bounded, hence $||Q||$ eventually decreases to 0 or FAILURE is reported.

The invariant of point 2 follows from Proposition 8, Corollary 9, and Lemma 4.

Point 3 can be checked by examining all five FAILURE reports present in the algorithm. The first two are obvious. The third occurrence follows from Proposition 8(2). The forth and fifth ones follow from Lemma 4.

The last point, item 4, stating that FAILURE is reported whenever $Q$ contains a pair of sequences which are not equivalent, is argued using the following proposition.

**Proposition 10.** Let $\alpha_1, \alpha_2 \in (N \cup \Omega \cup \overline{\Omega})^*$ and $w \in \Sigma^*$ be such that $F_G(\alpha_1)(w) \neq F_G(\alpha_2)(w)$. If at some point of the execution of the algorithm $(\alpha_1, \alpha_2)$ appears in $Q$ then the algorithm reports FAILURE.

**Proof.** By induction on the length of $w$.

If $|w| = 0$ then $\alpha_1$ or $\alpha_2$ is in $(\Omega \cup \overline{\Omega})^*$. Therefore, if $\alpha_1 \neq \alpha_2$ then FAILURE is reported in (2).

Assume that FAILURE is reported whenever $F_G(\alpha_1)(w) \neq F_G(\alpha_2)(w)$ with $|w| < k$. Consider $\alpha_1$ and $\alpha_2$ such that $F_G(\alpha_1)(w) \neq F_G(\alpha_2)(w)$ and $|w| = k$. There are three cases with respect to the shape of $\alpha_1$ and $\alpha_2$ (we will often write just $\alpha$, for $\alpha \in (N \cup \Omega \cup \overline{\Omega})^*$, as an abbreviation for $F_G(\alpha)$):

- $\alpha_1$ or $\alpha_2$ is of constant type.

  We report FAILURE in (2).

- $\alpha_1$ or $\alpha_2$ is context-free.
\( Q \) | simplified \((\alpha_1, \alpha_2)\) | \( \gamma \) | \( C \) | \( D \)  
---|---|---|---|---  
\((S, T)\) | \((\varepsilon, \varepsilon, \varepsilon, T.x)\) | \(\varepsilon\) | \((T, S)\)  
\((\varepsilon, \varepsilon), (\perp, \perp, \perp)\), \((\varepsilon, \varepsilon)\)  
\((\perp, \perp, \perp)\)  
\((\varepsilon, \varepsilon), (\perp, \perp, \perp), (\varepsilon, \varepsilon)\)  
\((\varepsilon, \varepsilon)\)  
\((\alpha, \alpha)\), \((\alpha, \alpha), (\perp, \perp, \perp, \perp)\), \((\alpha, \alpha)\)  
\((\alpha, \alpha), (\perp, \perp, \perp, \perp)\), \((\alpha, \alpha)\)  
\((\alpha, \alpha), (\perp, \perp, \perp, \perp)\), \((\alpha, \alpha)\)  
\((\alpha, \alpha), (\perp, \perp, \perp, \perp)\), \((\alpha, \alpha)\)  
\((\alpha, \alpha), (\perp, \perp, \perp, \perp)\), \((\alpha, \alpha)\)  
\((\alpha, \alpha), (\perp, \perp, \perp, \perp)\), \((\alpha, \alpha)\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\(\varepsilon, \varepsilon\)  
\emptyset  

Return SUCCESS

Fig. 2. Execution of EQUIVALENCE proving \( F_G(S) = F_G(T) \). In each iteration the active pair \((\alpha_1, \alpha_2)\) is underlined.

- \( \alpha_1 = u_1 A \alpha'_1 \) and \( \alpha_2 = u_2 A \alpha'_2 \).

Let \( \gamma = (xw_1 u_x)^{-1} u_2 A \), where \( (x, u_x) = \min A \). Since \( \alpha_1(w) \neq \alpha_2(w) \), there exists \( a \in \Sigma, w_A, w' \in \Sigma^* \) such that \( aw_A w' = w, aw_A \in L(A) \).

If \( \alpha'_1(w') \neq \gamma \alpha_2(w') \) then, by induction since \( |w'| < k \) and the fact that \((\alpha'_1, \gamma \alpha_2)\) is added to \( Q \), the algorithm reports FAILURE.

Otherwise, i.e., if \( \alpha'_1(w') = \gamma \alpha_2(w') \) then we have:

\[
\begin{align*}
  u_1 A \alpha'_1(aw_A w') &\neq u_2 A \alpha'_2(aw_A w') \\
u_1 A(aw_A) \alpha'_1(w') &\neq u_2 A(aw_A) \alpha'_2(w') \\
u_1 A(aw_A) \gamma &\neq u_2 A(aw_A) \\
u_1^{-1} u_2 A(aw_A) &\neq A \gamma (aw_A)
\end{align*}
\]

By Lemma 4, the inequality holds if and only if \( \text{power}(u_1^{-1} u_2) \neq \text{power}(\gamma) \) or \( a^{-1} \text{root}(u_1^{-1} u_2) A(aw_A) \neq a^{-1} A \text{root}(\gamma)(aw_A) \).
The inequality \( \text{power}(u_1^1u_2) \neq \text{power}(\gamma) \) is checked for in (5) and FAILURE is reported. Otherwise, \( (a^{-1}\text{root}(u_1^1u_2)A, a^{-1}\sqrt{\gamma}) \) is added to \( Q \). Hence, by the induction hypothesis, the program eventually reports failure, since \( |w_A| < k \).

- \( \alpha_1 = u_1A_1\alpha_1' \) and \( \alpha_2 = u_2A_2\alpha_2' \) with \( A_1 < A_2 \).

  Let \( \gamma = (u_1u_2)^{-1}u_2A_2 \), where \( (x, u_i) = \min A_1 \). If \( \gamma = \bot \) then in (3) we report FAILURE. Otherwise, we have two cases to consider:

  . One of \( A_1 \) and \( L(A_2) \), but not both, is not defined for any prefix of \( w \). Let \( aw_1 \) be the prefix of \( w \) such that \( A_1(aw_1) \) or \( A_2(aw_1) \) is defined. In such a case, \( a^{-1}u_1A_1\gamma(w_1) \neq a^{-1}u_2A_2(w_1) \), which is equivalent to \( a^{-1}u_2^{-1}u_1A_1\gamma(w_1) \neq a^{-1}A_2(w_1) \).

  . There exist \( w_1 \) and \( w_2 \) such that \( A_1(aw_1) \) and \( A_2(aw_1w_2w') \) are defined. Hence, \( w = aw_1w_2w' \) with \( a \in \Sigma \).

    If \( \alpha'_1(w_2w') \neq \gamma\alpha'_2(w_2w') \) then, by the induction hypothesis, the algorithm will report FAILURE.

    Otherwise, \( \alpha'_1(w_2w') = \gamma\alpha'_2(w_2w') \), and therefore

    \[
    u_1A_1\alpha'_1(aw_1w_2w') \neq u_2A_2\alpha'_2(aw_1w_2w')
    \]

    implies

    \[
    u_1A_1(aw)\gamma(w_2)\alpha'_2(w') \neq u_2A_2(aw_1w_2)\alpha'_2(w'),
    \]

    i.e., \( u_1A_1(aw)\gamma(w_2) \neq u_2A_2(aw_1w_2) \). \( \Box \)

More details on the proof of correctness may be found in [1].

3.3. Complexity

The efficiency of the algorithm follows from the fact that the number of insertions into \( D \) and \( C \) is polynomial.

**Proposition 11.** The algorithm \( \text{EQUIVALENCE}(A, B) \) works in polynomial time with respect to \( |G| + v(G) \).

**Proof.** Let \( k \overset{\text{def}}{=} \max\{|a| \mid (A \to a) \in P\} \), i.e., the length of a longest rule in \( P \). Therefore, for any \( A \in N \), \( (w, u) \in F_G(A) \) implies that \( |u| \leq k|w| \).

Since the number of insertions into \( D \) and \( C \) is \( O(|N|) \), in the worst case \( Q \) can contain \( O(|N||\Sigma|) \) targets \((\alpha, \beta)\). Notice that for all \((\alpha, \beta) \in Q \), \( \min(|\alpha|, ||\beta||) \leq k v(G) \). At that point no more targets can be added to \( Q \). Therefore, the number of iterations of the while loop is \( O(|N||\Sigma| k v(G)) \). Since the precomputing phase (calculating \( \min F_G(A), \) for all \( A \in N \), and calculating \( \text{SingleOut}(G), \) Lemma 7) takes polynomial time in \( |G| + v(G) \), and all operations in the algorithm are proportional to the size of the arguments (Lemma 3), the overall running time of the algorithm is polynomial. \( \Box \)

4. Conclusion

We showed an algorithm which tests the equality of two simple functions for a grammar \( G \) in time polynomial with respect to \( |G| + v(G) \) (the shortest-word complexity of \( G \)). In practical situations this algorithm works in polynomial time with respect to the size of \( G \), since usually \( v(G) \) is polynomial with respect to the size \( |G| \) of the grammar. However it is theoretically possible that \( v(G) \) is exponential with respect to \( |G| \).

Our algorithm is a step towards a fully polynomial (with respect only to \( |G| \) ) time algorithm.

References


