Equivalence of Functions Represented by Simple Context-Free Grammars with Output

Cédric Bastien[1], Jurek Czyzowicz[1], Wojciech Fraczak[1,2], and Wojciech Rytter[3]

[1] Dépt d'informatique, Université du Québec en Outaouais, Gatineau PQ, Canada
[2] IDT Canada Incorporation, Ottawa ON, Canada
[3] Institute of Informatics, Warsaw University, Warsaw, Poland

Abstract. A partial function $F : \Sigma^* \to \Omega^*$ is called a *simple* function if $F(w) \in \Omega^*$ is the output produced in the generation of a word $w \in \Sigma^*$ from a nonterminal of a simple context free grammar G with output alphabet Ω. In this paper we present an efficient algorithm for testing 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.

1 Introduction

The decidability problem of equivalence for functions defined by different classes of deterministic push-down automata and pushdown transducers (dpdt) was studied extensively, see for example [8, 11], 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. Simple functions were initially defined in [5] as the semantic domain of a network packet classification engine developed at IDT Canada, Inc. Simple functions are applied by IDT Canada to perform packet classification at wire speed. Classification policies are described with the aid of a class of context free grammars and implemented as so called *Concatenation State Machines*, a hardware implementation of single-state dpdts. In order to manage large sets of those classification policies in memory, it is useful to be able to identify if two classification policies are semantically the same. This was our motivation to investigate the problem of the equivalence of simple functions from a practical point of view and to develop an efficient and easy to implement algorithm for this task. The algorithm we propose in this paper is a nontrivial extension of the simple languages equivalence algorithms from [2, 7, 1] 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 [9] as languages recognized by a dpda with a single state, also called *simple dpda*, or, equivalently, as languages generated by *simple grammars*. We extend the definition of simple grammars to functions defined by grammars with output.

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

$$G = (\Sigma, \Omega, N, P),$$

where Σ, Ω, N are 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 $X \in N$ and $a \in \Sigma$ there is at most one $\alpha \in (N \cup \Omega)^*$, such that $(X, a, \alpha) \in P$.

Each production can be written as $A \to s\alpha$, where $\alpha \in (N \cup \Omega)^*$. We also write $A \xrightarrow{s} \alpha$. The relation $\xrightarrow{s}$ is extended in the following way.

We write $\alpha_1 \xrightarrow{s} \alpha_2$, iff $\alpha_1 = \beta_1 A \beta_2,\ \beta_1 \in \Omega^*,\ A \in N,\ \alpha_2 = \beta_1 \gamma \beta_2$ and $A \to s\,\gamma$ is a production. Intuitively, relation $\alpha_1 \xrightarrow{s} \alpha_2$ corresponds to a single-step leftmost derivation.

For $w = s_1 s_2 \dots s_n$ and $\alpha_i \in (N \cup \Omega)^*$ we write $\alpha_0 \xrightarrow{w} \alpha_n$ iff

$$\alpha_0 \xrightarrow{s_1} \alpha_1,\ \alpha_1 \xrightarrow{s_2} \alpha_2,\ \alpha_2 \xrightarrow{s_3} \alpha_3,\ \dots \alpha_{n-1} \xrightarrow{s_n} \alpha_n.$$

For $w \in \Sigma^*$ we write:

$$\beta = \texttt{Derived}(\alpha, w) \iff \alpha \xrightarrow{w} \beta, \text{ where } \beta \in (N \cup \Omega)^*.$$

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

$$F_G(\alpha) \stackrel{\texttt{def}}{=} \{\ (w, u) \in \Sigma^* \times \Omega^* \mid u = \texttt{Derived}(\alpha, w),\ u \neq \perp\ \}.$$

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 Ω 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 grammar with output $G = (\{0, 1\}, \{a, b\}, \{S_1, S_2, A_1, A_2\}, P)$, where P is given by rules:

$$S_1 \to 0aS_1A_1b, \quad S_1 \to 1, \quad A_1 \to 1, \quad S_2 \to 0aS_2A_2, \quad S_2 \to 1, \quad A_2 \to 1b$$

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:

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

Let $\alpha \in (N \cup \Omega)^*$. By $||\alpha||$ we denote the *shortest-word complexity* of α 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 $v(G) \overset{\mathtt{def}}{=} \max\{||A|| \mid A \in N\}$. $|G|$ denotes the size of the textual description of G.

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

Theorem 1. *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 $|G| + v(G)$.*

2 Free group over Ω and properties of simple functions

In the course of the algorithm we consider, as intermediate data, output sequences which are to be compensated 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 $\Omega^{\otimes}$ over the output alphabet Ω. 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 Ω.

By ε we denote an empty sequence. Simple functions together with concatenation defined by $fg \overset{\mathtt{def}}{=} \{(x_1x_2, y_1y_2) \mid (x_1, y_1) \in f, (x_2, y_2) \in g\}$, constitute a monoid with $\{(\varepsilon, \varepsilon)\}$ acting as unit[1] and with $\perp \overset{\mathtt{def}}{=} \{\}$ acting as zero. More details about simple functions seen as a monoid can be found in [5]. We will write w and u instead of $\{(w, \varepsilon)\}$ and $\{(\varepsilon, u)\}$, with $w \in \Sigma^*$ and $u \in \Omega^*$, respectively. In particular, the unit function $\{(\varepsilon, \varepsilon)\}$ will be denoted by ε.

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 Ω, denoted by $\Omega^{\otimes}$, so $w^{-1}f$ would be such that $w(w^{-1}f) = f$, for all $w \in \Omega^{\otimes}$. More precisely, $\Omega^{\otimes} \overset{\mathtt{def}}{=} (\Omega \cup \overline{\Omega})^*_{/\{a\overline{a}=\overline{a}a=\varepsilon | a \in \Omega\}}$, where $\overline{\Omega}$ is a copy of Ω with bijection $\bar{\ } : \Omega \mapsto \overline{\Omega}$ playing the role of the inverse. Therefore, apart from monoid properties, i.e., $x(yz) = (xy)z$, $x\varepsilon = \varepsilon x = x$, we have $a\overline{a} = \overline{a}a = \varepsilon$, for every $a \in \Omega$. For example, $(ab\overline{c})^{-1} = c\overline{b}\overline{a}$, or $bc(abc)^{-1} = \overline{a}$.

Given two strings u, v over $\Omega \cup \overline{\Omega}$, we write $u = v$ to say that they are equivalent in $\Omega^{\otimes}$. 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 $\Omega^{\otimes}$, i.e., an

[1] The unit simple function corresponds to $F_G(\varepsilon)$.

equivalence class over $(\Omega \cup \overline{\Omega})^*$, is to choose the shortest string from the class (such a word does not contain subwords $a\overline{a}$ or $\overline{a}a$, for any $a \in \Omega$). Given a string $u \in (\Omega \cup \overline{\Omega})^*$, by $\texttt{reduce}(u)$ we denote the shortest string over $\Omega \cup \overline{\Omega}$, such that $\texttt{reduce}(u) = u$.

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

Lemma 1. *Let $u \in (\Omega \cup \overline{\Omega})^*$. There is an algorithm for calculating $\texttt{power}(u)$ and $\texttt{root}(u)$ 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 $\texttt{reduce}(y^k) \equiv y^k$ for all $k \geq 1$ and u_2 can be treated as a word in a free monoid generated by the alphabet $(\Omega \cup \overline{\Omega})$. In this context, computing $\texttt{power}(u_2)$ and $\texttt{root}(u_2)$ 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 [4] for details), from which we can deduce the values of $\texttt{power}(u)$ and $\texttt{root}(u)$. $\square$

Lemma 2. *Let $X \subseteq \Omega^*$, $r_1, r_2 \in \Omega^{\otimes}$, such that $|X| \geq 2$, $r_1 X = X r_2$, and r_1, r_2 being primitive. For every $u, v \in \Omega^{\otimes}$, $uX = Xv$ iff $\texttt{power}(u) = \texttt{power}(v)$ and $(\texttt{root}(u), \texttt{root}(v)) \in \{(r_1, r_2), (r_1^{-1}, r_2^{-1})\}$.*

Proof. Firstly, we show that in $\Omega^{\otimes}$, $uw = wv$ iff $u = st$, $v = ts$ and $w = (st)^k s$ for some $s, t \in \Omega^{\otimes}$ and $k \in \mathbb{Z}$. Furthermore, if u and v are primitives, then s and t are unique. Using these facts, it is straightforward to prove the *if* part of the lemma.

Secondly, we observe that if $uX = Xv$ then $\texttt{root}(u)X = X\texttt{root}(v)$ and $\texttt{power}(u) = \texttt{power}(v)$. We elaborate a deterministic method of finding primitive words s and t from two words $w_1, w_2 \in X$ such that $r_1 = st, r_2 = ts$ and $w_i = (st)^{k_i} s$, for $i \in 1, 2$ and $k_i \in \mathbb{Z}$. Hence, there exists exactly one (modulo inverse) pair of primitive words r_1 and r_2 such that $r_1 X = X r_2$. $\square$

Example 2. Note that, if $|X| = 1$ then the lemma is not true. E.g., for $X = \{\varepsilon\}$, $uX = Xu$ for all $u \in \Omega^{\otimes}$.

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

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

Example 3. Let $\alpha = baabaAbaCAb$, where $a, b \in \Omega$, then $\texttt{First}(\alpha) = A$, $\texttt{OutPref}(\alpha) = baaba$, and $\texttt{Tail}(\alpha) = baCAb$.

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 $(w_A, u_A) \stackrel{\text{def}}{=} \min F_G(A)$ in time polynomial with respect to $||A||$.

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

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

Lemma 3. *We can calculate* $\texttt{SingleOut}(G)$ *in time* $O(|G| + v(G))$.

Proof. We assume G to be reduced. Associate with each rule $A \to a\alpha$ an integer value $n_{A \to a\alpha}$, initialized to the number of occurrences of different nonterminals which are in α, and associate also a boolean flag which can take either the value *marked* or *unmarked*, initially set to *unmarked*. Furthermore, assume that we have for each $A \in N$ a reference to all the rules $B \to a\alpha$ such that $\alpha = \alpha_1 A \alpha_2$, for some α_1, α_2. We also associate a word w_A to each A, initially set to *nil*, and a boolean flag specifying whether or not we have found that A produces more than one output. This information can be precomputed in $O(|G|)$.

Following this preprocessing, we can find for each nonterminal whether it generates more than one output word by iterating the following procedure:

> Consider all the *unmarked* rules $A \to a\alpha \in P$ such that $n_{A \to a\alpha} = 0$. If there is no such rule, terminate. Otherwise, set each such rule as *marked*. Then, for each of these rules, we know that $w_B \neq nil$ for every nonterminal B in α, and we can easily compute a word w' for A.
> If A already has an output word $w_A \neq w'$ associated with it, we have found that A generates more than one output word and we can mark it as such. We also recursively propagate this information to any nonterminal B such that $B \to a\alpha$, with $\alpha = \alpha_1 A \alpha_2$ (unless B is already marked as such), since B must necessarily generate at least 2 different output words.
> Otherwise, if $w_A = nil$, set $w_A := w'$ and subtract 1 from $n_{B \to a\alpha}$.

This procedure takes time $O(|G| + v(G))$ since we consider an occurrence of a nonterminal in a rule a constant number of times. $\square$

Proposition 1. *Let $G = (\Sigma, \Omega \cup \overline{\Omega}, N, P)$ be a simple function grammar, α, α', β, $\beta' \in (N \cup \Omega \cup \overline{\Omega})^*$ such that $||\alpha|| \leq ||\beta||$, $A \in N$, and $u, v \in (\Omega \cup \overline{\Omega})^*$.*

1. $F_G(\alpha) = F_G(\beta)$ *iff* ($\alpha, \beta \in (\Omega \cup \overline{\Omega})^*$ *and* $\alpha = \beta$) *or* $\forall a \in \Sigma$ $F_G(\texttt{Derived}(\alpha, a)) = F_G(\texttt{Derived}(\beta, a))$
2. $F_G(\alpha\alpha') = F_G(\beta\beta')$ *iff* $F_G(\alpha\gamma) = F_G(\beta)$ *and* $F_G(\alpha') = F_G(\gamma\beta')$, *where* $(w_\alpha, u_\alpha) = \min F_G(\alpha)$ *and* $\gamma = u_\alpha^{-1}\texttt{Derived}(\beta, w_\alpha)$.

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. γ 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 $||\alpha|| \leq ||\beta||$, $F_G(\alpha\alpha')$ cannot be equal to $F_G(\beta\beta')$. $\square$

Corollary 1. *Let G be a simple function grammar, $\alpha, \beta \in (N \cup \Omega \cup \overline{\Omega})^*$, $A \in N$, and $u, v \in (\Omega \cup \overline{\Omega})^*$.*

$$F_G(uA\alpha) = F_G(vA\beta) \quad \textit{iff} \quad F_G(\alpha) = F_G(\gamma\beta) \textit{ and } F_G(uA\gamma) = F_G(vA),$$

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

3 Equivalence algorithm

The algorithm 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 terminology of [3], $\mathcal{R}$ would be called *self-proving relation*. In our case $\mathcal{R}$ will consist of two relations $\mathcal{D}$, called *decomposition* relation, and $\mathcal{C}$, called *conjugation* relation.

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

Decomposition relation and *unfolding*. Decomposition relation is a partial mapping $\mathcal{D} : N \to (N \cup \Omega \cup \overline{\Omega})^+$ 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 β. This means that if $(A, \alpha) \in \mathcal{D}$ then A is replaced in β by α, such an operation is iterated until the resulting string β 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 $\mathcal{D}^*(aAA) = aCbcCbCbcCb$.

Conjugation relation. The relation $\mathcal{C}$ contains *conjugation equations* of form $r_1A = Ar_2$, where $A \in N$ and $r_1, r_2 \in \Omega^{\otimes}$. 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, $A \notin \texttt{SingleOut}(G)$, and that r_1 and r_2 are primitive and reduced. By Lemma 2, it is enough to keep in $\mathcal{C}$ only one such conjugation equation per A. Hence, the size of conjugation relation $|\mathcal{C}|$ is bounded by $|N|$.

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

Algorithm $\textsc{Equivalence}(A, B)$; $\{A, B \in N;\}$
$\{$*the algorithm returns* `SUCCESS` *iff* $F_G(A) = F_G(B)\}$
$Q := \{(A, B)\}$; $\mathcal{C} := \emptyset$; $\mathcal{D} := \emptyset$;

while Q is not empty **do**:
$(\alpha_1, \alpha_2) := \mathit{delete}(Q)$;
If $\alpha_1 = \perp$ and $\alpha_2 = \perp$ **then** start the next iteration.
If $\alpha_1 = \perp$ or $\alpha_2 = \perp$ **then return** `FAILURE`.

$\alpha_1 \leftarrow \mathcal{D}^*(\alpha_1)$, $\alpha_2 \leftarrow \mathcal{D}^*(\alpha_2)$ — *Unfolding* α_1 *and* α_2 *by* $\mathcal{D}$.
Simplify (α_1, α_2) by eliminating the common prefix.
If $\alpha_1 = \alpha_2 = \varepsilon$ **then** start the next iteration. (1)
If α_1 or α_2 is of *output* type **then return** `FAILURE`. (2)

— **Comment**: *At this stage* α_1 *and* α_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.*

$u_1 \leftarrow$ `OutPref`(α_1), $A \leftarrow$ `First`(α_1), $\alpha_1' \leftarrow$ `Tail`(α_1)
$u_2 \leftarrow$ `OutPref`(α_2), $B \leftarrow$ `First`(α_2), $\alpha_2' \leftarrow$ `Tail`(α_2)
— **Comment**: *Without loss of generality, assume* $A \leq B$.
$(w_A, u_A) \leftarrow \min F_G(A)$; $\gamma \leftarrow u_A^{-1} u_1^{-1}$`Derived`$(u_2\, B, w_A)$
If $\gamma = \perp$ **then return** `FAILURE`. (3)
Add $(\alpha_1', \gamma\alpha_2')$ to Q. (4)

If $A = B$ **then**:
If $A \in$ `SingleOut`(G) **then** start the next iteration.
If `power`$(u_1^{-1}u_2) \neq$ `power`(γ) **then return** `FAILURE`. (5)
$x \leftarrow$ `root`$(u_1^{-1}u_2)$, $y \leftarrow$ `root`(γ)
— **Comment**: *Equation* $u_1 A \gamma = u_2 A$ *corresponds to conjugation* $u_1^{-1}u_2 A = A\gamma$, *and, by Lemma 2, to* $xA = Ay$.

If $(r_1 A, A r_2)$ is in $\mathcal{C}$ **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, Ay) to $\mathcal{C}$
For each $a \in \Sigma$ **do**:
$\beta_1 \leftarrow$ `Derived`(xA, a), $\beta_2 \leftarrow$ `Derived`(Ay, a), Add (β_1, β_2) to Q
else — **Comment**: $A < B$ (6)
Add $(B, u_2^{-1} u_1 A \gamma)$ to $\mathcal{D}$
For each $a \in \Sigma$ **do**:
$\beta_1 \leftarrow$ `Derived`(B, a), $\beta_2 \leftarrow$ `Derived`$(u_2^{-1} u_1 A \gamma, a)$, Add (β_1, β_2) to Q
end {of while}

return `SUCCESS`.

Fig. 1. Simple Function Equivalence algorithm.

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 `FAILURE` is reported. Intuitively, the processing of a target (α_1, α_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 trivialy true or false it is immediatly 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 $(\alpha_1', \gamma\alpha_2')$, 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 immediatly, 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.

3.1 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 $Q \cup \mathcal{D} \cup \mathcal{C}$ 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.,

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

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

The invariant of point 2 follows from Proposition 1, Corollary 1, and Lemma 2.

Point 3 can be checked by examining all five `FAILURE` reports present in the algorithm. First two are obvious. The third occurrence follows from Proposition 1(2). The forth and fifth ones follow from Lemma 2.

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 2. *Let $\alpha_1, \alpha_2 \in (N \cup \Omega \cup \Omega)^*$ and $w \in \Sigma^*$ such that $F_G(\alpha_1)(w) \neq F_G(\alpha_2)(w)$. If at some point of the execution of the algorithm (α_1, α_2) appears in Q then the algorithm reports* `FAILURE`.

Proof. By induction on the length of w.

If $|w| = 0$ then α_1 or α_2 is in $(\Omega \cup \Omega)^*$. 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 α_1 and α_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 α_1 and α_2 (we will often write just α, for $\alpha \in (N \cup \Omega \cup \Omega)^*$, as an abbreviation for $F_G(\alpha)$):

- α_1 or α_2 is of constant type.
  We report `FAILURE` in (2).
- $\alpha_1 = u_1 A \alpha'_1$ and $\alpha_2 = u_2 A \alpha'_2$.
  Let $\gamma = (xu_1u_x)^{-1}u_2A$, 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 $aw_Aw' = w$, $aw_A \in L(A)$.
  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:
  $$\begin{aligned} u_1A\alpha'_1(aw_Aw') &\neq u_2A\alpha'_2(aw_Aw') \\ u_1A(aw_A)\alpha'_1(w') &\neq u_2A(aw_A)\alpha'_2(w') \\ u_1A(aw_A)\gamma &\neq u_2A(aw_A) \\ u_1^{-1}u_2A(aw_A) &\neq A\gamma(aw_A) \end{aligned}$$
  By Lemma 2, the inequality holds if and only if $\mathtt{power}(u_1^{-1}u_2) \neq \mathtt{power}(\gamma)$ or $a^{-1}\mathtt{root}(u_1^{-1}u_2)A(w_A) \neq a^{-1}A\mathtt{root}(\gamma)(w_A)$.
  The inequality $\mathtt{power}(u_1^{-1}u_2) \neq \mathtt{power}(\gamma)$ is checked for in (5) and `FAILURE` is reported. Otherwise, $(a^{-1}\mathtt{root}(u_1^{-1}u_2)A, a^{-1}A\mathtt{root}(\gamma))$ is added to Q. Hence, by the induction hypothesis, the program eventually reports failure, since $|w_A| < k$.
- $\alpha_1 = u_1 A_1 \alpha'_1$ and $\alpha_2 = u_2 A_2 \alpha'_2$ with $A_1 < A_2$.
  Let $\gamma = (xu_1u_x)^{-1}u_2A_2$, where $(x, u_x) = \min A_1$.
  If $\gamma = \perp$ 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_γ such that $A_1(aw_1)$ and $A_2(aw_1w_\gamma)$. Hence, $w = aw_1w_\gamma w'$ with $a \in \Sigma$.
    If $\alpha'_1(w_\gamma w') \neq \gamma\alpha'_2(w_\gamma w')$ then, by induction hypothesis, the algorithm will report `FAILURE`.

    Otherwise, $\alpha'_1(w_\gamma w') = \gamma\alpha'_2(w_\gamma w')$, and therefore
    $$u_1A_1\alpha'_1(aw_1w_\gamma w') \neq u_2A_2\alpha'_2(aw_1w_\gamma w') \text{ implies}$$
    $$u_1A_1(aw)\gamma(w_\gamma)\alpha'_2(w') \neq u_2A_2(aw_1w_\gamma)\alpha'_2(w'),$$
    i.e., $u_1A_1(aw)\gamma(w_\gamma) \neq u_2A_2(aw_1w_\gamma)$. $\square$

3.2 Complexity

The efficiency of the algorithm follows from the fact that the number of insertions into $\mathcal{D}$ and $\mathcal{C}$ is polynomial.

Proposition 3. *The algorithm* EQUIVALENCE(A, B) *works in polynomial time with respect to* $|G| + v(G)$.

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

Since the number of insertions into $\mathcal{D}$ and $\mathcal{C}$ is $O(|N|)$, in the worst case Q can contain $O(|N||\Sigma|)$ targets (α, β). Notice that for all $(\alpha, \beta) \in Q$, $\min(||\alpha||, ||\beta||) \le k\, v(G)$. 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 $\texttt{SingleOut}(G)$, Lemma 3) takes polynomial time in $|G| + v(G)$, and all operations in the algorithm are proportional to the size of the arguments (Lemma 1), the overall running time of the algorithm is polynomial. $\square$

3.3 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:

$$S \to 1bZbaab,\ T \to 1YbaaY,\ X \to 0abba,\ X \to 1aYba,$$
$$Y \to 0bbaXab,\ Y \to 1bZbaab,\ Z \to 0baXaY,\ Z \to 1ZbaaY.$$

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

We start by precomputing $\min F_G(A)$, for $A \in \{S,T,X,Y,Z\}$:
$(w_S, u_S) = (w_T, u_T) = (10000, bbaabbaabbaabbaabbaab)$,
$(w_X, u_X) = (0, abba)$, $(w_Y, u_Y) = (00, bbaabbaab)$, 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 also to precompute $\texttt{SingleOut}(G)$, which in our case is empty.

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

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 $\mathcal{D}$ which does not change its state since $\mathcal{D}$ is empty. The longest common prefix of α_1 and α_2 is then removed, which again does not modify the target since S and T have no common prefix. Both α_1 and α_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

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

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

Finally, since the first nonterminal of α_1 and α_2 are different (i.e. $S \neq T$) and $T > S$, we add (T, S) to $\mathcal{D}$. We also compute $\texttt{Derived}(T, 0) = \perp$, $\texttt{Derived}(S, 0) = \perp$, $\texttt{Derived}(T, 1) = YbaaY$ and $\texttt{Derived}(S, 1) = bZbaab$, from which we set $Q = Q \cup \{(\perp, \perp), (YbaaY, bZbaab)\}$. This completes the iteration.

Q	simplified (α_1, α_2)	γ	$\mathcal{C}$	$\mathcal{D}$
$\underline{(S, T)}$	$(\varepsilon.S.\varepsilon, \varepsilon.T.\varepsilon)$	ε		(T, S)
$\underline{(\varepsilon, \varepsilon)}$, $(\perp, \perp)$, $(YbaaY, bZbaab)$	$(\varepsilon, \varepsilon)$			
$\underline{(\perp, \perp)}$, $(YbaaY, bZbaab)$				
$\underline{(YbaaY, bZbaab)}$	$(\varepsilon.Y.baaY, b.Z.baab)$	$\overline{b}Y$		$(Z, \overline{b}Y\overline{b}Y)$
$\underline{(baaY, \overline{b}Ybaab)}$, $(baXaY, baXaY)$, $(ZbaaY, ZbaaY)$	$(baa.Y.\varepsilon, \overline{b}.Y.baab)$	$\overline{b}\overline{a}\overline{a}\overline{b}$	$(\overline{a}\overline{a}\overline{b}\overline{b}Y, Y\overline{b}\overline{a}\overline{a}\overline{b})$	
$\underline{(baXaY, baXaY)}$, $(ZbaaY, ZbaaY)$, $(\varepsilon, \varepsilon)$, $(\overline{a}Xab, bbaX\overline{a}\overline{b})$, $(\overline{a}\overline{a}\overline{b}Zbaab, bZ)$	$(\varepsilon, \varepsilon)$			
$\underline{(ZbaaY, ZbaaY)}$, $(\varepsilon, \varepsilon)$, $(\overline{a}Xab, bbaX\overline{a}\overline{b})$, $(\overline{a}\overline{a}\overline{b}Zbaab, bZ)$	$(\varepsilon, \varepsilon)$			
$\underline{(\varepsilon, \varepsilon)}$, $(\overline{a}Xab, bbaX\overline{a}\overline{b})$, $(\overline{a}\overline{a}\overline{b}Zbaab, bZ)$	$(\varepsilon, \varepsilon)$			
$\underline{(\overline{a}Xab, bbaX\overline{a}\overline{b})}$, $(\overline{a}\overline{a}\overline{b}Zbaab, bZ)$	$(\overline{a}.X.ab, bba.X.\overline{a}\overline{b})$	$abba$	$(abbaX, Xabba)$	
$\underline{(\overline{a}\overline{a}\overline{b}Zbaab, bZ)}$, (ab, ab), $(abbaabba, abbaabba)$, $(abbaaYba, aYbaabba)$	$(\overline{a}\overline{a}\overline{b}\overline{b}.Y.\overline{b}Ybaab, \varepsilon.Y.\overline{b}Y)$	$baab$	$(bbaaY, Ybaab)$	
$\underline{(ab, ab)}$, $(abbaabba, abbaabba)$, $(abbaaYba, aYbaabba)$, $(\overline{b}Ybaab, baaY)$	$(\varepsilon, \varepsilon)$			
$\underline{(abbaabba, abbaabba)}$, $(abbaaYba, aYbaabba)$, $(\overline{b}Ybaab, baaY)$	$(\varepsilon, \varepsilon)$			
$\underline{(abbaaYba, aYbaabba)}$, $(\overline{b}Ybaab, baaY)$	$(bbaa.Y.ba, \varepsilon.Y.baabba)$	$\overline{b}\overline{a}\overline{a}\overline{b}$	$(\overline{a}\overline{a}\overline{b}\overline{b}Y, Y\overline{b}\overline{a}\overline{a}\overline{b})$	
$\underline{(\overline{b}Ybaab, baaY)}$, (ba, ba)	$(\overline{b}.Y.baab, baa.Y.\varepsilon)$	baab	$(bbaaY, Ybaab)$	
$\underline{(ba, ba)}$,$(baab, baab)$	$(\varepsilon, \varepsilon)$			
$\underline{(baab, baab)}$	$(\varepsilon, \varepsilon)$			
empty	Return SUCCESS			

Fig. 2. Execution of EQUIVALENCE proving $F_G(S) = F_G(T)$. In each iteration the *active* pair (α_1, α_2) is underlined.

The full trace of the execution of the algorithm has been summarized in Fig. 2. The underlined elements in column Q are the targets (α_1, α_2) considered by the algorithm during the iteration. Column "*simplified* (α_1, α_2)" corresponds to the result of the simplification by $\mathcal{D}$ 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 $\mathcal{C}$ contains the conjugation equations added (black ones) or checked for (gray ones) in the iteration. The last column, $\mathcal{D}$, contains the decomposition added in the iteration.

4 Conclusions

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 first step towards a fully polynomial (with respect only to $|G|$) time algorithm.

References

1. Cédric Bastien, Jurek Czyzowicz, Wojciech Fraczak, and Wojciech Rytter. Prime normal form and equivalence of simple grammars. In *CIAA 2005*, volume 3845 of *LNCS*, pages 78–89. Springer, 2006.
2. Didier Caucal. A fast algorithm to decide on simple grammars equivalence. In *Optimal Algorithms*, volume 401 of *LNCS*, pages 66–85. Springer, 1989.
3. Bruno Courcelle. An axiomatic approach to the Korenjak-Hopcroft algorithms. *Mathematical Systems Theory*, 16(3):191–231, 1983.
4. Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.
5. Wojciech Debski and Wojciech Fraczak. Concatenation state machines and simple functions. In *CIAA 2004*, volume 3317 of *LNCS*, pages 113–124. Springer, 2004.
6. M.A. Harrison. *Introduction to formal language theory*. Addison Wesley, 1978.
7. Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1–2):143–159, 1996.
8. Oscar H. Ibarra and Louis E. Rosier. On the decidability of equivalence of deterministic pushdown transducers. *Information Processing Letters*, 13(3):89–93, 1981.
9. A. J. Korenjak and J. E. Hopcroft. Simple deterministic languages. In *Proc. IEEE 7th Annual Symposium on Switching and Automata Theory*, IEEE Symposium on Foundations of Computer Science, pages 36–46, 1966.
10. Lothaire. *Combinatorics on words*. Cambridge University Press, United Kingdom, 1997.
11. Gérard Sénizergues. $T(A) = T(B)$?. In *ICALP'99*, volume 1644 of *LNCS*, pages 665–675. Springer, 1999.