Acta Universitatis Sapientiae

Informatica

Volume 3, Number 1, 2011

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

Acta Univ. Sapientiae, Informatica, **3,** 1 (2011) 5–34

Primitive words and roots of words

Gerhard LISCHKE
Fakultät für Mathematik und Informatik
Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 1-4, D-07743 Jena, Germany
email: gerhard.lischke@uni-jena.de

Abstract. In the algebraic theory of codes and formal languages, the set Q of all primitive words over some alphabet Σ has received special interest. With this survey article we give an overview about relevant research to this topic during the last twenty years including own investigations and some new results. In Section 1 after recalling the most important notions from formal language theory we illustrate the connection between coding theory and primitive words by some facts. We define primitive words as words having only a trivial representation as the power of another word. Nonprimitive words (without the empty word) are exactly the periodic words. Every nonempty word is a power of an uniquely determined primitive word which is called the root of the former one. The set of all roots of nonempty words of a language is called the root of the language. The primitive words have interesting combinatorial properties which we consider in Section 2. In Section 3 we investigate the relationship between the set Q of all primitive words over some fixed alphabet and the language classes of the Chomsky Hierarchy and the contextual languages over the same alphabet. The computational complexity of the set Q and of the roots of languages are considered in Section 4. The set of all powers of the same degree of all words from a language is the power of this language. We examine the powers of languages for different sets of exponents, and especially their regularity and context-freeness, in Section 5, and the decidability of appropriate questions in Section 6. Section 7 is dedicated to several generalizations of the notions of periodicity and primitivity of words.

Computing Classification System 1998: F.4.3
Mathematics Subject Classification 2010: 03-2, 68-02, 68Q45, 68R15, 03D15
Key words and phrases: primitivity of words, periodicity of words, roots of words and languages, powers of languages, combinatorics on words, Chomsky hierarchy, contextual languages, computational complexity, decidability

1 Preliminaries

1.1 Words and languages

First, we repeat the most important notions which we will use in our paper.

Σ should be a fixed alphabet, which means, it is a finite and nonempty set of symbols. Mostly, we assume that it is a **nontrivial** alphabet, which means that it has at least two symbols which we will denote by a and b, $a \neq b$. $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ denotes the set of all natural numbers. Σ^* is the free monoid generated by Σ or the set of all words over Σ. The number of letters of a word p, with their multiplicities, is the **length** of the word p, denoted by $|p|$. If $|p| = n$ and $n = 0$, then p is the **empty word**, denoted by ϵ (in other papers also by e or λ). The set of words of length n over Σ is denoted by Σ^n. Then $\Sigma^* = \bigcup\limits_{n \in \mathbb{N}} \Sigma^n$ and $\Sigma^0 = \{\epsilon\}$. For the set of nonempty words over Σ we will use the notation $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$.

The **concatenation** of two words $p = x_1 x_2 \cdots x_m$ and $q = y_1 y_2 \cdots y_n$, $x_i, y_j \in \Sigma$, is the word $pq = x_1 x_2 \cdots x_m y_1 y_2 \cdots y_n$. We have $|pq| = |p| + |q|$. The **powers** of a word $p \in \Sigma^*$ are defined inductively: $p^0 = \epsilon$, and $p^n = p^{n-1}p$ for $n \geq 1$. p^* denotes the set $\{p^n : n \in \mathbb{N}\}$, and $p^+ = p^* \setminus \{\epsilon\}$.

For $p \in \Sigma^*$ and $1 \leq i \leq |p|$, $p[i]$ is the letter at the i-th position of p.
Then $p = p[1]p[2] \cdots p[|p|]$.

For words $p, q \in \Sigma^*$, p is a **prefix of** q, in symbols $p \sqsubseteq q$, if there exists $r \in \Sigma^*$ such that $q = pr$. p is a **strict prefix of** q, in symbols $p \sqsubset q$, if $p \sqsubseteq q$ and $p \neq q$. $\mathrm{Pr}(q) =_{\mathrm{Df}} \{p : p \sqsubset q\}$ is the **set of all strict prefixes of** q (including ϵ if $q \neq \epsilon$).
p is a **suffix of** q, if there exists $r \in \Sigma^*$ such that $q = rp$.

For an arbitrary set M, $|M|$ denotes the cardinality of M, and $\mathcal{P}(M)$ denotes the set of all subsets of M.

A **language over** Σ or a **formal language over** Σ is a subset L of Σ^*.
$\{L : L \subseteq \Sigma^*\} = \mathcal{P}(\Sigma^*)$ is the set of all languages over Σ. If L is a nonempty strict subset of Σ^*, $L \subset \Sigma^*$, then we call it a nontrivial language.

For languages L_1, L_2, and L we define:
$L_1 \cdot L_2 = L_1 L_2 =_{\mathrm{Df}} \{pq : p \in L_1 \wedge q \in L_2\}$,
$L^0 =_{\mathrm{Df}} \{\epsilon\}$, and $L^n =_{\mathrm{Df}} L^{n-1} \cdot L$ for $n \geq 1$.
If one of L_1, L_2 is a one-element set $\{p\}$, then, usually, in $L_1 L_2$ we write p instead of $\{p\}$.

Languages can be classified in several ways, for instance according to the Chomsky hierarchy, which we will assume the reader to be familiar with (otherwise, see, for instance, in [8, 9, 23]). These are the classes of regular, context-free, context-sensitive, and enumerable languages, respectively. Later on we will also consider linear languages and contextual languages and define them in Section 3.

1.2 Periodic words, primitive words, and codes

Two of the fundamental problems of the investigations of words and languages are the questions how a word can be decomposed and whether words are powers of a common word. These occur for instance in coding theory and in the longest repeating segment problem which is one of the most important problems of sequence comparing in molecular biology. The study of primitivity of sequences is often the first step towards the understanding of sequences.

We will give two definitions of periodic words and primitive words, respectively, and show some connections to coding theory.

Definition 1 *A word* $u \in \Sigma^+$ *is said to be* **periodic** *if there exists a word* $v \in \Sigma^*$ *and a natural number* $n \geq 2$ *such that* $u = v^n$*. If* $u \in \Sigma^+$ *is not periodic, then it is called a* **primitive word over** Σ*.*

Obviously, this definition is equivalent to the following.

Definition 1′ *A word* $u \in \Sigma^+$ *is said to be* **primitive** *if it is not a power of another word, that is,* $u = v^n$ *with* $v \in \Sigma^*$ *implies* $n = 1$ *and* $v = u$*. If* $u \in \Sigma^+$ *is not primitive, then it is called a* **periodic word over** Σ*.*

Definition 2 *The set of all periodic words over* Σ *is denoted by* $\mathsf{Per}(\Sigma)$*, the set of all primitive words over* Σ *is denoted by* $Q(\Sigma)$*.*

Obviously, $Q(\Sigma) = \Sigma^+ \setminus \mathsf{Per}(\Sigma)$.

In the sequel, if Σ is understood, and for simplicity, instead of $\mathsf{Per}(\Sigma)$ and $Q(\Sigma)$ we will write Per and Q, respectively.

Now we cite some fundamental definitions from coding theory.

Definition 3 *A nonempty set* $\mathcal{C} \subseteq \Sigma^*$ *is called a* **code** *if every equation* $u_1u_2\cdots u_m = v_1v_2\cdots v_n$ *with* $u_i, v_j \in \mathcal{C}$ *for all* i *and* j *implies* $n = m$ *and* $u_i = v_i$ *for all i.*
A nonempty set $\mathcal{C} \subseteq \Sigma^*$ *is called an* n**-code** *for* $n \in \mathbb{N}$*, if every nonempty subset of* $\mathcal{C}$ *with at most* n *elements is a code. A nonempty set* $\mathcal{C} \subseteq \Sigma^+$ *is called an* **intercode** *if there is some* $m \geq 1$ *such that* $\mathcal{C}^{m+1} \cap \Sigma^+\mathcal{C}^m\Sigma^+ = \emptyset$*.*

Connections to primitive words are stated by the following theorems.

Theorem 4 *If $\mathcal{C} \subseteq \Sigma^+$ and for all $p, q \in \mathcal{C}$ with $p \neq q$ holds that $pq \in Q$, then $\mathcal{C}$ is a 2-code.*

The proof will be given in Section 2.

Theorem 5 *If $\mathcal{C}$ is an intercode, then $\mathcal{C} \subseteq Q$.*

Proof. Assume that $\mathcal{C} \not\subseteq Q$ is an intercode and $\mathcal{C}^{m+1} \cap \Sigma^+\mathcal{C}^m\Sigma^+ = \emptyset$ for some $m \geq 1$. Then we have a periodic word u in $\mathcal{C}$ which means $u = v^n \in \mathcal{C}$ for some $v \in \Sigma^+$ and $n \geq 2$. Then $u^{m+1} = v^{n(m+1)} = v(v^{nm})v^{n-1} \in \mathcal{C}^{m+1} \cap \Sigma^+\mathcal{C}^m\Sigma^+$, which is a contradiction. $\square$

1.3 Roots of words and languages

Every nonempty word $p \in \Sigma^+$ is either the power of a shorter word q (if it is periodic) or it is not a power of another word (if it is primitive). The shortest word q with this property (in the first case) resp. p itself (in the second case) is called the root of p.

Definition 6 *The* **root of a word** *$p \in \Sigma^+$ is the unique primitive word q such that $p = q^n$ for some also unique natural number n. It is denoted by $\sqrt{p}$ or $\mathrm{root}(p)$. The number n in this equation is called the* **degree of** *p, denoted by $\deg(p)$. For a language L, $\sqrt{L} =_{\mathrm{Df}} \{\sqrt{p} : p \in L \wedge p \neq \epsilon\}$ is the* **root of** *L, $\deg(L) =_{\mathrm{Df}} \{\deg(p) : p \in L \wedge p \neq \epsilon\}$ is the* **degree of** *L.*

Remark. The uniqueness of root and degree is obvious, a formal proof will be given in Section 2.

Corollary 7 *$p = \sqrt{p}^{\deg(p)}$ for each word $p \neq \epsilon$; $\sqrt{L} \subseteq Q$ for each language L; $\sqrt{\Sigma^*} = Q$; $\sqrt{L} = L$ if and only if $L \subseteq Q$.*

2 Primitivity and combinatorics on words

Combinatorics on words is a fundamental part of the theory of words and languages. It is profoundly connected to numerous different fields of mathematics and its applications and it emphasizes the algorithmic nature of problems on words. Its objects are elements from a finitely generated free monoid and therefore combinatorics on words is a part of noncommutative discrete mathematics. For its comprehensive results and its influence to coding theory and

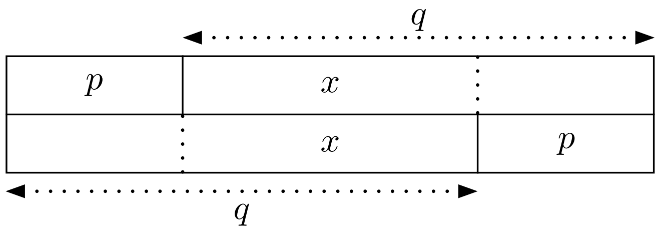

Figure 1: To the proof of Theorem 8

primitive words we refer to the textbooks of Yu [28], Shyr [24], Lothaire [19], and to Chapter 6 in [23]. Here we summarize some results from this theory which are important for studying primitive words or which will be used later.

The following theorem was first proved for elements of a free monoid.

Theorem 8 (Lyndon and Schützenberger [20]). *If* $pq = qp$ *for nonempty words* p *and* q*, then* p *and* q *are powers of a common word and therefore* pq *is not primitive.*

Proof. We prove the theorem by induction on the length of pq, which is at least 2. For $|pq| = 2$ and $pq = qp$, $p, q \neq \epsilon$, we must have $p = q = a$ for some $a \in \Sigma$, and the conclusion is true. Now suppose the theorem is true for all pq with $|pq| \leq n$ for a fixed $n \geq 2$. Let $|pq| = n + 1$, $pq = qp$, $p, q \neq \epsilon$, and, without loss of generality, $|p| \leq |q|$. We have a situation as in Figure 1. There must exist $x \in \Sigma^*$ such that $q = px = xp$.

Case 1) $x = \epsilon$. Then $p = q$, and the conclusion is true.

Case 2) $x \neq \epsilon$. Since $|px| \leq n$, by induction hypothesis p and x are powers of a common word. Then also q is a power of this common word.

The theorem follows from induction. □

Corollary 9 $w \notin Q$ *if and only if there exist* $p, q \in \Sigma^+$ *such that* $w = pq = qp$*.*

Theorem 10 (Shyr and Thierrin [25]) *For words* $p, q \in \Sigma^*$*, the two-element set* $\{p, q\}$ *is a code if and only if* $pq \neq qp$*.*

Proof. First note, that both statements in the theorem imply, that $p, q \neq \epsilon$ and $p \neq q$. It is trivial that for a code $\{p, q\}$, $pq \neq qp$ must hold. Now we show, that no set $\{p, q\}$ with $pq \neq qp$ can exist which is not a code. Assume the opposite. Then

$\mathcal{M} =_{\mathrm{Df}} \{\{p,q\} :\ p,q \in \Sigma^* \wedge pq \neq qp \wedge \{p,q\} \text{ is not a code}\} \neq \emptyset$.
Let $\{p,q\} \in \mathcal{M}$ where $|pq|$ is minimal, and let w be a word with minimal length having two different representations over $\{p,q\}$. Then $|w| > 2$ and one of the following must be true:
either (a) $w = pup = qu'q$ or (b) $w = pvq = qv'p$ for some $u, u', v, v' \in \{p,q\}^*$. Because of $p \neq q$, $p \sqsubset q$ or $q \sqsubset p$ must follow. Let us assume that $p \sqsubset q$. For the case $q \sqsubset p$ the proof can be carried out symmetrically. Then from both (a) and (b) it follows that $q = pr = sp$ for some $r, s \in \Sigma^+$. We have $|r| = |s| \neq |p|$ (because otherwise $r = s = p$ and $q = pp$), $|pr| < |pq|$, and $pr \neq rp$ (because otherwise $r = s$ and $pq = psp = prp = qp$). With $q = pr$ follows either (a') $pup = pru'pr$ from (a), or (b') $pvpr = prv'p$ from (b). Because of $|pr| < |pq|$, the choice of $\{p,q\}$ having minimal length, and the definition of $\mathcal{M}$, it must follow that $\{p,r\}$ is a code. But then from both (a') and (b') follows $p = r$, which is a contradiction. Hence $\mathcal{M}$ must be empty. □

From the last two theorems we get the following corollary which for its part proves Theorem 4.

Corollary 11 *If* $pq \in Q$ *for words* $p, q \neq \epsilon$*, then* $\{p,q\}$ *is a code.*

Note, that the reversal of this corollary is not true. For example, $\{aba, b\}$ is a code, but $abab \notin Q$.

A weaker variant of the next theorem has been proved also by Lyndon and Schützenberger [20] for elements of a free monoid. Our proof follows that presented by Lothaire [19].

Theorem 12 (Fine and Wilf [7]) *Let* p *and* q *be nonempty words,* $|p| = n$*,* $|q| = m$*, and* $d = gcd(n,m)$ *be the greatest common divisor of* n *and* m*. If* p^i *and* q^j *for some* $i, j \in \mathbb{N}$ *have a common prefix* u *of length* $n + m - d$*, then* p *and* q *are powers of a common word of length* d *and therefore* $\sqrt{p} = \sqrt{q}$.

Proof. Assume that the premises of the theorem are fulfilled and, without loss of generality, $1 \leq n \leq m - 1$ (otherwise $n = m = d$ and $p = q = u$). We first assume $d = 1$ and show, that p and q are powers of a common letter.
Because of $u \sqsubseteq p^i$ and $|u| = m - 1 + n$ we have
(1) $u[x] = u[x+n]$ for $1 \leq x \leq m - 1$.
Because of $u \sqsubseteq q^j$ we have
(2) $u[y] = u[y+m]$ for $1 \leq y \leq n - 1$.
Because of (1) and $1 \leq m - n \leq m - 1$ we have
(3) $u[m] = u[m-n]$.

Let now $1 \leq x \leq y \leq m-1$ with $y - x \equiv n \mod m$. Then we have two cases.

Case a). $y = x + n \leq m-1$, and therefore $u[x] = u[y]$ by (1).

Case b). $y = x + n - m$. Since $x \leq m-1$ we have $x + n - m \leq n-1$ and $u[y] = u[x+n-m] = u[x+n] = u[x]$ by (2) and (1).

Hence $u[x] = u[y]$ whenever $1 \leq x \leq y \leq m-1$ and $y - x \equiv n \mod m$. It follows by (1) that $u[x] = u[y]$ whenever $1 \leq x \leq y \leq m-1$ and $y - x \equiv k \cdot n \mod m$ for some $k \in \mathbb{N}$. Because of $\gcd(n,m) = 1$, the latter is true if $y - x$ is any value of $\{1, 2, \ldots, m-1\}$. This means, under inclusion of (3), $u[1] = u[2] = \cdots = u[m]$, and p and q are powers of the letter $u[1]$.

If $d > 1$, we argue in exactly the same way assuming Σ^d instead of Σ as the alphabet. □

If we assume, $p^i = q^j$ for primitive words p and q and $i, j \in \mathbb{N} \setminus \{0\}$, then by Theorem 12, p and q are powers of a common word which can only be $p = q$ itself because of its primitivity. This means the uniqueness of the root of a word which also implies the uniqueness of its degree.

Using Theorem 12 we can easily prove the next theorem.

Theorem 13 (Borwein) *If $w \notin Q$ and $wa \notin Q$, where $w \in \Sigma^+$ and $a \in \Sigma$, then $w \in a^+$.*

The next theorem belongs to the most frequently referred properties concerning primitive words.

Theorem 14 (Shyr and Thierrin [26]) *If $u_1u_2 \neq \epsilon$ and $u_1u_2 = p^i$ for some $p \in Q$, then $u_2u_1 = q^i$ for some $q \in Q$. This means, if $u = u_1u_2 \neq \epsilon$ and $u' = u_2u_1$, then $\deg(u) = \deg(u')$, $|\sqrt{u}| = |\sqrt{u'}|$, and therefore u primitive if and only if u' primitive.*

Proof. Let $u_1u_2 = p^i \neq \epsilon$ and $p \in Q$. We consider two cases.

Case 1). $i = 1$, which means, u_1u_2 is primitive. Assume that u_2u_1 is not primitive and therefore $u_2u_1 = q^j$ for some $q \in Q$ and $j \geq 2$. Then $q = q_1q_2 \neq \epsilon$ such that $u_2 = (q_1q_2)^n q_1$, $u_1 = q_2(q_1q_2)^m$, and $j = n + m + 1$. It follows that $u_1u_2 = (q_2q_1)^{m+n+1} = (q_2q_1)^j$ is not primitive. By this contradiction, $u_2u_1 = q^1$ is primitive.

Case 2). $i \geq 2$. Then $p = p_1p_2 \neq \epsilon$ such that $u_1 = (p_1p_2)^n p_1$, $u_2 = p_2(p_1p_2)^m$, and $i = n + m + 1$. Since $p = p_1p_2$ is primitive, by Case 1 also $q =_{\mathrm{Df}} p_2p_1$ is primitive, and $u_2u_1 = (p_2p_1)^{m+n+1} = q^i$. □

The proof of the following theorem, which was first done by Lyndon and Schützenberger [20] for a free group, is rather difficult and therefore omitted here.

Theorem 15 *If $u^m v^n = w^k \neq \epsilon$ for words $u, v, w \in \Sigma^*$ and natural numbers $m, n, k \geq 2$, then u,v and w are powers of a common word.*
We say, that the equation $u^m v^n = w^k$, where $m, n, k \geq 2$ has only trivial solutions.

The next two theorems are consequences of Theorem 15.

Theorem 16 *If $p, q \in Q$ with $p \neq q$, then $p^i q^j \in Q$ for all $i, j \geq 2$.*

This theorem is not true if $i = 1$ or $j = 1$. For instance, let $p = aba$, $q = baab$, $i = 2$, $j = 1$.

Theorem 17 *If $p, q \in Q$ with $p \neq q$ and $i \geq 1$, then there are at most two periodic words in each of the languages $p^i q^*$ and $p^* q^i$.*

Proof. Assume that there are periodic words in $p^i q^*$, and $p^i q^j$ should be the smallest of them. Then $p^i q^j = r^k$ for some $r \in Q$, $k \geq 2$, $r \neq q$. Let also $p^i q^l = s^m \in Per$, $s \in Q$, $l > j$, $m \geq 2$. Then $s^m = r^k q^{l-j}$, and $l - j = 1$ by Theorem 15. Therefore at most two words $p^i q^j$ and $p^i q^{j+1}$ in $p^i q^*$ can be periodic. For $p^* q^i$ the proof is done analogously. □

With essentially more effort, the following can be shown.

Theorem 18 (Shyr and Yu [27, 28]) *If $p, q \in Q$ with $p \neq q$, then there is at most one periodic word in the language $p^+ q^+$.*

3 Primitivity and language classes

As soon as the set Q of primitive words (over a fixed alphabet Σ) was defined, the question arose which is the exact relationship between Q and several known language classes. Here it is important that Σ is a nontrivial alphabet because in the other case all results become trivial or meaningless: If $\Sigma = \{a\}$ then $Q(\Sigma) = \Sigma = \{a\}$ and $Per(\Sigma) = \{a^n : n \geq 2\}$.

First we will examine the relationship of Q to the classes of the Chomsky hierarchy, and second that to the Marcus contextual languages.

3.1 Chomsky hierarchy

Let us denote by REG, CF and CS the class of all regular languages, the class of all context-free languages and the class of all context-sensitive languages (all over the nontrivial alphabet Σ), respectively. It is known from Chomsky Hierarchy that REG $\subset$ CF $\subset$ CS (see, e.g., the textbooks [8, 9, 23]). It is easy to show that $Q \in \mathrm{CS} \setminus \mathrm{REG}$, and hence it remains the question whether Q is context-free. Before stating the theorem let us remember that CF is the class of languages which are acceptable by nondeterministic pushdown automata, and CS is the class of languages which are acceptable by nondeterministic linear bounded automata. The latter are Turing machines where the used space on its tapes (this is the number of tape cells touched by the head) is bounded by a constant multiple of the length of the input string. If the accepting automaton is a deterministic one the corresponding language is called a deterministic context-free or a deterministic context-sensitive language, respectively. It can be shown that the deterministic context-free languages are a strict subclass of the context-free languages, whereas it is not yet known whether this inclusion is also strict in the case of context-sensitive languages (This is the famous LBA-problem).

Theorem 19 Q *is deterministic context-sensitive but not regular.*

Proof. 1. It is easy to see that by a deterministic Turing machine for a given word u can be checked whether it fulfills Definition 1 and thus whether it is not primitive or primitive, and this can be done in space which is a constant multiple of $|u|$.

2. is a corollary from the next theorem.

Theorem 20 *A language containing only a bounded number of primitive words and having an infinite root cannot be regular.*

If Q would be regular, then also $\overline{Q} = \mathsf{Per} \cup \{\epsilon\}$ would be regular because the class of regular languages is closed under complementation. But $\sqrt{\overline{Q}} = Q$ is infinite and therefore by Theorem 20 it cannot be regular. □

Proof of Theorem 20. Let L be a language with an infinite root and a bounded number of primitive words. Further let
$m =_{\mathrm{Df}} \max(\{|p| : p \in L \cap Q\} \cup \{0\})$. Assume that L is regular. By the pumping lemma for regular languages, there exists a natural number $n \geq 1$, such that any word $u \in L$ with $|u| \geq n$ has the form $u = xyz$ such that $|xy| \leq n$, $y \neq \epsilon$, and $xy^k z \in L$ for all $k \in \mathbb{N}$. Let now $u \in L$ with $|\sqrt{u}| > n$ and $|u| > m$. Then $u = xyz$ such that $1 \leq |y| \leq |xy| \leq n$, $z \neq \epsilon$, and $xy^k z \in L$ for all $k \in \mathbb{N}$. By Theorem 14, for each $k \geq 1$, zxy^k is periodic (since $|xy^k z| \geq |u| > m$). Let $p =_{\mathrm{Df}} \sqrt{zx}$, $i =_{\mathrm{Df}} \mathsf{deg}(zx)$, and $q =_{\mathrm{Df}} \sqrt{y}$. It is $p \neq q$ because otherwise, by Theorem 14, $|\sqrt{u}| = |\sqrt{zxy}| = |\sqrt{y}| \leq |y| \leq n$ contradicting the assumption $|\sqrt{u}| > n$. Then we have infinitely many periodic words in $p^i q^*$ contradicting Theorem 17. □

In 1991 it was conjectured by Dömösi, Horváth and Ito [4] that Q is not context-free. Even though up to now all attempts to prove or disprove this conjecture failed, it is mostly assumed to be true. Some approximations to the solution of this problem will be given with the following theorems.

Theorem 21 Q *is not deterministic context-free.*

Proof. We use the fact that the class of deterministic context-free languages is closed under complementation and under intersection with regular sets. Assume that Q is deterministic context-free. Then also $\overline{\mathsf{Q}} \cap \mathsf{a}^*\mathsf{b}^*\mathsf{a}^*\mathsf{b}^* = \{\mathsf{a}^i\mathsf{b}^j\mathsf{a}^i\mathsf{b}^j : i, j \in \mathbb{N}\}$ must be deterministic context-free. But using the pumping lemma for context-free languages, it can be shown that the latter is not even context-free. $\square$

In the same way (using the pumping lemma for $\mathsf{Per} \cap \mathsf{a}^*\mathsf{b}^*\mathsf{a}^*\mathsf{b}^*$) it also follows that Per is not context-free.

The next theorem has a rather difficult proof. Therefore and because we will not explain what unambiguity means, we omit the proof.

Theorem 22 (Petersen [22]) Q *is not an unambigous context-free language.*

Another interesting language class which is strictly between the context-free and the regular languages is the class LIN of all linear languages.

Definition 23 *A grammar* $G = [N, T, P, S]$ *is* **linear** *if its productions are of the form* $A \to aB$ *or* $A \to Ba$ *or* $A \to a$*, where* $a \in T$ *and* $A, B \in N$*. A production of the form* $S \to \epsilon$ *can also be accepted if the start symbol* S *does not occur in the right-hand side of any production.*

A **linear language** *is a language which can be generated by a linear grammar. LIN is the class of all linear languages.*

It can be shown that REG $\subset$ LIN $\subset$ CF.

Theorem 24 (Horváth [10]) Q *is not a linear language.*

The proof can be done by using a special pumping lemma for linear languages and will be omitted here.

Let $\mathcal{L}$ be the union of the classes of linear languages, unambigous context-free languages and deterministic context-free languages. Then $\mathcal{L} \subset$ CF and, by the former theorems, $\mathsf{Q} \notin \mathcal{L}$. But, whether $\mathsf{Q} \in$ CF or not, is still unknown.

3.2 Contextual languages

Though we do not know the exact position of Q in the Chomsky Hierarchy, its position in the system of contextual languages is clear. First, we cite the basic definitions from [21], see also [15], and then, after three examples we prove our result.

Definition 25 *A* **(Marcus) contextual grammar** *is a structure* $G = [\Sigma, A, C, \phi]$ *where* Σ *is an alphabet,* A *is a finite subset of* Σ^* *(called the set of axioms),* C *is a finite subset of* $\Sigma^* \times \Sigma^*$ *(called the set of contexts), and* ϕ *is a function from* Σ^* *into* $\mathcal{P}(C)$ *(called the choice function). If* $\phi(u) = C$ *for every* $u \in \Sigma^*$ *then* G *is called a* **(Marcus) contextual grammar without choice.**

With such a grammar the following relations on Σ^* are associated: For $w, w' \in \Sigma^*$,
(1) $w \Rightarrow_{ex} w'$ if and only if there exists $[p_1, p_2] \in \phi(w)$ such that $w' = p_1 w p_2$,
(2) $w \Rightarrow_{in} w'$ if and only if there exists $w_1, w_2, w_3 \in \Sigma^*$ and $[p_1, p_2] \in \phi(w_2)$ such that $w = w_1 w_2 w_3$ and $w' = w_1 p_1 w_2 p_2 w_3$.

$\Rightarrow^*_{ex}$ and $\Rightarrow^*_{in}$ denote the reflexive and transitive closure of these two relations.

Definition 26 *For a contextual grammar* $G = [\Sigma, A, C, \phi]$ *(with or without choice),*
$\mathcal{L}_{ex}(G) =_{Df} \{w : \exists u(u \in A \wedge u \Rightarrow^*_{ex} w)\}$ *is the* **external contextual language (with or without choice) generated by** G*,*
and $\mathcal{L}_{in}(G) =_{Df} \{w : \exists u(u \in A \wedge u \Rightarrow^*_{in} w)\}$ *is the* **internal contextual language (with or without choice) generated by** G.

For every contextual grammar $G = [\Sigma, A, C, \phi]$, $A \subseteq \mathcal{L}_{ex}(G) \subseteq \mathcal{L}_{in}(G)$ holds.

The above definitions are illustrated by the following examples.

Example 1 Let $G = [\Sigma, A, C, \phi]$ be a contextual grammar where $\Sigma = \{a, b\}$, $A = \{\epsilon, ab\}$, $C = \{[\epsilon, \epsilon], [a, b]\}$, $\phi(\epsilon) = \{[\epsilon, \epsilon]\}$, $\phi(ab) = \{[a, b]\}$ and $\phi(w) = \emptyset$ if $w \notin A$. Then $\mathcal{L}_{ex}(G) = \{\epsilon, ab, aabb\}$ and
$\mathcal{L}_{in}(G) = \{a^n b^n : n \in \mathbb{N}\}$ since $ab \Rightarrow_{ex} aabb$, $ab \Rightarrow^*_{in} a^n b^n$ for every $n \geq 1$, and there does not exist any w' such that $aabb \Rightarrow_{ex} w'$.

Example 2 Let $G = [\Sigma, A, C, \phi]$ be a contextual grammar where $\Sigma = \{a, b\}$, $A = \{a\}$, $C = \{[\epsilon, \epsilon], [\epsilon, a], [\epsilon, b]\}$,

$\phi(\epsilon) = \{[\epsilon, \epsilon]\}$, $\phi(ua) = \{[\epsilon, b]\}$ for $u \in \Sigma^*$ and $\phi(ub) = \{[\epsilon, a]\}$ for $u \in \Sigma^*$. Then $\mathcal{L}_{ex}(G) = \{a, ab, aba, abab, \ldots\} = a(ba)^* \cup a(ba)^*b$ and $\mathcal{L}_{in}(G) = a\Sigma^* \setminus aa\Sigma^*$.

Example 3 Let $u = a_1a_2a_3\cdots$ be an ω-word over a nontrivial alphabet Σ where $a_i \in \Sigma$ for all $i \geq 1$. Let $G = [\Sigma, A, C, \phi]$ be a contextual grammar where $A = \{\epsilon, a_1\}$, $C = \{[\epsilon, \epsilon]\} \cup \{[\epsilon, a] : a \in \Sigma\}$, $\phi(\epsilon) = \{[\epsilon, \epsilon]\}$, $\phi(a_1a_2\cdots a_i) = \{[\epsilon, a_{i+1}]\}$ and $\phi(w) = \emptyset$ if w is not a prefix of u. Then $\mathcal{L}_{ex}(G) = \{\epsilon, a_1, a_1a_2, a_1a_2a_3, \ldots\} = \mathsf{Pr}(u)$ is the set of all prefixes of u. Hence, there exist contextual grammars generating languages which are not recursively enumerable.

Theorem 27 (Ito [5]) Q *is an external contextual language with choice but not an external contextual language without choice or an internal contextual language with or without choice.*

Proof. 1. Let $G = [\Sigma, \Sigma, \{[u,v] : uv \in \Sigma^* \wedge |uv| \leq 2\}, \phi]$ be a contextual grammar, where $\phi(w) = \{[u,v] : uv \in \Sigma^* \wedge |uv| \leq 2 \wedge uwv \in Q\}$ for every $w \in \Sigma^*$. Then obviously $\mathcal{L}_{ex}(G) \subseteq Q$. We prove $Q \subseteq \mathcal{L}_{ex}(G)$ by induction. First we have $\Sigma \subseteq (\Sigma \cup \Sigma^2) \cap Q \subseteq \mathcal{L}_{ex}(G)$. Now assume that for a fixed $n \geq 2$ all primitive words p with $|p| \leq n$ are in $\mathcal{L}_{ex}(G)$. Let u be a primitive word of smallest length $\geq n+1$. We have two cases.

Case a). $u = wx_1x_2$ with $x_1, x_2 \in \Sigma$ and at least one of w and wx_1 is in Q. Then, by induction hypothesis, $w \in \mathcal{L}_{ex}(G)$ or $wx_1 \in \mathcal{L}_{ex}(G)$. But then $w \Rightarrow_{ex} wx_1x_2$ or $wx_1 \Rightarrow_{ex} wx_1x_2$, and thus $u \in \mathcal{L}_{ex}(G)$.

Case b). $u = wx_1x_2$ with $x_1, x_2 \in \Sigma$ and none of w and wx_1 is in Q. Then, by Theorem 13, $w = x_1^i$ for some $i \geq 1$, hence $u = x_1^{i+1}x_2$ with $x_1 \neq x_2$, and $x_2 \Rightarrow_{ex} x_1x_2 \Rightarrow_{ex} x_1x_1x_2 \Rightarrow_{ex} \cdots \Rightarrow_{ex} x_1^{i+1}x_2$, and therefore $u \in \mathcal{L}_{ex}(G)$.

2. Assume that there exists a contextual grammar $G = [\Sigma, A, C, \phi]$ without choice such that $Q = \mathcal{L}_{ex}(G)$. There must be at least one pair $[u,v] \in \phi(w)$ with $uv \neq \epsilon$ for all $w \in \Sigma^*$. Let $p = \sqrt{vu}$ and $i = \mathsf{deg}(vu) \geq 1$. Because of $p \in Q = \mathcal{L}_{ex}(G)$, also upv would be in $\mathcal{L}_{ex}(G)$. We have $vup = p^{i+1}$. By Theorem 14, $\mathsf{deg}(upv) = \mathsf{deg}(vup) = i+1 \geq 2$ and therefore $upv \notin Q$, which is a contradiction.

3. Assume $Q = \mathcal{L}_{in}(G)$ for some contextual grammar $G = [\Sigma, A, C, \phi]$ (with or without choice). There must be words $u, v, w \in \Sigma^*$ with $uv \neq \epsilon$ and $[u,v] \in \phi(w)$. Let $n = |uwv|$ and $a, b \in \Sigma$ with $a \neq b$. Then $a^nb^nwa^nb^nuwv \in Q$, but $a^nb^nwa^nb^nuwv \Rightarrow_{in} a^nb^nuwva^nb^nuwv = (a^nb^nuwv)^2 \notin Q$, contradicting $\mathcal{L}_{in}(G) = Q$. $\square$

Theorem 28 Per *is not a contextual language of any kind.*

Proof. Assume $\mathtt{Per} = \mathcal{L}_{ex}(G)$ or $\mathtt{Per} = \mathcal{L}_{in}(G)$ for some contextual grammar $G = [\Sigma, A, C, \phi]$ (with or without choice). Let m be a fixed number with $m > \max\{|p| : p \in A \vee \exists u([p,u] \in C \vee [u,p] \in C)\}$. Because $a^m b^m a^m b^m \in \mathtt{Per}$ we must have $q \in \mathtt{Per}$ such that $q \Rightarrow_{ex} a^m b^m a^m b^m$ or $q \Rightarrow_{in} a^m b^m a^m b^m$. In the first case, $q = a^i b^m a^m b^j$ with $i < m \vee j < m$ must follow. But then $q \notin \mathtt{Per}$. In the second case, $q = a^i b^j a^k b^l$ with $i < m \vee j < m \vee k < m \vee l < m$ must follow. But then $qq \Rightarrow_{in} a^m b^m a^m b^m a^i b^j a^k b^l \notin \mathtt{Per}$ whereas $qq \in \mathtt{Per}$. Therefore $\mathtt{Per} \neq \mathcal{L}_{ex}(G)$ and $\mathtt{Per} \neq \mathcal{L}_{in}(G)$. □

4 Primitivity and complexity

To investigate the computational complexity of Q and that of roots of languages on the one hand is interesting for itself, on the other hand - because $Q = \sqrt{\Sigma^*}$ - there was some speculation to get hints for solving the problem of context-freeness of Q. First, let us repeat some basic notions from complexity theory.

If $\mathcal{M}$ is a deterministic Turing machine, then $t_{\mathcal{M}}$ is the **time complexity** of $\mathcal{M}$, defined as follows. If $p \in \Sigma^*$, where Σ is the input alphabet of $\mathcal{M}$, and $\mathcal{M}$ on input p reaches a final state (we also say $\mathcal{M}$ **halts on** p), then $t_{\mathcal{M}}(p)$ is the number of computation steps required by $\mathcal{M}$ to halt. If $\mathcal{M}$ does not halt on p, then $t_{\mathcal{M}}(p)$ is undefined. For natural numbers n, $t_{\mathcal{M}}(n) =_{Df} \max\{t_{\mathcal{M}}(p) : p \in \Sigma^* \wedge |p| = n\}$ if $\mathcal{M}$ halts on each word of length n. If t is a function over the natural numbers, then TIME(t) denotes the class of all sets which are accepted by multitape deterministic Turing machines whose time complexity is bounded from above by t. Restricting to one-tape machines, the time complexity class is denoted by 1-TIME(t).

For simplicity, let us write TIME(n^2) instead of the more exact notation TIME(f), where $f(n) = n^2$.

Theorem 29 (Horváth and Kudlek [12]) $Q \in$ *1-TIME*(n^2).

The proof which will be omitted is based on Corollary 9 and the linear speed-up of time complexity. The latter means that 1-TIME$(t') \subseteq$ 1-TIME(t) if $t' \in O(t)$ and $t(n) \geq n^2$ for all n.

The time bound n^2 is optimal for accepting Q (or $\mathtt{Per}$) by one-tape Turing machines, which is shown by the next theorem.

Theorem 30 ([17]) *For each one-tape Turing machine $\mathcal{M}$ deciding Q, $t_{\mathcal{M}} \in \Omega(n^2)$ must hold. The latter means:*
$\exists c \exists n_0 (c > 0 \wedge n_0 \in \mathbb{N} \wedge \forall n(n \geq n_0 \rightarrow t_{\mathcal{M}}(n) \geq c \cdot n^2))$.

The proof which will be omitted also, uses the for complexity theorists well-known method of counting the crossing sequences.

Now we turn to the relationship between the complexity of a language and that of its root. It turns out that there is no general relation, even more, there can be an arbitrary large gap between the complexity of a language and that of its root.

Theorem 31 ([17, 16]) *Let* t *and* f *be arbitrary total functions over* $\mathbb{N}$ *such that* $t \in \omega(n)$ *is monotone nondecreasing and* f *is monotone nondecreasing, unbounded, and time constructible. Then there exists a language* L *such that* $L \in$ *1-TIME*$(O(t))$ *but* $\sqrt{L} \notin$ *TIME*(f).

Instead of the proof which is a little bit complicated we only explain the notions occuring in the theorem. $t \in \omega(n)$ means $\lim\limits_{n \to \infty} \frac{n}{t(n)} = 0$. A time constructible function is a function f for which there is a Turing machine halting in exactly $f(n)$ steps on every input of length n for each $n \in \mathbb{N}$. One can show that the most common functions have these properties. Finally, 1-TIME$(O(t)) = \bigcup\{$1-TIME$(t') : \exists c \exists n_0 (c > 0 \land n_0 \in \mathbb{N} \land \forall n (n \geq n_0 \to t'(n) \leq c \cdot t(n)))\}$.

Let us still remark, that from Theorem 31 we can deduce that there exist regular languages the roots of which are not even context-sensitive, see [15, 16].

5 Powers of languages

In arithmetics powers in some sense are counterparts to roots. Also for formal languages we can define powers, and also here we shall establish some connections to roots. For the first time, the power $pow(L)$ of a language L was defined by Calbrix and Nivat in [3] in connection with the study of properties of period and prefix languages of ω-languages. They also raised the problem to characterize those regular languages whose powers are also regular, and to decide the problem whether a given regular language has this property. Cachat [2] gave a partial solution to this problem showing that for a regular language L over a one-letter alphabet, it is decidable whether $pow(L)$ is regular. Also he suggested to consider as the set of exponents not only the whole set $\mathbb{N}$ of natural numbers but also an arbitrary regular set of natural numbers. This suggestion was taken up in [13] with the next definition.

Definition 32 *For a language* $L \subseteq \Sigma^*$ *and a natural number* $k \in \mathbb{N}$,
$L^{(k)} =_{Df} \{p^k : p \in L\}$. *For* $H \subseteq \mathbb{N}$,

$\mathsf{pow}_H(L) =_{Df} \bigcup_{k\in H} L^{(k)} = \{p^k : p \in L \wedge k \in H\}$ *is the* H**-power** *of* L.
Instead of $\mathsf{pow}_H(L)$ *we also write* $L^{(H)}$*, and also it is usual to write* $\mathsf{pow}(L)$ *instead of* $\mathsf{pow}_{\mathbb{N}}(L) = L^{(\mathbb{N})}$.

Note the difference between $L^{(k)}$ and L^k. For instance, if $L = \{a, b\}$ then $L^{(2)} = \{aa, bb\}$, $L^2 = \{aa, ab, ba, bb\}$ and $L^{(\mathbb{N})} = a^* \cup b^*$.

We say that a set H of natural numbers has some language theoretical property if the corresponding one-symbol language $\{a^k : k \in H\} = \{a\}^{(H)}$ which is isomorphic to H has this property.

It is easy to see that every regular power of a regular language is context-sensitive. More generally, we have the following theorem.

Theorem 33 ([13]) *If* $H \subseteq \mathbb{N}$ *is context-sensitive and* $L \in$ *CS then also* $\mathsf{pow}_H(L) = L^{(H)}$ *is context-sensitive.*

Proof. Let $L \subseteq \Sigma^*$ be context-sensitive and also $H \subseteq \mathbb{N}$ be context-sensitive. By the following algorithm, for a given word $u \in \Sigma^*$ we can decide whether $u \in L^{(H)}$.

```
1   if $(u \in L \wedge 1 \in H) \vee (u = \epsilon \wedge 0 \in H)$
2     then return "$u$ is in $L^{(H)}$"
3     else compute $p = \sqrt{u}$ and $d = \mathsf{deg}(u)$
4          for $i \leftarrow 1$ to $\lfloor \frac{d}{2} \rfloor$
5              do if $p^i \in L \wedge \frac{d}{i} \in H$
6                    then return "$u$ is in $L^{(H)}$"
7          return "$u$ is not in $L^{(H)}$"
```

$\lfloor \frac{d}{2} \rfloor$ in line 4 is $\frac{d}{2}$ if d is even, and $\frac{d-1}{2}$ if d is odd. Each step of the algorithm can be done by a linear bounded automaton or by a Turing machine where the used space is bounded by a constant multiple of $|u|$. Crucial for this are that $|p| \leq |u|$, $d \leq |u|$, and the decisions in line 1 and in line 5 can also be done by a linear bounded automaton with this boundary, because L and H are context-sensitive and therefore acceptable by linear bounded automata. □

The last theorem raises the question whether and when $L^{(H)}$ is in a smaller class of the Chomsky hierarchy, especially if L is regular. This essentially depends on whether the root of L is finite or not. Therefore we will introduce the notions FR for the class of all regular languages L such that $\sqrt{L}$ is finite, and IR $=_{Df}$ REG $\setminus$ FR for the class of all regular languages L such that $\sqrt{L}$ is infinite.

Theorem 34 ([13]) *The class FR of regular sets having a finite root is closed under the power with finite sets.*

Proof. Let L be a regular language with a finite root $\{p_1, \dots, p_k\}$ and $\epsilon \notin L$, and let $L_i =_{Df} L \cap p_i^*$ for each $i \in \{1, \dots, k\}$. Since $L_i \subseteq p_i^*$ and $L_i \in$ REG, L_i is isomorphic to a regular set M_i of natural numbers, namely $M_i = deg(L_i)$. For each $n \in \mathbb{N}$, $M_i \cdot n =_{Df} \{m \cdot n : m \in M_i\}$ is regular too. Therefore, for a finite set $H \subseteq \mathbb{N}$, also $\bigcup\limits_{n \in H} M_i \cdot n$ is regular which is isomorphic to $L_i^{(H)}$. Then $L^{(H)} = \bigcup\limits_{i=1}^{k} L_i^{(H)}$ is regular, and $\sqrt{L^{(H)}} = \sqrt{L}$ is finite. If the empty word is in the language then, because of $pow_H(L \cup \{\epsilon\}) = pow_H(L) \cup \{\epsilon\}$ we get the same result. □

If H is infinite then $pow_H(L)$ may be nonregular and even non-context-free. This is true even in the case of a one-letter alphabet where the root of each nonempty set (except $\{\epsilon\}$) has exactly one element. This is illustrated by the following example.
Let $L = \{a^{2m+3} : m \in \mathbb{N}\}$. Then $L \in$ FR but
$L^{(\mathbb{N})} = \{a^k : k \in \mathbb{N} \setminus \{2^m : m \geq 0\}\} \notin$ CF.
Therefore it remains a problem to characterize those regular sets L with finite roots where $pow_H(L)$ is regular for any (maybe regular) set H.

Our next theorem shows that the powers of arbitrary (not necessarily regular) languages which have infinite roots are not regular, even more, they are not even context-free, if the exponent set is an arbitrary set of natural numbers containing at least one element which is greater than 2 and does not contain the number 1, or some other properties are fulfilled.

Theorem 35 ([13]) *For every language L which has an infinite root and for every set $H \subseteq \mathbb{N}$ containing at least one number greater than 2, $pow_H(L)$ is not context-free if one of the following conditions is true:*

(a) $1 \notin H$, *(c)* $L \cap \sqrt{L} \in$ *REG,*
(b) $\sqrt{L} \in$ *REG,* *(d)* $L \in$ *REG and* $\sqrt{L^{(H)} \setminus L}$ *is infinite.*

Proof. Let $L \subseteq \Sigma^*$ be a language such that $\sqrt{L}$ is infinite, and let $H \subseteq \mathbb{N}$ with $H \setminus \{0, 1, 2\} \neq \emptyset$. We define

$$L' =_{Df} \begin{cases} pow_H(L) & \text{if } (a) \text{ is true,} \\ pow_H(L) \setminus \sqrt{L} & \text{if } (b) \text{ is true,} \\ pow_H(L) \setminus (L \cap \sqrt{L}) & \text{if } (c) \text{ is true,} \\ pow_H(L) \setminus L & \text{if } (d) \text{ is true.} \end{cases}$$

If more than one of the conditions (a), (b), (c), (d) are true simultaneously, then it doesn't matter which of the appropriate lines in the definition of L' we choose. It is important that in each case, $\sqrt{L'}$ is infinite, there is no primitive word in L' and, if $pow_H(L)$ was context-free then also L' would be context-free. But we show that the latter is not true.
Assume that L' is context-free, and let $n \geq 3$ be a fixed number from H. By the pumping lemma for context-free languages, there exists a natural number m such that every $z \in L'$ with $|z| > m$ is of the form $w_1w_2w_3w_4w_5$ where: $w_2w_4 \neq \epsilon$, $|w_2w_3w_4| < m$, and $w_1w_2^iw_3w_4^iw_5 \in L'$ for all $i \in \mathbb{N}$.
Now let $z \in L'$ with $deg(z) \geq n$ and $|\sqrt{z}| > 2m$ which exists because $\sqrt{L'}$ is infinite. Let $p =_{Df} \sqrt{z}$ and $k =_{Df} deg(z)$. Then $|z| = k \cdot |p| > 2km$. By the pumping lemma, $z = p^k = w_1w_2w_3w_4w_5$ where $w_2w_4 \neq \epsilon$, $|w_2w_3w_4| < m < \frac{|p|}{2}$, and $w_1w_2^iw_3w_4^iw_5 \in L'$ for each $i \in \mathbb{N}$. Especially, for $i = 0$, $x =_{Df} w_1w_3w_5 \in L'$ and therefore x is nonprimitive. Now let $z' =_{Df} w_5w_1w_2w_3w_4$, $q =_{Df} \sqrt{z'}$, $x' =_{Df} w_5w_1w_3$, and $s =_{Df} \sqrt{x'}$. By Theorem 14 we have $deg(z') = deg(z) = k$ and x' nonprimitive, therefore $|q| = |p| > 2m$ and $|s| \leq \frac{|x'|}{2}$. It follows $z' = q^k$ and $x' = q^{k-1}q'$ for some word q' with $\frac{|q|}{2} < |q'| < |q|$ (because of $0 < |w_2w_4| \leq |w_2w_3w_4| < \frac{|q|}{2}$). The words z' and x' which are powers of q and s, respectively, have a common prefix w_5w_1 of length $|z| - |w_2w_3w_4| > k \cdot |q| - \frac{|q|}{2}$. Because of $|s| \leq \frac{|x'|}{2} < \frac{k}{2} \cdot |q|$ and $k \geq 3$, we have $|q| + |s| < (\frac{k}{2} + 1)|q| \leq (k - \frac{1}{2})|q|$, and therefore $q = s$ by Theorem 12. But then $x' = s^{k-1}q'$ with $0 < |q'| < |s|$ which contradicts $\sqrt{x'} = s$. □

It remains open whether the H-power of a regular language is regular or context-free or neither, if $H = \mathbb{N}$ or $H \subseteq \{0, 1, 2\}$. First, we consider the exceptions 0, 1, and 2 where we find out a different behavior.

Theorem 36 ([13]) *(i) For each* $L \in$ *REG and* $H \subseteq \{0, 1\}$, $L^{(H)} \in$ *REG.*
(ii) For each $L \in$ *FR,* $L^{(2)} \in$ *FR.*
(iii) For each $L \in$ *IR,* $L^{(2)} \notin$ *REG.*

Proof. (i) is trivial, (ii) follows from Theorem 34. (iii) follows from Theorem 20. □

A set $pow_{\{2\}}(L) = L^{(2)}$ we call also the **square** of L. Because of the former theorem, only the squares of regular languages with infinite roots remain for interest. In contrast to the former results where the power of a regular set either is regular again or not context-free, this is not true for the squares. It is illustrated by the following examples:

Let $L_1 =_{Df} a \cdot \{b\}^*$ and $L_2 =_{Df} \{a, b\}^*$. Then both L_1 and L_2 are regular with infinite roots, but $L_1^{(2)} \in$ CF and $L_2^{(2)} \notin$ CF.

To characterize those regular languages whose squares are context-free we introduce the following notion.

Definition 37 *Let $p \in Q$ and $w, w' \in \Sigma^*$ such that p is not a suffix of w and $w'w \notin p^+$. The sets wp^*w' and p^*wp^* are called* **inserted iterations of the primitive word** *p. The words p, w, w' are called the* **modules** *of wp^*w', and p, w are called the* **modules** *of p^*wp^*. A* **FIP-set** *is a finite union $L_1 \cup \ldots \cup L_n$ of inserted iterations of primitive words. The sets $L_1, \ldots, L_n$ are also called the* **components** *of the FIP-set.*

Using this notion we can give the following reformulation and simplification of a theorem by Ito and Katsura from 1991 (see [14]) which has a rather difficult proof.

Theorem 38 *If $L^{(2)} \in$ CF and $L^{(2)} \subseteq Q^{(2)}$ then L must be a subset of a FIP-set.*

Using this theorem and the proof idea from Theorem 35 we can show the following characterization.

Theorem 39 ([18]) *For a regular language L, $L^{(2)}$ is context-free if and only if L is a subset of a FIP-set.*

Proof. We show here only one direction. Let L be regular and $L^{(2)} \in$ CF. We consider three cases. Case a). $L \in$ FR. Let $\sqrt{L} = \{p_1, \ldots, p_n\}$. Then $L \subseteq p_1^* \cup \cdots \cup p_n^*$ and $p_1^* \cup \cdots \cup p_n^*$ is a FIP-set.

Case b). $L \in$ IR and $\sqrt{L \cap Per}$ is infinite. This means, L has infinitely many periodic words with altogether infinitely many roots of unbounded lengths. Then $L^{(2)}$ contains words z with $|\sqrt{z}| > 2m$ for arbitrary m and $\mathrm{deg}(z) \geq 4$. If $L^{(2)}$ would be context-free then we would get the same contradiction as in the proof of Theorem 35. Therefore case b) cannot occur.

Case c). $L \in$ IR and $\sqrt{L \cap Per}$ is finite. Let $L_1 =_{Df} L \cap Q$, $L_2 =_{Df} L \cap \overline{Q}$, and $\sqrt{L_2} = \{p_1, \ldots, p_k\}$. Then $L = L_1 \cup L_2$, $L_1 \cap L_2 = \emptyset$, and $L_2 = ((p_1^* \cup \cdots \cup p_k^*) \setminus \{p_1, \ldots, p_k\}) \cap L$ is in FR. Therefore also $L_2^{(2)} \in$ FR by Theorem 36, and $L_1^{(2)} \in$ CF because $L^{(2)} = L_1^{(2)} \cup L_2^{(2)} \in$ CF. We have $L_1^{(2)} \subseteq Q^{(2)}$, and by Theorem 38 follows that L_1 is a subset of a FIP-set. L_2 is a subset of a FIP-set by case a), and so is $L = L_1 \cup L_2$. □

Now it is easy to clarify the situation for the n-th power of a regular or even context-free set for an arbitrary natural number n, where it is trivial that $L^{(0)} = \{\epsilon\}$, $L^{(1)} = L$.

Theorem 40 ([18]) *For an arbitrary context-free language* L *and a natural number* $n \geq 2$*, if* $L^{(n)}$ *is context-free, then either* $n \geq 3$ *and* $L \in FR$ *or* $n = 2$ *and* $L \cap \mathsf{Per} \in FR$*.*

Proof. If $n \geq 3$ and $\sqrt{L}$ is infinite then $L^{(n)} \notin \mathrm{CF}$ by Theorem 35. It is well-known that every context-free language over a single-letter alphabet is regular. Using this fact it is easy to show that every context-free language with finite root is regular too. Therefore, if $\sqrt{L}$ is finite and $L \in \mathrm{CF}$ then $L \in \mathrm{FR}$, and $L^{(n)} \in \mathrm{FR}$ by Theorem 34. If $n = 2$, $L^{(n)} \in \mathrm{CF}$ and $\sqrt{L}$ is infinite, then $L \cap \mathsf{Per} \in \mathrm{FR}$ must be true by the proof of Theorem 39. □

Now we consider the full power $pow(L) = pow_{\mathbb{N}}(L)$ for a regular language L.

Theorem 41 (Fazekas [6]) *For a regular language* L*,* $pow(L)$ *is regular if and only if* $pow(L) \setminus L \in FR$*.*

Proof. If $pow(L) \setminus L \in \mathrm{FR} \subseteq \mathrm{REG}$ then $(pow(L) \setminus L) \cup L = pow(L) \in \mathrm{REG}$ because the class of regular languages is closed under union. For the opposite direction assume $pow(L) \in \mathrm{REG}$. Then also $L' =_{\mathrm{Df}} pow(L) \setminus L$ is regular because the class of regular languages is closed under difference of two sets. There are no primitive words in L' and therefore, by Theorem 20, it must have a finite root. □

6 Decidability questions

Questions about the decidability of several properties of sets or decidability of problems belong to the most important questions in (theoretical) computer science. Here we consider the decidability of properties of languages regarding their roots and powers. We will cite the most important theorems in chronological order of their proofs but we omit the proofs because of their complexity.

Theorem 42 (Horváth and Ito [11]) *For a context-free language* L *it is decidable whether* $\sqrt{L}$ *is finite.*

Theorem 43 (Cachat [2]) *For a regular or context-free language* L *over single-letter alphabet it is decidable whether* $pow(L)$ *is regular.*

Using Cachat's algorithm, Horváth showed (but not yet published) the following.

Theorem 44 (Horváth) *For a regular or context-free language* L *with finite root it is decidable whether* $\mathsf{pow}(\mathsf{L})$ *is regular.*

Remark. Since the context-free languages with finite root are exactly the languages in FR (Remark in the proof of Theorem 40), it doesn't matter whether we speak of regularity or context-freeness in the last theorems.

Remarkable in this connection is also the only negative decidability result by Bordihn.

Theorem 45 (Bordihn [1]) *For a context-free language* L *with infinite root it is not decidable whether* $\mathsf{pow}(\mathsf{L})$ *is context-free.*

The problem of Calbrix and Nivat [3] and the open question of Cachat [2] for languages over any finite alphabet and almost any sets of exponents, but not for all, was answered in [13]. Especially the regularity of $\mathsf{pow}(\mathsf{L})$ for a regular set L remained open, but it was conjectured that the latter is decidable. Using these papers, finally Fazekas [6] could prove this conjecture.

Theorem 46 (Fazekas [6]) *For a regular language* L *it is decidable whether* $\mathsf{pow}(\mathsf{L})$ *is regular.*

Finally, we look at the squares of regular and context-free languages.

Theorem 47 ([18]) *For a regular language* L *it is decidable whether* $\mathsf{L}^{(2)}$ *is regular or context-free or none of them.*

Proof. Let L be a regular language generated by a right-linear grammar $\mathsf{G} = [\Sigma, \mathsf{N}, \mathsf{S}, \mathsf{R}]$ and let $m = |\mathsf{N}| + 1$. By Theorem 36, $\mathsf{L}^{(2)}$ is regular if and only if $\sqrt{\mathsf{L}}$ is finite. The latter is decidable by Theorem 42. If $\sqrt{\mathsf{L}}$ is infinite then by Theorem 39, $\mathsf{L}^{(2)}$ is context-free if and only if L is a subset of a FIP-set. If L is a subset of a FIP-set then we can show that there exists a FIP-set F such that $\mathsf{L} \subseteq \mathsf{F}$ and all modules of all components of F have lengths smaller than m. Thus there are only finitely many words which can be modules and only finitely many inserted iterations of primitive words having these modules. The latter can be effectively computed. Let $\mathsf{L}_1, \ldots, \mathsf{L}_n$ be all these inserted iterations of primitive words. Then $\mathsf{L}^{(2)}$ is context-free if and only if $\mathsf{L} \subseteq \mathsf{L}_1 \cup \cdots \cup \mathsf{L}_n$ which is equivalent to $\mathsf{L} \cap \overline{(\mathsf{L}_1 \cup \cdots \cup \mathsf{L}_n)} = \emptyset$. The latter is decidable for regular languages L and $\mathsf{L}_1, \ldots, \mathsf{L}_n$. $\square$

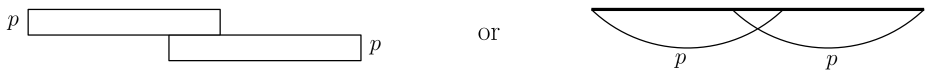

Figure 2: Concatenation with overlap

7 Generalizations of periodicity and primitivity

If u is a periodic word then we have a strict prefix v of u such that u is exhausted by concatenation of two or more copies of v, $u = v^n$, $n \geq 2$ (see Figure 3). But it could be that such an exhaustion is not completely possible, there may remain a strict prefix of v and the rest of v overhangs u, i.e. $u = v^n v'$, $n \geq 2$, $v' \sqsubset v$ (see Figure 4). In such case we call u to be semi-periodic. A third possibility is to exhaust u by concatenation of two or more copies of v where several consecutive copies may overlap (see Figure 5). In this case we speak about quasi-periodic words. If a nonempty word is not periodic, semi-periodic, or quasi-periodic, respectively, we call it a primitive, strongly primitive, or hyperprimitive word, respectively. Of course, periodic and primitive words are those we considered before in this paper. Finally, we can combine the possibilities to get three further types which we will summarize in the forthcoming Definition 49. Before doing so, we give a formal definition of concatenation with overlaps. All these generalizations have been introduced and detailed investigated in [15]. Most of the material in this section is taken from there.

Definition 48 *For* $p, q \in \Sigma^*$, *we define*
$p \otimes q =_{\mathrm{Df}} \{w_1w_2w_3 : w_1w_3 \neq \epsilon \wedge w_1w_2 = p \wedge w_2w_3 = q\}$,
$p^{\otimes 0} =_{\mathrm{Df}} \{\epsilon\}$, $p^{\otimes k+1} =_{\mathrm{Df}} \bigcup\{w \otimes p : w \in p^{\otimes k}\}$ *for* $k \in \mathbb{N}$,
$A \otimes B =_{\mathrm{Df}} \bigcup\{p \otimes q : p \in A \wedge q \in B\}$ *for sets* $A, B \subseteq \Sigma^*$.

The following example shows that in general, $p \otimes q$ is a set of words:
Let $p = aabaa$. Then $p \otimes p = p^{\otimes 2} = \{aabaaaabaa, aabaaabaa, aabaabaa\}$.
We can illustrate this by Figure 2.

In the following definition we repeat our Definitions 1 and 2 and give the generalizations suggested above.

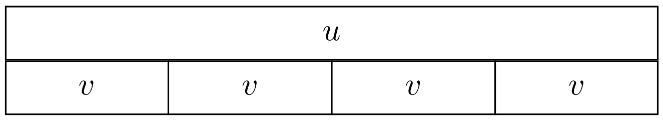

Figure 3: u is periodic, $u \in \mathsf{Per}$, $v = \mathsf{root}(u)$

Definition 49

$\mathsf{Per} =_{Df} \{u : \exists v \exists n(v \sqsubset u \wedge n \geq 2 \wedge u = v^n)\}$ *is the set of* **periodic** *words.*

$\mathsf{Q} =_{Df} \Sigma^+ \setminus \mathsf{Per}$ *is the set of* **primitive** *words.*

$\mathsf{SPer} =_{Df} \{u : \exists v \exists n(v \sqsubset u \wedge n \geq 2 \wedge u \in v^n \cdot \mathsf{Pr}(v))\}$ *is the set of* **semi-periodic** *words.*

$\mathsf{SQ} =_{Df} \Sigma^+ \setminus \mathsf{SPer}$ *is the set of* **strongly primitive** *words.*

$\mathsf{QPer} =_{Df} \{u : \exists v \exists n(v \sqsubset u \wedge n \geq 2 \wedge u \in v^{\otimes n})\}$ *is the set of* **quasi-periodic** *words.*

$\mathsf{HQ} =_{Df} \Sigma^+ \setminus \mathsf{QPer}$ *is the set of* **hyperprimitive** *words.*

$\mathsf{PSPer} =_{Df} \{u : \exists v \exists n(v \sqsubset u \wedge n \geq 2 \wedge u \in \{v^n\} \otimes \mathsf{Pr}(v))\}$ *is the set of* **pre-periodic** *words.*

$\mathsf{SSQ} =_{Df} \Sigma^+ \setminus \mathsf{PSPer}$ *is the set of* **super strongly primitive** *words.*

$\mathsf{SQPer} =_{Df} \{u : \exists v \exists n(v \sqsubset u \wedge n \geq 2 \wedge u \in v^{\otimes n} \cdot \mathsf{Pr}(v))\}$ *is the set of* **semi-quasi-periodic** *words.*

$\mathsf{SHQ} =_{Df} \Sigma^+ \setminus \mathsf{SQPer}$ *is the set of* **strongly hyperprimitive** *words.*

$\mathsf{QQPer} =_{Df} \{u : \exists v \exists n(v \sqsubset u \wedge n \geq 2 \wedge u \in v^{\otimes n} \otimes \mathsf{Pr}(v))\}$ *is the set of* **quasi-quasi-periodic** *words.*

$\mathsf{HHQ} =_{Df} \Sigma^+ \setminus \mathsf{QQPer}$ *is the set of* **hyperhyperprimitive** *words.*

The different kinds of generalized periodicity are illustrated in the Figures 3–8.

Theorem 50 *The sets from Definition 49 have the inclusion structure as given in Figure 9. The lines in this figure denote strict inclusion from bottom to top. Sets which are not connected by such a line are incomparable under inclusion.*

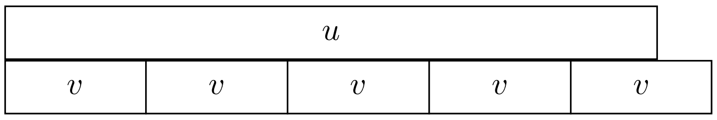

Figure 4: u is semi periodic, $u \in \mathsf{SPer}$, $v = \mathsf{sroot}(u)$

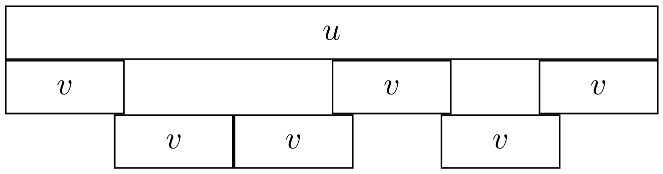

Figure 5: u is quasi-periodic, $u \in \mathsf{QPer}$, $v = \mathsf{hroot}(u)$

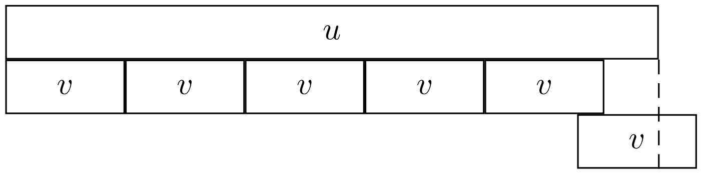

Figure 6: u is pre-periodic, $u \in \mathsf{PSPer}$, $v = \mathsf{ssroot}(u)$

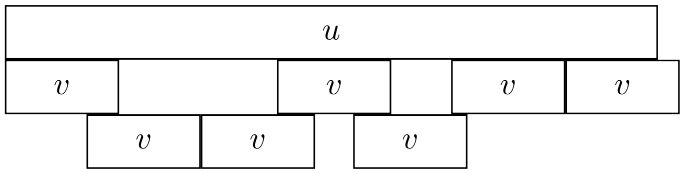

Figure 7: u is semi-quasi-periodic, $u \in \mathsf{SQPer}$, $v = \mathsf{shroot}(u)$

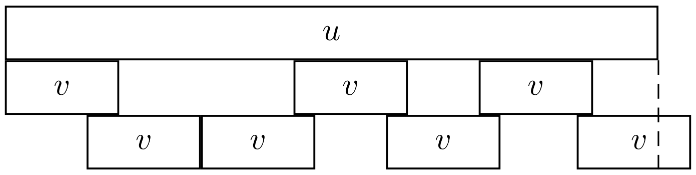

Figure 8: u is quasi-quasi-periodic, $u \in \mathsf{QQPer}$, $v = \mathsf{hhroot}(u)$

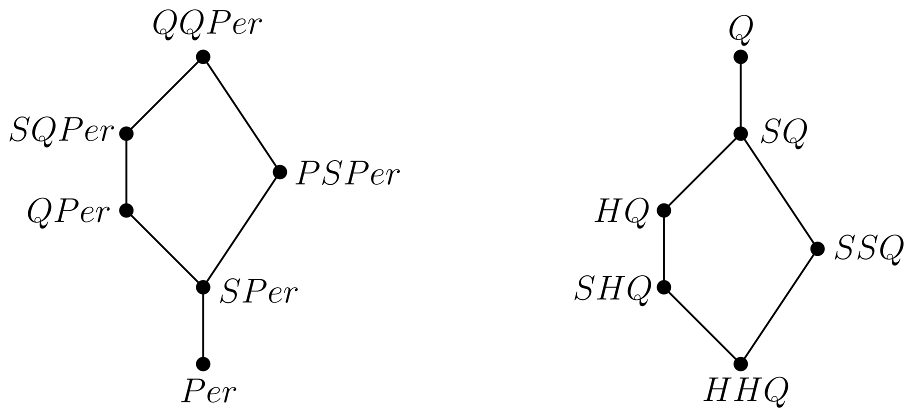

Figure 9: Inclusion structure

Proof. Because of the duality between the sets, it is enough to prove the left structure in Figure 9. Let $u \in \mathsf{SPer}$, it means, $u = v^n q$ where $n \geq 2$ and $q \sqsubset v$. Thus $v = qr$ for some $r \in \Sigma^*$ and $u = (qr)^n q \in (qrq)^{\otimes n}$ and therefore $u \in \mathsf{QPer}$ and $\mathsf{SPer} \subseteq \mathsf{QPer}$. The remaining inclusions are clear by the definition. To show the strictness of the inclusions we can use the following examples:
$u_1 = abaababab$, $u_2 = aababaababaabaab$, $u_3 = aabaaabaaba$,
$u_4 = abaabab$, $u_5 = ababa$.
Then $u_1 \in \mathsf{QQPer} \setminus (\mathsf{SQPer} \cup \mathsf{PSPer})$, $u_2 \in \mathsf{SQPer} \setminus \mathsf{QPer}$,
$u_3 \in \mathsf{QPer} \setminus \mathsf{PSPer}$, $u_4 \in \mathsf{PSPer} \setminus \mathsf{SQPer}$, and $u_5 \in \mathsf{SPer} \setminus \mathsf{Per}$.
u_3 and u_4 also prove the incomparability. $\square$

The six different kinds of periodicity resp. primitivity of words give rise to define six types of roots where the first one is again that from Definition 6.

Definition 51 *Let* $u \in \Sigma^+$.
The shortest word v *such that there exists a natural number* n *with*
$u = v^n$ *is called the* **root** *of* u, *denoted by* $\mathsf{root}(u)$.
The shortest word v *such that there exists a natural number* n *with*
$u \in v^n \cdot \mathsf{Pr}(v)$ *is called the* **strong root** *of* u, *denoted by* $\mathsf{sroot}(u)$.
The shortest word v *such that there exists a natural number* n *with*
$u \in v^{\otimes n}$ *is called the* **hyperroot** *of* u, *denoted by* $\mathsf{hroot}(u)$.
The shortest word v *such that there exists a natural number* n *with*
$u \in \{v^n\} \otimes \mathsf{Pr}(v)$ *is called the* **super strong root** *of* u, *denoted by* $\mathsf{ssroot}(u)$.
The shortest word v *such that there exists a natural number* n *with*

$u \in v^{\otimes n} \cdot \mathsf{Pr}(v)$ *is called the* **strong hyperroot** *of* u*, denoted by* $\mathsf{shroot}(u)$*.* *The shortest word* v *such that there exists a natural number* n *with* $u \in v^{\otimes n} \otimes \mathsf{Pr}(v)$ *is called the* **hyperhyperroot** *of* u*, denoted by* $\mathsf{hhroot}(u)$*.* *If* L *is a language, then* $\mathsf{root}(L) =_{\mathrm{Df}} \{\mathsf{root}(p) : p \in L \wedge p \neq \epsilon\}$ *is the* **root of** L*. Analogously* $\mathsf{sroot}(L)$*,* $\mathsf{hroot}(L)$*,* $\mathsf{ssroot}(L)$*,* $\mathsf{shroot}(L)$ *and* $\mathsf{hhroot}(L)$ *are defined.*

The six kinds of roots are illustrated in the Figures 3–8 (if v is the shortest prefix with the appropriate property).

root, sroot, hroot, ssroot, shroot and hhroot are word functions over Σ^+, i.e., functions from Σ^+ to Σ^+. Generally, for word functions we define the following partial ordering, also denoted by $\sqsubseteq$.
$\mathsf{dom}(f)$ for a function f denotes the **domain** of f.

Definition 52 *For word functions* f *and* g *having the same domain,* $f \sqsubseteq g =_{\mathrm{Df}} \forall u(u \in \mathsf{dom}(f) \rightarrow f(u) \sqsubseteq g(u))$*.*

Theorem 53 *The partial ordering* $\sqsubseteq$ *for the functions from Definition 51 is given in Figure 10.*

Proof. It follows from the definition, that for an arbitrary word $u \in \Sigma^+$ and its roots we have the prefix relationship as shown in the figure. It remains to show the strict prefixes and incomparability. This can be done, for instance, by the following examples. Let $u_1 = \mathsf{abaabaababaabaabab}$, $u_2 = \mathsf{abaabaabab}$, and $u_3 = \mathsf{abaababaabaabaab}$. Then
$\mathsf{hhroot}(u_1) = \mathsf{aba} \sqsubset \mathsf{shroot}(u_1) = \mathsf{abaab} \sqsubset \mathsf{ssroot}(u_1) = \mathsf{sroot}(u_1) = \mathsf{abaabaab} \sqsubset \mathsf{hroot}(u_1) = \mathsf{abaabaabab} \sqsubset \mathsf{root}(u_1) = u_1$,
$\mathsf{ssroot}(u_2) = \mathsf{aba} \sqsubset \mathsf{shroot}(u_2) = \mathsf{abaab} \sqsubset \mathsf{sroot}(u_2) = \mathsf{abaabaab} \sqsubset \mathsf{hroot}(u_2) = u_2$, and
$\mathsf{hroot}(u_3) = \mathsf{abaab} \sqsubset \mathsf{sroot}(u_3) = \mathsf{abaababaaba}$, which proves our figure.
□

For most words u, some of the six roots coincide, and we have the question how many roots of u are different, and whether there exist words u such that all the six roots of u are different from each other. This last question was raised in [15], and it was first assumed that they do not exist. But in 2010 Georg Lohmann discovered the first of such words.

Definition 54 *Let* $k \in \{1,2,3,4,5,6\}$*. A word* $u \in \Sigma^+$ *is called a* **k-root word** *if*
$|\{\mathsf{root}(u), \mathsf{sroot}(u), \mathsf{hroot}(u), \mathsf{ssroot}(u), \mathsf{shroot}(u), \mathsf{hhroot}(u)\}| = k$.

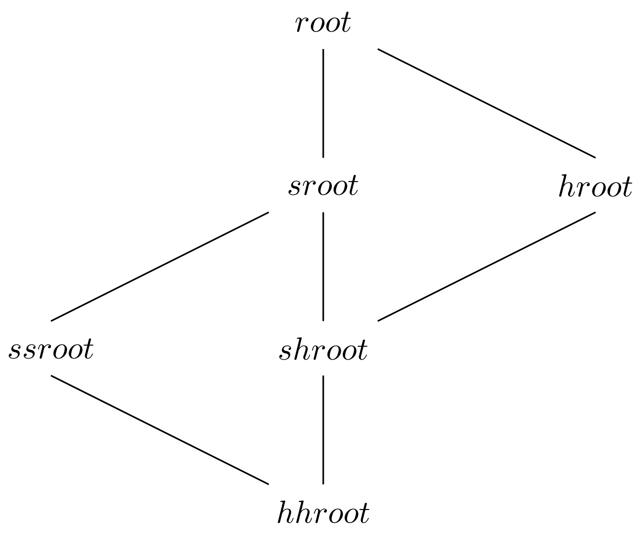

Figure 10: Partial ordering of the root-functions

A 6-root word is also called a **Lohmann word***.*
u *is called a* **strong** k**-root word** *if it is a* k*-root word and* $\mathsf{root}(u) \neq u$*, it means, it is a periodic* k*-root word.*

The following theorems give answers to our questions. The proofs are easy or will be published elsewhere.

Theorem 55 *The lexicographic smallest* k*-root words are* a *for* $k = 1$*,* aba *for* $k = 2$*,* $ababa$ *for* $k = 3$*,* $abaabaabab$ *for* $k = 4$*,* $abaabaababaabaabab$ *for* $k = 5$*, and* $ababaabababaababaababababaabab$ *for* $k = 6$*.*
The lexicographic smallest strong k*-root words are* aa *for* $k = 1$*,* $abaababaab$ *for* $k = 2$*,* $(ab^3abab^3abab^3)^2$ *for* $k = 3$*, and* $(ababaabababaabab)^2$ *for* $k = 4$*.*

Theorem 56 *There exist no strong* k*-root words for* $k = 5$ *and* $k = 6$*.*

Theorem 57 *Let* v *and* w *be words such that* $\epsilon \sqsubset v \sqsubset w$*,* $wv \not\sqsubseteq p^l$ *for some* $p \sqsubset w$ *and* $l > 1$ *and* k_1, k_2, k_3 *be natural numbers with* $2 \leq k_1 < k_2 < k_3 \leq 2k_1$*. Then* $u = w^{k_1}vw^{k_2}vw^{k_1}vw^{k_3}vw^{k_3-k_1}$ *is a Lohmann word.*

It is still open whether the sufficient condition in the last theorem is also a necessary condition for Lohmann words.

Let us now examine whether the results from the former sections are also true for generalized periodicity and primitivity. First, we give generalizations of Corollary 9 and Theorem 13. For their proofs we refer to [15].

Lemma 58 *$w \notin \mathsf{SQ}$ if and only if $w = pq = qr$ for some $p, q, r \in \Sigma^+$ and $|q| \geq \frac{|w|}{2}$.*

Lemma 59 *If $aw \notin \mathsf{SQ}$ and $wb \notin \mathsf{SQ}$, where $w \in \Sigma^+$ and $a, b \in \Sigma$, then $awb \notin \mathsf{SQ}$.*

Lemma 60 *If $aw \notin \mathsf{HQ}$ and $wb \notin \mathsf{HQ}$, where $w \in \Sigma^+$ and $a, b \in \Sigma$, then $awb \notin \mathsf{HQ}$.*

Theorem 19 remains true for each of the sets from Definition 49. The Theorems 21, 22, and 24 with their proofs are passed to each of the languages SQ, HQ, SSQ, SHQ, and HHQ. Also the non-context-freeness of each of the sets of generalized periodic words is simple as remarked after Theorem 21. The context-freeness of the sets of generalized primitive words is open just as that of Q.

Using Lemma 59 and Lemma 60 it can be shown that Theorem 27 is also true for SQ and HQ. Also none of SSQ, SHQ, HHQ, and the sets of generalized periodic words is a contextual language of any kind.

Theorem 30 and its proof remain true for each of the sets from Definition 49. Theorem 29 is true for SQ where the proof uses Lemma 58. Whether the time bound n^2 is also optimal for accepting one of the remaining sets remains open. Theorem 31 and its proof remain true for each of the roots from Definition 51.

Acknowledgements

The author is grateful to Antal Iványi in Budapest for his suggestion to write this paper, for Martin Hünniger in Jena for his help with the figures, and to Peter Leupold in Kassel and to the anonymous referee for some hints.

This work was supported by the project under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003 (Eötvös Loránd University, Budapest) financed by the European Union and the European Social Fund.

References

[1] H. Bordihn, Context-freeness of the power of context-free languages is undecidable, *Theoret. Comput. Sci.* **314,** 3 (2004) 445–449. ⇒24

[2] T. Cachat, The power of one-letter rational languages, *Proc. 5th International Conference Developments in Language Theory*, Wien, July 16–21, 2001, *Lecture Notes in Comput. Sci.* **2295** (2002) 145–154. ⇒18, 23, 24

[3] H. Calbrix, M. Nivat, Prefix and period languages of rational ω-languages, *Proc. Developments in Language Theory II, At the Crossroads of Mathematics, Computer Science and Biology,* Magdeburg, Germany, July 17–21, 1995, World Scientific, 1996, pp. 341–349. ⇒18, 24

[4] P. Dömösi, S. Horváth, M. Ito, On the connection between formal languages and primitive words, *Proc. First Session on Scientific Communication*, Univ. of Oradea, Oradea, Romania, June 1991, pp. 59–67. ⇒ 14

[5] P. Dömösi, M. Ito, S. Marcus, Marcus contextual languages consisting of primitive words, *Discrete Math.* **308,** 21 (2008) 4877–4881. ⇒16

[6] S. Z. Fazekas, Powers of regular languages, *Proc. Developments in Language Theory*, Stuttgart 2009, *Lecture Notes in Comput. Sci.* **5583** (2009) 221–227. ⇒23, 24

[7] N. J. Fine, H. S. Wilf, Uniqueness theorems for periodic functions, *Proc. Amer. Math. Soc.*, **16,** 1 (1965) 109–114. ⇒10

[8] M. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978. ⇒6, 12

[9] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979. ⇒ 6, 12

[10] S. Horváth, Strong interchangeability and nonlinearity of primitive words, *Proc. Algebraic Methods in Language Processing*, Univ. of Twente, Enschede, the Netherlands, December 1995, pp. 173–178. ⇒14

[11] S. Horváth, M. Ito, Decidable and undecidable problems of primitive words, regular and context-free languages, *J. UCS* **5,** 9 (1999) 532–541. ⇒23

[12] S. Horváth, M. Kudlek, On classification and decidability problems of primitive words, *Pure Math. Appl.* **6,** 2–3 (1995) 171–189. ⇒17

[13] S. Horváth, P. Leupold, G. Lischke, Roots and powers of regular languages, *Proc. 6th International Conference Developments in Language Theory*, Kyoto 2002, *Lecture Notes in Comput. Sci.* **2450** (2003) 220–230 ⇒18, 19, 20, 21, 24

[14] M. Ito, M. Katsura, Context-free languages consisting of non-primitive words, *Internat. J. Comput. Math.* **40,** 3–4 (1991) 157–167. ⇒22

[15] M. Ito, G. Lischke, Generalized periodicity and primitivity for words, *Math. Log. Quart.* **53,** 1 (2007) 91–106. ⇒15, 18, 25, 29, 31

[16] M. Ito, G. Lischke, Corrigendum to "Generalized periodicity and primitivity for words", *Math. Log. Quart.* **53,** 6 (2007) 642–643. ⇒18

[17] G. Lischke, The root of a language and its complexity, *Proc. 5th International Conference Developments in Language Theory*, Wien 2001, *Lecture Notes in Comput. Sci.*, **2295** (2002) 272–280 ⇒17, 18

[18] G. Lischke, Squares of regular languages, *Math. Log. Quart.*, **51,** 3 (2005) 299–304. ⇒22, 23, 24

[19] M. Lothaire, *Combinatorics on Words*, Addison-Wesley, Reading, MA, 1983. ⇒9, 10

[20] R. C. Lyndon, M. P. Schützenberger, On the equation $a^M = b^N c^P$ in a free group, *Michigan Math. J.*, **9,** 4 (1962) 289–298. ⇒9, 10, 11

[21] G. Păun, *Marcus Contextual Grammars*, Kluwer, Dordrecht-Boston-London, 1997. ⇒15

[22] H. Petersen, The ambiguity of primitive words, *Proc. STACS 94, Lecture Notes in Comput. Sci.*, **775** (1994) 679–690. ⇒14

[23] G. Rozenberg, A. Salomaa (Eds.), *Handbook of Formal Languages, Vol. 1*, Springer, Berlin-Heidelberg, 1997. ⇒6, 9, 12

[24] H. J. Shyr, *Free Monoids and Languages*, Hon Min Book Company, Taichung, 1991. ⇒9

[25] H. J. Shyr, G. Thierrin, Codes and binary relations, *Séminare d'Algèbre, Paul Dubreil*, Paris 1975–1976, *Lecture Notes in Math.* **586** (1977) 180–188. ⇒9

[26] H. J. Shyr, G. Thierrin, Disjunctive languages and codes, *Proc. International Conference Mathematical Foundations of Computer Science*, Poznan 1977, *Lecture Notes in Comput. Sci.* **56** (1977) 171–176 ⇒11

[27] H. J. Shyr, S. S. Yu, Non-primitive words in the language $\mathtt{p}^+\mathtt{q}^+$, *Soochow J. Math.* **20,** 4 (1994) 535–546. ⇒12

[28] S. S. Yu, *Languages and Codes*, Tsang Hai Book Publishing Co., Taichung, 2005. ⇒9, 12

Received: December 16, 2010 • Revised: February 22, 2011

Acta Univ. Sapientiae, Informatica, **3,** 1 (2011) 35–47

Arc-preserving subsequences of arc-annotated sequences

Vladimir Yu. POPOV
Department of Mathematics and Mechanics
Ural State University
620083 Ekaterinburg, RUSSIA
email: Vladimir.Popov@usu.ru

Abstract. Arc-annotated sequences are useful in representing the structural information of RNA and protein sequences. The longest arc-preserving common subsequence problem has been introduced as a framework for studying the similarity of arc-annotated sequences. In this paper, we consider arc-annotated sequences with various arc structures. We consider the longest arc preserving common subsequence problem. In particular, we show that the decision version of the 1-FRAGMENT LAPCS(CROSSING,CHAIN) and the decision version of the 0-DIAGONAL LAPCS(CROSSING,CHAIN) are **NP**-complete for some fixed alphabet Σ such that $|\Sigma| = 2$. Also we show that if $|\Sigma| = 1$, then the decision version of the 1-FRAGMENT LAPCS(UNLIMITED, PLAIN) and the decision version of the 0-DIAGONAL LAPCS(UNLIMITED, PLAIN) are **NP**-complete.

1 Introduction

Algorithms on sequences of symbols have been studied for a long time and now form a fundamental part of computer science. One of the very important problems in analysis of sequences is the longest common subsequence (LCS) problem. The computational problem of finding the longest common subsequence of a set of k strings has been studied extensively over the last thirty years (see [5, 19, 21] and references). This problem has many applications.

Computing Classification System 1998: F.1.3
Mathematics Subject Classification 2010: 68Q15
Key words and phrases: longest common subsequence, sequence annotation, **NP**-complete

When $k = 2$, the longest common subsequence is a measure of the similarity of two strings and is thus useful in molecular biology, pattern recognition, and text compression [26, 27, 34]. The version of LCS in which the number of strings is unrestricted is also useful in text compression [27], and is a special case of the multiple sequence alignment and consensus subsequence discovery problem in molecular biology [11, 12, 32].

The k-unrestricted LCS problem is **NP**-complete [27]. If the number of sequences is fixed at k with maximum length n, their longest common subsequence can be found in $O(n^{k-1})$ time, through an extension of the pairwise algorithm [21]. Suppose $|S_1| = n$ and $|S_2| = m$, the longest common subsequence of S_1 and S_2 can be found in time $O(nm)$ [8, 18, 35].

Sequence-level investigation has become essential in modern molecular biology. But to consider genetic molecules only as long sequences consisting of the 4 basic constituents is too simple to determine the function and physical structure of the molecules. Additional information about the sequences should be added to the sequences. Early works with these additional information are primary structure based, the sequence comparison is basically done on the primary structure while trying to incorporate secondary structure data [2, 9]. This approach has the weakness that it does not treat a base pair as a whole entity. Recently, an improved model was proposed [13, 14].

Arc-annotated sequences are useful in describing the secondary and tertiary structures of RNA and protein sequences. See [13, 4, 16, 22, 23] for further discussion and references. Structure comparison for RNA and for protein sequences has become a central computational problem bearing many challenging computer science questions. In this context, the longest arc preserving common subsequence problem (LAPCS) recently has received considerable attention [13, 14, 22, 23, 25]. It is a sound and meaningful mathematical formalization of comparing the secondary structures of molecular sequences. Studies for this problem have been undertaken in [5, 16, 1, 3, 6, 7, 10, 15, 20, 28, 29, 30, 33].

2 Preliminaries and problem definitions

Given two sequences S and T over some fixed alphabet Σ, the sequence T is a subsequence of S if T can be obtained from S by deleting some letters from S. Notice that the order of the remaining letters of S bases must be preserved. The length of a sequence S is the number of letters in it and is denoted as $|S|$. For simplicity, we use $S[i]$ to denote the ith letter in sequence S, and $S[i, j]$ to denote the substring of S consisting of the ith letter through the jth letter.

Given two sequences S_1 and S_2 (over some fixed alphabet Σ), the classic longest common subsequence problem asks for a longest sequence T that is a subsequence of both S_1 and S_2.

An arc-annotated sequence of length n on a finite alphabet Σ is a couple $A = (S, P)$ where S is a sequence of length n on Σ and P is a set of pairs (i_1, i_2), with $1 \leq i_1 < i_2 \leq n$. In this paper we will then call an element of S a base. A pair $(i_1, i_2) \in P$ represents an arc linking bases $S[i_1]$ and $S[i_2]$ of S. The bases $S[i_1]$ and $S[i_2]$ are said to belong to the arc (i_1, i_2) and are the only bases that belong to this arc.

Given two annotated sequences S_1 and S_2 with arc sets P_1 and P_2 respectively, a common subsequence T of S_1 and S_2 induces a bijective mapping from a subset of $\{1, \ldots, |S_1|\}$ to subset of $\{1, \ldots, |S_2|\}$. The common subsequence T is arc-preserving if the arcs induced by the mapping are preserved, i.e., for any (i_1, j_1) and (i_2, j_2) in the mapping,

$$(i_1, i_2) \in P_1 \Leftrightarrow (j_1, j_2) \in P_2.$$

The LAPCS problem is to find a longest common subsequence of S_1 and S_2 that is arc-preserving (with respect to the given arc sets P_1 and P_2) [13].

LAPCS:

INSTANCE: An alphabet Σ, annotated sequences S_1 and S_2, $S_1, S_2 \in \Sigma^*$, with arc sets P_1 and P_2 respectively.

QUESTION: Find a longest common subsequence of S_1 and S_2 that is arc-preserving.

The arc structure can be restricted. We consider the following four natural restrictions on an arc set P which are first discussed in [13]:

1. no sharing of endpoints:
   $\forall (i_1, i_2), (i_3, i_4) \in P, i_1 \neq i_4, i_2 \neq i_3$, and $i_1 = i_3 \Leftrightarrow i_2 = i_4$.
2. no crossing:
   $\forall (i_1, i_2), (i_3, i_4) \in P, i_1 \in [i_3, i_4] \Leftrightarrow i_2 \in [i_3, i_4]$.
3. no nesting:
   $\forall (i_1, i_2), (i_3, i_4) \in P, i_1 \leq i_3 \Leftrightarrow i_2 \leq i_3$.
4. no arcs:
   $P = \emptyset$.

These restrictions are used progressively and inclusively to produce five distinct levels of permitted arc structures for LAPCS:

– UNLIMITED — no restrictions;
– CROSSING — restriction 1;
– NESTED — restrictions 1 and 2;

– CHAIN — restrictions 1, 2 and 3;
– PLAIN — restriction 4.

The problem LAPCS is varied by these different levels of restrictions as LAPCS(x, y) which is problem LAPCS with S_1 having restriction level x and S_2 having restriction level y. Without loss of generality, we always assume that x is the same level or higher than y.

We give the definitions of two special cases of the LAPCS problem, which were first studied in [25]. The special cases are motivated from biological applications [17, 24].

THE c-FRAGMENT LAPCS PROBLEM $(c \geq 1)$:

INSTANCE: An alphabet Σ, annotated sequences S_1 and S_2, $S_1, S_2 \in \Sigma^*$, with arc sets P_1 and P_2 respectively, where S_1 and S_2 are divided into fragments of lengths exactly c (the last fragment can have a length less than c).

QUESTION: Find a longest common subsequence of S_1 and S_2 that is arc-preserving. The allowed matches are those between fragments at the same location.

The c-DIAGONAL LAPCS problem, $(c \geq 0)$, is an extension of the c-FRAGMENT LAPCS problem, where base $S_2[i]$ is allowed only to match bases in the range $S_1[i - c, i + c]$.

The c-DIAGONAL LAPCS and c-FRAGMENT LAPCS problems are relevant in the comparison of conserved RNA sequences where we already have a rough idea about the correspondence between bases in the two sequences.

3 Previous results

It is shown in [25] that the 1-FRAGMENT LAPCS(CROSSING, CROSSING) and 0-DIAGONAL LAPCS(CROSSING, CROSSING) are solvable in time $O(n)$. An overview on known **NP**-completeness results for c-DIAGONAL LAPCS and c-FRAGMENT LAPCS is given in Figure 1.

	unlimited	crossing	nested	chain	plain
unlimited	**NP**-h [25]	**NP**-h [25]	**NP**-h [25]	?	?
crossing	—	**NP**-h [25]	**NP**-h [25]	?	?
nested	—	—	**NP**-h [25]	?	?

Figure 1: **NP**-completeness results for c-DIAGONAL LAPCS (with $c \geq 1$) and c-FRAGMENT LAPCS (with $c \geq 2$)

4 The c-FRAGMENT LAPCS(UNLIMITED,PLAIN) and the c-DIAGONAL LAPCS(UNLIMITED,PLAIN) problem

Let us consider the decision version of the c-FRAGMENT LAPCS problem.

INSTANCE: An alphabet Σ, a positive integer k, annotated sequences S_1 and S_2, $S_1, S_2 \in \Sigma^*$, with arc sets P_1 and P_2 respectively, where S_1 and S_2 are divided into fragments of lengths exactly c (the last fragment can have a length less than c).

QUESTION: Is there a common subsequence T of S_1 and S_2 that is arc-preserving, $|T| \geq k$? (The allowed matches are those between fragments at the same location).

Similarly, we can define the decision version of the c-DIAGONAL LAPCS problem.

Theorem 1 *If* $|\Sigma| = 1$*, then* 1-FRAGMENT LAPCS(UNLIMITED, PLAIN) *and* 0-DIAGONAL LAPCS(UNLIMITED, PLAIN) *are* **NP***-complete.*

Proof. It is easy to see that 1-FRAGMENT LAPCS(UNLIMITED, PLAIN) = 0-DIAGONAL LAPCS(UNLIMITED, PLAIN).

Let $G = (V, E)$ be an undirected graph, and let $I \subseteq V$. We say that the set I is independent if whenever $i, j \in I$ then there is no edge between i and j. We make use of the following problem:

INDEPENDENT SET (IS): INSTANCE: A graph $G = (V, E)$, a positive integer k.

QUESTION: Is there an independent set I, $I \subseteq V$, with $|I| \geq k$?

IS is **NP**-complete (see [31]).

Let us suppose that $\Sigma = \{a\}$. We will show that IS can be polynomially reduced to problem 1-FRAGMENT LAPCS(UNLIMITED, PLAIN).

Let $\langle G = (V, E), V = \{1, 2, \ldots, n\}, k\rangle$ be an instance of IS. Now we transform an instance of the IS problem to an instance of the 1-FRAGMENT LAPCS(UNLIMITED, PLAIN) problem as follows.

- $S_1 = S_2 = a^n$.
- $P_1 = E, P_2 = \emptyset$.
- $\langle (S_1, P_1), (S_2, P_2), k\rangle$.

First suppose that the graph G has an independent set I of size k. By definition of independent set, $(i, j) \notin E$ for each $i, j \in I$. For a given subset I, let

$$M = \{(i, i) : i \in I\}.$$

Since I is an independent set, if $(i, j) \in E = P_1$ then either $(i, i) \notin M$ or

$(j, j) \notin M$. This preserves arcs since P_2 is empty. Clearly, $S_1[i] = S_2[i]$ for each $i \in I$, and the allowed matches are those between fragments at the same location. Therefore, there is a common subsequence T of S_1 and S_2 that is arc-preserving, $|T| = k$, and the allowed matches are those between fragments at the same location.

Now suppose that there is a common subsequence T of S_1 and S_2 that is arc-preserving, $|T| = k$, and the allowed matches are those between fragments at the same location. In this case there is a valid mapping M, with $|M| = k$. Since $c = 1$, it is easy to see that if $(i, j) \in M$ then $i = j$. Let

$$I = \{i : (i, i) \in M\}.$$

Clearly,

$$|I| = |M| = k.$$

Let i_1 and i_2 be any two distinct members of I. Then let $(i_1, j_1), (i_2, j_2) \in M$. Since

$$i_1 = j_1, i_2 = j_2, i_1 \neq i_2,$$

it is easy to see that $j_1 \neq j_2$. Since P_2 is empty, $(j_1, j_2) \notin P_2$, so $(i_1, i_2) \notin P_1$. Since $P_1 = E$, the set I of vertices is a size k independent set of G. □

5 The c-FRAGMENT LAPCS(CROSSING,CHAIN) and the c-DIAGONAL LAPCS(CROSSING,CHAIN) problem

Theorem 2 *If* $|\Sigma| = 2$*, then* 1-FRAGMENT LAPCS(CROSSING,CHAIN) *and* 0-DIAGONAL LAPCS(CROSSING,CHAIN) *are* **NP**-*complete.*

Proof. It is easy to see that 1-FRAGMENT LAPCS(CROSSING, CHAIN) = 0-DIAGONAL LAPCS(CROSSING, CHAIN).

Let us suppose that $\Sigma = \{a, b\}$. We will show that IS can be polynomially reduced to problem 1-FRAGMENT LAPCS(CROSSING, CHAIN).

Let $\langle G = (V, E), V = \{1, 2, \ldots, n\}, k\rangle$ be an instance of IS. Note that IS remains **NP**-complete when restricted to connected graphs with no loops and multiple edges. Let $G = (V, E)$ be such a graph. Now we transform an instance of the IS problem to an instance of the 1-FRAGMENT LAPCS(CROSSING, CHAIN) problem as follows.

There are two cases to consider.

Case I. $k > n$

- $S_1 = S_2 = a$
- $P_1 = P_2 = \emptyset$
- $\langle (S_1, P_1), (S_2, P_2), k \rangle$

Clearly, if I is an independent set, then $I \subseteq V$ and $|I| \le |V| = n$. Therefore, there is no an independent set I, with $|I| \ge k$.

Since $k > n$ and $n \in \{1, 2, \dots\}$, it is easy to see that $k > 1$. Since $S_1 = S_2 = a$ and $P_1 = P_2 = \emptyset$, $T = a$ is the longest arc-preserving common subsequence. Therefore, there is no an arc-preserving common subsequence T such that $|T| \ge k$.

Case II. $k \le n$

- $S_1 = S_2 = (ba^n b)^n$
- Let $\alpha < \beta$. Then

$$(\alpha, \beta) \in P_1 \Leftrightarrow [\exists i \in \{1, 2, \dots, n\} \exists j \in \{1, 2, \dots, n\}$$

$$((i, j) \in E \wedge \alpha = (i-1)(n+2) + j + 1 \wedge$$

$$\wedge \beta = (j-1)(n+2) + i + 1)] \vee$$

$$\vee [\exists i \in \{1, 2, \dots, n\} (\alpha = (i-1)(n+2) + 1 \wedge \beta = i(n+2))],$$

$$(\alpha, \beta) \in P_2 \Leftrightarrow \exists i \in \{1, 2, \dots, n\}$$

$$(\alpha = (i-1)(n+2) + 1 \wedge \beta = i(n+2)).$$

- $\langle (S_1, P_1), (S_2, P_2), k(n+2) \rangle$

First suppose that G has an independent set I of size k. By definition of independent set, $(i, j) \notin E$ for each $i, j \in I$. For a given subset I, let

$$M = \{(j, j) : j = (n+2)(i-1) + l, i \in I,$$

$$l \in \{1, 2, \dots, n+2\}\}.$$

Let $(j, j) \in M$, and there exist i such that $j = (n+2)(i-1)+1$. By definition of M,

$$((n+2)(i-1) + 1, (n+2)(i-1) + 1) \in M \Leftrightarrow$$

$$\Leftrightarrow ((n+2)i, (n+2)i) \in M.$$

By definition of P_l, $((n+2)(i-1)+1,(n+2)i) \in P_l$ where $l = 1,2$. Let $(j,j) \in M$, and there exist i such that $j = (n+2)i$. By definition of M,

$$((n+2)i,(n+2)i) \in M \Leftrightarrow$$

$$\Leftrightarrow ((n+2)(i-1)+1,(n+2)(i-1)+1) \in M.$$

By definition of P_l,

$$((n+2)(i-1)+1,(n+2)i) \in P_l$$

where $l = 1,2$. Let $(j,j) \in M$, and

$$j = (n+2)(i-1)+l$$

where $1 < l < n+2$. By definition of M, $i \in$ I. Since I is an independent set, if $(i,l-1) \in E$ then $l-1 \notin$ I. Since

$$1 < l < n+2,$$

by definition of P_1, either

$$((n+2)(i-1)+l,(n+2)(l-2)+i+1) \in P_1$$

or

$$((n+2)(i-1)+l,t) \notin P_1$$

for each t. Since

$$1 < l < n+2,$$

by definition of P_2,

$$((n+2)(i-1)+l,t) \notin P_2$$

for each t. If

$$((n+2)(i-1)+l,(n+2)(l-2)+i+1) \in P_1,$$

then in view of $l-1 \notin$ I,

$$((n+2)(l-2)+i+1,(n+2)(l-2)+i+1) \notin M.$$

This preserves arcs. Since $|\mathrm{I}| = k$, it is easy to see that

$$|M| = k(n+2).$$

Clearly, $S_1[i] = S_2[i]$ for each $i \in I$, and the allowed matches are those between fragments at the same location. Therefore, there is a common subsequence T of S_1 and S_2 that is arc-preserving, $|T| = k(n+2)$, and the allowed matches are those between fragments at the same location.

Now suppose that there is a common subsequence T of S_1 and S_2 that is arc-preserving, $|T| = k$, and the allowed matches are those between fragments at the same location. In this case there is a valid mapping M, with $|M| = k$. Since $c = 1$, it is easy to see that if $(i,j) \in M$ then $i = j$. Let $I = \{i : (i,i) \in M\}$. Clearly, $|I| = |M| = k$. Let i_1 and i_2 be any two distinct members of I. Then let $(i_1, j_1), (i_2, j_2) \in M$. Since $i_1 = j_1, i_2 = j_2, i_1 \neq i_2$, it is easy to see that $j_1 \neq j_2$. Since P_2 is empty, $(j_1, j_2) \notin P_2$, so $(i_1, i_2) \notin P_1$. Since $P_1 = E$, the set I of vertices is a size k independent set of G. □

6 Conclusions

In this paper, we considered two special cases of the LAPCS problem, which were first studied in [25]. We have shown that the decision version of the 1-FRAGMENT LAPCS(CROSSING,CHAIN) and the decision version of the 0-DIAGONAL LAPCS(CROSSING,CHAIN) are **NP**-complete for some fixed alphabet Σ such that $|\Sigma| = 2$. Also we have shown that if $|\Sigma| = 1$, then the decision version of the 1-FRAGMENT LAPCS(UNLIMITED, PLAIN) and the decision version of the 0-DIAGONAL LAPCS(UNLIMITED, PLAIN) are **NP**-complete. This results answers some open questions in [16] (see Table 4.2. in [16]).

Acknowledgements

The work was partially supported by Grant of President of the Russian Federation MD-1687.2008.9 and Analytical Departmental Program "Developing the scientific potential of high school" 2.1.1/1775.

References

[1] J. Alber, J. Gramm, J. Guo, R. Niedermeier, Computing of two sequences with nested arc notations, *Theoret. Comput. Sci.* **312,** 2-3 (2004) 337–358. ⇒36

[2] V. Bafna, S. Muthukrishnan, R. Ravi, Comparing similarity between RNA strings, *Proc. 6th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Comput. Sci.* **937** (1995) 1–16. ⇒36

[3] G. Blin, H. Touzet, How to compare arc-annotated sequences: The alignment hierarchy, *Proc. 13th International Symposium on String Processing and Information Retrieval (SPIRE), Lecture Notes in Comput. Sci.* **4209** (2006) 291–303. ⇒36

[4] G. Blin, M. Crochemore, S. Vialette, Algorithmic aspects of arc-annotated sequences, in: *Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications* (ed. M. Elloumi, A. Y. Zomaya), John Wiley & Sons, Inc., Hoboken, NJ, 2011, pp. 171–183. ⇒36

[5] H. L. Bodlaender, R. G. Downey, M. R. Fellows, H. T. Wareham, The parameterized complexity of sequence alignment and consensus, *Theoret. Comput. Sci.* **147,** 1-2 (1995) 31–54. ⇒35, 36

[6] H. L. Bodlaender, R. G. Downey, M. R. Fellows, M. T. Hallett, H. T. Wareham, Parameterized complexity analysis in computational biology, *Computer Applications in the Biosciences* **11,** 1 (1995) 49–57. ⇒36

[7] J. Chen, X. Huang, I. A. Kanj, G. Xia, W-hardness under linear FPT-reductions: structural properties and further applications, *Proc. of COCOON*, Kunming, China, 2005, pp. 975–984. ⇒36

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Third edition, The MIT Press, Cambridge, Massachusetts, 2009. ⇒36

[9] F. Corpet, B. Michot, Rnalign program: alignment of RNA sequences using both primary and secondary structures, *Computer Applications in the Biosciences* **10,** 4 (1994) 389–399. ⇒36

[10] P. Damaschke, A remark on the subsequence problem for arc-annotated sequences with pairwise nested arcs, *Inform. Process. Lett.* **100,** 2 (2006) 64–68. ⇒36

[11] W. H. E. Day, F. R. McMorris, Discovering consensus molecular sequences, in: *Information and Classification – Concepts, Methods, and Applications* (ed. O. Opitz, B. Lausen, R. Klar), Springer-Verlag, Berlin, 1993, pp. 393–402. ⇒36

[12] W. H. E. Day, F. R. McMorris, The computation of consensus patterns in DNA sequences, *Math. Comput. Modelling* **17,** 10 (1993) 49–52. ⇒36

[13] P. A. Evans, *Algorithms and Complexity for Annotated Sequence Analysis*, PhD Thesis, University of Victoria, Victoria, 1999. ⇒36, 37

[14] P. A. Evans, Finding common subsequences with arcs and pseudoknots, *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, *Lecture Notes in Comput. Sci.* **1645** (1999) 270–280. ⇒36

[15] J. Gramm, J. Guo, R. Niedermeier, Pattern matching for arc-annotated sequences, *ACM Trans. Algorithms* **2,** 1 (2006) 44–65. ⇒36

[16] J. Guo, *Exact algorithms for the longest common subsequence problem for arc-annotated sequences*, Master Thesis, Eberhard-Karls-Universität, Tübingen, 2002. ⇒36, 43

[17] D. Gusfield, *Algorithm on Strings, Trees, and Sequences: Computer Science and Computational Biology,* Cambridge University Press, Cambridge, 1997. ⇒38

[18] D. S. Hirschberg, *The Longest Common Subsequence Problem*, PhD Thesis, Princeton University, Princeton, 1975. ⇒36

[19] D. S. Hirschberg, Recent results on the complexity of common subsequence problems, in: *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison* (ed. D. Sankoff, J. B. Kruskal), Addison-Wesley Publishing Company, Reading/Menlo Park, NY, 1983, pp. 325–330. ⇒35

[20] C. S. Iliopouéos, M. S. Rahman, Algorithms for computing variants of the longest common subsequence problem, *Theoret. Comput. Sci.* **395,** 2-3 (2008) 255–267. ⇒36

[21] R. W. Irving, C. B. Fraser, Two algorithms for the longest common subsequence of three (or more) strings, *Proc. Third Annual Symposium on Combinatorial Pattern Matching*, *Lecture Notes in Comput. Sci.* **644** (1992) 214–229. ⇒35, 36

[22] T. Jiang, G.-H. Lin, B. Ma, K. Zhang, The longest common subsequence problem for arc-annotated sequences, *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, *Lecture Notes in Comput. Sci.* **1848** (2000) 154–165. ⇒36

[23] T. Jiang, G.-H. Lin, B. Ma, K. Zhang, The longest common subsequence problem for arc-annotated sequences, *J. Discrete Algorithms* **2,** 2 (2004) 257–270. ⇒36

[24] M. Li, B. Ma, L. Wang, Near optimal multiple alignment within a band in polynomial time, *Proc. Thirty-second Annual ACM Symposium on Theory of Computing (STOC'00)*, Portland, OR, 2000, pp. 425–434. ⇒38

[25] G. H. Lin, Z. Z. Chen, T. Jiang, J. J. Wen, The longest common subsequence problem for sequences with nested arc annotations, *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Comput. Sci.* **2076** (2001) 444–455. ⇒36, 38, 43

[26] S. Y. Lu, K. S. Fu, A sentence-to-sentence clustering procedure for pattern analysis, *IEEE Transactions on Systems, Man, and Cybernetics* **8,** 5 (1978) 381–389. ⇒36

[27] D. Maier, The complexity of some problems on subsequences and supersequences, *J. ACM* **25,** 2 (1978) 322–336. ⇒36

[28] D. Marx, I. Schlotter, Parameterized complexity of the arc-preserving subsequence problem, *Proc. 36th International Workshop on Graph Theoretic Concepts in Computer Science (WG 2010)*, *Lecture Notes in Comput. Sci.* **6410** (2010) 244–255. ⇒36

[29] A. Ouangraoua, C. Chauve, V. Guignon, S. Hamel, New algorithms for aligning nested arc-annotated sequences, Laboratoire Bordelais de Recherche en Informatique, Research Report RR-1443-08, Université Bordeaux, 2008. ⇒36

[30] A. Ouangraoua, V. Guignon, S. Hamel, C. Chauve, A new algorithm for aligning nested arc-annotated sequences under arbitrary weight schemes, *Theoret. Comput. Sci.* **412,** 8-10 (2011) 753–764. ⇒36

[31] C. H. Papadimitriou, *Computational complexity*, Addison-Wesley Publishing Company, Reading/Menlo Park, NY, 1994. ⇒39

[32] P. A. Pevzner, Multiple alignment, communication cost, and graph matching, *SIAM J. Appl. Math.* **52,** 6 (1992) 1763–1779. ⇒36

[33] K. Pietrzak, On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems, *J. Comput. System Sci.* **67,** 4 (2003) 757–771. ⇒36

[34] D. Sankoff, Matching comparisons under deletion/insertion constraints, *Proc. Natl. Acad. Sci. USA* **69,** 1 (1972) 4–6. ⇒36

[35] R. A. Wagner, M. J. Fischer, The string-to-string correction problem, *J. ACM* **21,** 1 (1974) 168–173. ⇒36

Received: November 17, 2010• Revised: March 11, 2011

Acta Univ. Sapientiae, Informatica, **3,** 1 (2011) 76–98

Implementing a non-strict purely functional language in JavaScript

László DOMOSZLAI
Eötvös Loránd University, Budapest, Hungary
Radboud University Nijmegen, the Netherlands
email: dlacko@gmail.com

Eddy BRUËL
Vrije Universiteit Amsterdam
the Netherlands
email: ejpbruel@gmail.com

Jan Martin JANSEN
Faculty of Military Sciences
Netherlands Defence Academy
Den Helder, the Netherlands
email: jm.jansen.04@nlda.nl

Abstract. This paper describes an implementation of a non-strict purely functional language in JavaScript. This particular implementation is based on the translation of a high-level functional language such as Haskell or Clean into JavaScript via the intermediate functional language Sapl. The resulting code relies on the use of an evaluator function to emulate the non-strict semantics of these languages. The speed of execution is competitive with that of the original Sapl interpreter itself and better than that of other existing interpreters.

1 Introduction

Client-side processing for web applications has become an important research subject. Non-strict purely functional languages such as Haskell and Clean have many interesting properties, but their use in client-side processing has been limited so far. This is at least partly due to the lack of browser support for these languages. Therefore, the availability of an implementation for non-strict

Computing Classification System 1998: D.1.1
Mathematics Subject Classification 2010: 68N18
Key words and phrases: web programming, functional programming, Sapl, JavaScript, Clean

purely functional languages in the browser has the potential to significantly improve the applicability of these languages in this area.

Several implementations of non-strict purely functional languages in the browser already exist. However, these implementations are either based on the use of a Java Applet (e.g. for Sapl, a client-side platform for Clean [8, 14]) or a dedicated plug-in (e.g. for HaskellScript [11] a Haskell-like functional language). Both these solutions require the installation of a plug-in, which is often infeasible in environments where the user has no control over the configuration of his/her system.

1.1 Why switch to JavaScript?

As an alternative solution, one might consider the use of JavaScript. A JavaScript interpreter is shipped with every major browser, so that the installation of a plug-in would no longer be required. Although traditionally perceived as being slower than languages such as Java and C, the introduction of JIT compilers for JavaScript has changed this picture significantly. Modern implementations of JavaScript, such as the V8 engine that is shipped with the Google Chrome browser, offer performance that sometimes rivals that of Java.

As an additional advantage, browsers that support JavaScript usually also expose their HTML DOM through a JavaScript API. This allows for the association of JavaScript functions to HTML elements through the use of event listeners, and the use of JavaScript functions to manipulate these same elements.

This notwithstanding, the use of multiple formalisms complicates the development of Internet applications considerably, due to the close collaboration required between the client and server parts of most web applications.

1.2 Results at a glance

We implemented a compiler that translates Sapl to JavaScript expressions. Its implementation is based on the representation of unevaluated expressions (thunks) as JavaScript arrays, and the just-in-time evaluation of these thunks by a dedicated evaluation function (different form the `eval` function provided by JavaScript itself).

Our final results show that it is indeed possible to realize this translation scheme in such a way that the resulting code runs at a speed competitive with that of the original Sapl interpreter itself. Summarizing, we obtained the following results:

- We realized an implementation of the non-strict purely functional programming language Clean in the browser, via the intermediate language Sapl, that does not require the installation of a plug-in.
- The performance of this implementation is competitive with that of the original Sapl interpreter and faster than that of many other interpreters for non-strict purely functional languages.
- The underlying translation scheme is straightforward, constituting a one-to-one mapping of Sapl onto JavaScript functions and expressions.
- The implementation of the compiler is based on the representation of unevaluated expressions as JavaScript arrays and the just-in-time evaluation of these thunks by a dedicated evaluation function.
- The generated code is compatible with JavaScript in the sense that the namespace for functions is shared with that of JavaScript. This allows generated code to interact with JavaScript libraries.

1.3 Organization of the paper

The structure of the remainder of this paper is as follows: we start with introducing Sapl, the intermediate language we intend to implement in JavaScript in Section 2. The translation scheme underlying this implementation is presented in Section 3. We present the translation scheme used by our compiler in two steps. In step one, we describe a straightforward translation of Sapl to JavaScript expressions. In step two, we add several optimizations to the translation scheme described in step one. Section 4 presents a number of benchmark tests for the implementation. A number of potential applications is presented in Section 5. Section 6 compares our approach with that of others. Finally, we end with our conclusions and a summary of planned future work in Section 7.

2 The Sapl programming language and interpreter

Sapl stands for **S**imple **A**pplication **P**rogramming **L**anguage. The original version of Sapl provided no special constructs for algebraic data types. Instead, they are represented as ordinary functions. Details on this encoding and its consequences can be found in [8]. Later a Clean like type definition style was adopted for readability and to allow for the generation of more efficient code (as will become apparent in Section 3).

The syntax of the language is the following:

$\langle program \rangle$ ::= {$\langle function \rangle$ | $\langle type \rangle$}+
$\langle type \rangle$ ::= '::' $\langle ident \rangle$ '=' $\langle ident \rangle$ $\langle ident \rangle$* {'|' $\langle ident \rangle$ $\langle ident \rangle$*}*
$\langle function \rangle$ ::= $\langle ident \rangle$ $\langle ident \rangle$* '=' $\langle let\text{-}expr \rangle$
$\langle let\text{-}expr \rangle$ ::= ['let' $\langle let\text{-}defs \rangle$ 'in'] $\langle main\text{-}expr \rangle$
$\langle let\text{-}defs \rangle$ ::= $\langle ident \rangle$ '=' $\langle application \rangle$ {',' $\langle ident \rangle$ '=' $\langle application \rangle$}*
$\langle main\text{-}expr \rangle$ ::= $\langle select\text{-}expr \rangle$ | $\langle if\text{-}expr \rangle$ | $\langle application \rangle$
$\langle select\text{-}expr \rangle$ ::= 'select' $\langle factor \rangle$ {'(' {$\langle lambda\text{-}expr \rangle$ | $\langle let\text{-}expr \rangle$} ')'}+
$\langle if\text{-}expr \rangle$ ::= 'if' $\langle factor \rangle$ '(' $\langle let\text{-}expr \rangle$ ')' '(' $\langle let\text{-}expr \rangle$ ')'
$\langle lambda\text{-}expr \rangle$ ::= '\' $\langle ident \rangle$+ '=' $\langle let\text{-}expr \rangle$
$\langle application \rangle$ ::= $\langle factor \rangle$ $\langle factor \rangle$*
$\langle factor \rangle$ ::= $\langle ident \rangle$ | $\langle literal \rangle$ | '(' $\langle application \rangle$ ')'

An identifier can be any identifier accepted by Clean, including operator notations. For literals characters, strings, integer or floating-point numbers and boolean values are accepted.

We illustrate the use of Sapl by giving a number of examples. We start with the encoding of the list data type, together with the `sum` function.

```
:: List = Nil | Cons x xs
sum xxs = select xxs 0 (λx xs = x + sum xs)
```

The `select` keyword is used to make a case analysis on the data type of the variable `xxs`. The remaining arguments handle the different constructor cases in the same order as they occur in the type definition (all cases must be handled separately). Each case is a function that is applied to the arguments of the corresponding constructor.

As a more complex example, consider the `mappair` function written in Clean, which is based on the use of pattern matching:

```
mappair f Nil          zs          = Nil
mappair f (Cons x xs)  Nil         = Nil
mappair f (Cons x xs)  (Cons y ys) = Cons (f x y) (mappair f xs ys)
```

This definition is transformed to the following Sapl function (using the above definitions for `Nil` and `Cons`).

```
mappair f as zs
  = select as Nil (λx xs = select zs Nil (λy ys = Cons (f x y) (mappair f xs ys)))
```

Sapl is used as an intermediate formalism for the interpretation of non-strict purely functional programming languages such as Haskell and Clean. The Clean compiler includes a Sapl back-end that generates Sapl code. Recently, the Clean compiler has been extended to be able to compile Haskell programs as well [5].

2.1 Some remarks on the definition of **Sapl**

Sapl is very similar to the core languages of Haskell and Clean. Therefore, we choose not to give a full definition of its semantics. Rather, we only say something about its main characteristics and give a few examples to illustrate these.

The only keywords in Sapl are `let`, `in`, `if` and `select`. Only constant (non-function) `let` expressions are allowed that may be mutually recursive (for creating cyclic expressions). They may occur at the top level in a function and at the top level in arguments of an `if` and `select`. λ-expressions may only occur as arguments to a `select`. If a Clean program contains nested λ-expressions, and you compile it to Sapl, they should be lifted to the top-level.

3 A **JavaScript** based implementation for **Sapl**

Section 1 motivated the choice for implementing a Sapl interpreter in the browser using JavaScript. Our goal was to make the implementation as efficient as possible.

Compared to Java, JavaScript provides several features that offer opportunities for a more efficient implementation. First of all, the fact that JavaScript is a *dynamic* language allows both functions and function calls to be generated at run-time, using the built-in functions `eval` and `apply`, respectively. Second, the fact that JavaScript is a dynamically *typed* language allows the creation of heterogeneous arrays. Therefore, rather than building an interpreter, we have chosen to build a compiler/interpreter hybrid that exploits the features mentioned above.

Besides these, the evaluation procedure is heavily based on the use of the `typeof` operator and the runtime determination of the number of formal parameters of a function which is another example of the dynamic properties of the JavaScript language.

For the following Sapl constructs we must describe how they are translated to JavaScript:

- literals, such as booleans, integers, real numbers, and strings;
- identifiers, such as variable and function names;
- function definitions;
- constructor definitions;
- let constructs;
- applications;

- select statements;
- if statements;
- built-in functions, such as `add`, `eq`, etc.

Literals Literals do not have to be transformed. They have the same representation in Sapl and JavaScript.

Identifiers Identifiers in Sapl and JavaScript share the same namespace, therefore, they need not to be transformed either.

However, the absence of block scope in JavaScript can cause problems. The scope of variables declared using the `var` keyword is hoisted to the entire containing function. This affects the `let` construct and the λ-expressions, but can be easily avoided by postfixing the declared identifiers to be unique. In this way, the original variable name can be restored if needed.

With this remark we will neglect these transformations in the examples of this paper for the sake of readability.

Function definitions Due to JavaScript's support for higher-order functions, function definitions can be translated from Sapl to JavaScript in a straightforward manner:

```
T⟦f x1 ... xn = body⟧  =  function f(x1, ..., xn) { T⟦body⟧ }
```

So Sapl functions are mapped one-to-one to JavaScript functions with the same name and the same number of arguments.

Constructor definitions Constructor definitions in Sapl are translated to arrays in JavaScript, in such a way that they can be used in a `select` construct to select the right case. A Sapl type definition containing constructors is translated as follows:

```
T⟦:: typename = ... | Ck xk0 ... xkn | ...⟧
   = ... function Ck(xk0, ..., xkn) { return [k, 'Ck', xk0, ..., xkn]; } ...
```

where `k` is a positive integer, corresponding to the position of the constructor in the original type definition. The name of the constructor, `'Ck'`, is put into the result for printing purposes only. This representation of the constructors together with the use of the `select` statement allows for a very efficient JavaScript translation of the Sapl language.

Let constructs `Let` constructs are translated differently depending on whether they are cyclic or not. Non-cyclic lets in Sapl can be translated to `var` declarations in JavaScript, as follows:

```
T⟦let x = e in b⟧  =  var x = T⟦e⟧; T⟦b⟧
```

Due to JavaScript's support for closures, cyclic lets can be translated from Sapl to JavaScript in a straightforward manner. The idea is to take any occurrences of `x` in `e` and replace them with:

```
function () { return x; }
```

This construction relies on the fact that the scope of a JavaScript closure is the whole function itself. This means that after the declaration the call of this closure will return a valid reference. In Section 3.1 we present an example to illustrate this.

Applications Every Sapl expression is an application. Due to JavaScript's eager evaluation semantics, applications cannot be translated from Sapl to JavaScript directly. Instead, unevaluated expressions (or *thunks*) in Sapl are translated to arrays in JavaScript:

```
T⟦x0 x1 .. xn⟧  =  [T⟦x0⟧, [T⟦x1⟧, ..., T⟦xn⟧]]
```

Thus, a thunk is represented with an array of two elements. The first one is the function involved, and the second one is an array of the arguments. This second array is used for performance reasons. In this way one can take advantage of the JavaScript `apply()` method and it is very straightforward and fast to join such two arrays, which is necessary to do during evaluation.

select statements A `select` statement in Sapl is translated to a `switch` statement in JavaScript, as follows:

```
T⟦select f (\x0 ... xn = b) ...⟧

=

var _tmp = Sapl.feval(T⟦f⟧);
switch(_tmp[0]) {
    case 0: var x0 = _tmp[2], ..., xn = _tmp[n+2];
            T⟦b⟧;
            break;
    ...
};
```

Evaluating the first argument of a `select` statement yields an array representing a constructor (see above). The first argument in this array represents the position of the constructor in its type definition, and is used to select the right case in the definition. The parameters of the λ- expression for each case are bound to the corresponding arguments of the constructor in the `var` declaration (see also examples).

if statements An `if` statement in Sapl is translated to an `if` statement in JavaScript straightforwardly:

$T[\![\mathbf{if}\ \mathtt{p\ t\ f}]\!] = \mathbf{if}\ (\mathtt{Sapl.feval}(T[\![p]\!]))\{\ T[\![t]\!];\ \}\ \mathtt{else}\ \{\ T[\![f]\!];\ \}$

This translation works because booleans in Sapl and JavaScript have the same representation.

Built-in functions Sapl defines several built-in functions for arithmetic and logical operations. As an example, the `add` function is defined as follows:

```
function add(x, y) { return Sapl.feval(x) + Sapl.feval(y); }
```

Unlike user-defined functions, a built-in function such as `add` has strict evaluation semantics. To guarantee that they are in normal form when the function is called, the function `Sapl.feval` is applied to its arguments (see Section 3.2).

3.1 Examples

The following definitions in Sapl:

```
:: List = Nil | Cons x xs

ones = let os = Cons 1 os in os
fac n = if (eq n 0) 1 (mult n (fac (sub n 1)))
sum xxs = select xxs 0 (λx xs = add x (sum xs))
```

are translated to the following definitions in JavaScript:

```
function Nil() { return [0, 'Nil']; }
function Cons(x, xs) { return [1, 'Cons', x, xs]; }

function ones() { var os = Cons(1, function() { return os; }); return os; }

function fac(n) {
    if (Sapl.feval(n) == 0) {
        return 1;
    } else {
        return [mult, [n, [fac, [[sub, [n, 1]]]]]];
    }
}
```

```
function sum(as) {
    var _tmp = Sapl.feval(as);
    switch (_tmp[0]) {
        case 0: return 0;
        case 1: var x = _tmp[2], xs = _tmp[3];
                return [add, [x, [sum, [xs]]]];
    }
}
```

The examples show that the translation is straightforward and preserves the structure of the original definitions.

3.2 The `feval` function

To emulate Sapl's non-strict evaluation semantics for function applications, we represented unevaluated expressions (thunks) as arrays in JavaScript. Because JavaScript treats these arrays as primitive values, some way is needed to explicitly reduce thunks to normal form when their value is required. This is the purpose of the `Sapl.feval` function. It reduces expressions to weak head normal form. Further evaluation of expressions is done by the printing routine. `Sapl.feval` performs a case analysis on an expression and undertakes different actions based on its type:

Literals If the expression is a literal or a constructor, it is returned immediately. Literals and constructors are already in normal form.

Thunks If the expression is a thunk of the form `[f, [xs]]`, it is transformed into a function call `f(xs)` with the JavaScript `apply` function, and `Sapl.feval` is applied recursively to the result (this is necessary because the result of a function call may be another thunk).

Due to JavaScript's reference semantics for arrays, thunks may become shared between expressions over the course of evaluation. To prevent the same thunk from being reduced twice, the result of the call is written back into the array. If this result is a primitive value, the array is transformed into a *boxed value* instead. Boxed values are represented as arrays of size one. Note that in JavaScript, the size of an array can be altered in-place.

If the number of arguments in the thunk is smaller than the arity of the function, it cannot be further reduced (is already in normal form), so it is returned immediately. Conversely, if the number of arguments in the thunk is larger than the arity of the function, a new thunk is constructed from the result of the call and the remainder of the arguments, and `Sapl.feval` is applied iteratively to the result.

Boxed values If the expression is a boxed value of the form `[x]`, the value `x` is unboxed and returned immediately (only literals and constructors can be boxed).

Curried applications If the expression is a curried application of the form `[[f, [xs]], [ys]]`, it is transformed into `[f, [xs ++ ys]]`, and `Sapl.feval` is applied iteratively to the result.

More details on evaluation For the sake of deeper understanding we also give the full source code of `feval`:

```
feval = function (expr) {
  var y, f, xs;
  while (1) {
      if (typeof(expr) == "object") {                // closure
          if (expr.length == 1) return expr[0];      // boxed value
          else if (typeof(expr[0]) == "function") { // application -> make call
              f = expr[0];  xs = expr[1];
              if (f.length == xs.length) {           // most often occurring case
                  y = f.apply(null, xs);             // turn chunk into call
                  expr[0] = y;                       // overwrite for sharing!
                  expr.length = 1;                   // adapt size
              } else if (f.length < xs.length) {     // less likely case
                  y = f.apply(null,xs.splice(0, f.length));
                  expr[0] = y;                       // slice of arguments
              } else
                  return expr;                       // not enough arguments
          } else if (typeof(expr[0])=="object") {    // curried app -> uncurry
              y = expr[0];
              expr[0] = y[0];
              expr[1] = y[1].concat(expr[1]);
          } else
              return expr;                           // constructor
      } else if (typeof(expr) == "function")         // function
          expr = [expr, []];
        else                                         // literal
          return expr;
  }
}
```

3.3 Further optimizations

Above we described a straightforward compilation scheme from Sapl to JavaScript, where unevaluated expressions (thunks) are translated to arrays.

The `Sapl.feval` function is used to reduce thunks to normal form when their value is required. For ordinary function calls, our measurements indicate that the use of `Sapl.feval` is more than 10 times slower than doing the same call directly. This constitutes a significant overhead. Fortunately, a simple compile time analysis reveals many opportunities to eliminate unnecessary thunks in favor of such direct calls. Thus, expressions of the form:

```
Sapl.feval([f, [x1, ..., xn])
```

are replaced by:

```
f(x1, ..., xn)
```

This substitution is only possible if `f` is a function with known arity at compile-time, and the number of arguments in the thunk is equal to the arity of the function. It can be performed wherever a call to Sapl.feval occurs:

- The first argument to a `select` or `if`;
- The arguments to a built-in function;
- Thunks that follow a `return` statement in JavaScript. These expressions are always evaluated immediately after they are returned.

As an additional optimization, arithmetic operations are inlined wherever they occur. With these optimizations added, the earlier definitions of `sum` and `fac` are now translated to:

```
function fac(n) {
    if (Sapl.feval(n) == 0) {
        return 1;
    } else {
        return Sapl.feval(n) * fac(Sapl.feval(n) - 1);
    }
}

function sum(xxs) {
    var _tmp = Sapl.feval(xxs);
    switch(_tmp[0]){
        case 0: return 0;
        case 1: var x = _tmp[2], xxs = _tmp[3];
                return Sapl.feval(x) + sum(xs);
    }
}
```

Moreover, let's consider the following definition of the Fibonacci function, `fib`, in Sapl:

```
fib n = if (gt 2 n) 1 (add (fib (sub n 1)) (fib (sub n 2)))
```

This is translated to the following function in JavaScript:

```
function fib(n) {
    if (2 > Sapl.feval(n)) {
        return 1;
    } else {
        return (fib([sub, [n, 1]]) + fib([sub, [n, 2]]));
    }
}
```

A simple strictness analysis reveals that this definition can be turned into:

```
function fib(n) {
    if (2 > n) {
        return 1;
    } else {
        return (fib(n - 1) + fib(n - 2));
    }
}
```

The calls to `feval` are now gone, which results in a huge improvement in performance. Indeed, this is how `fib` would have been written, had it been defined in JavaScript directly. In this particular example, the use of eager evaluation did not affect the semantics of the function. However, this is not true in general. For the use of such an optimization we adopted a Clean like strictness annotation. Thus, the above code can be generated from the following Sapl definition:

```
fib !n = if (gt 2 n) 1 (add (fib (sub n 1)) (fib (sub n 2)))
```

But strictly defined arguments also have their price. In case one does not know if an argument in a function call is already in evaluated form, an additional wrapper function call is needed that has as only task to evaluate the strict arguments:

```
function fib$eval(a0) {
    return fib(Sapl.feval(a0));
}
```

As a possible further improvement, a more thorough static analysis on the propagation of strict arguments could help to avoid some of these wrapper calls.

Finally, the Sapl to JavaScript compiler provides simple tail recursion optimization, which has impact on not only the execution time, but also reduces stack use.

The optimizations only affect the generated code and not the implementation of `feval`. In the next section an indication of the speed-up obtained by the optimizations is given.

4 Benchmarks

In this section we present the results of several benchmark tests for the JavaScript implementation of Sapl (which we will call Sapljs) and a comparison with the Java Applet implementation of Sapl. We ran the benchmarks on a MacBook 2.26 MHz Core 2 Duo machine running MacOS X10.6.4. We used Google Chrome with the V8 JavaScript engine to run the programs. At this moment V8 offers one of the fastest platforms for running Sapljs programs. However, there is a heavy competition on JavaScript engines and they tend to become much faster. The benchmark programs we used for the comparison are the same as the benchmarks we used for comparing Sapl with other interpreters and compilers in [8]. In that comparison it turned out that Sapl is at least twice as fast (and often even faster) as other interpreters like Helium, Amanda, GHCi and Hugs. Here we used the Java Applet version for the comparison. This version is about 40% slower than the C version of the interpreter described in [8] (varying from 25 to 50% between benchmarks), but is still faster than the other interpreters mentioned above. The Java Applet and JavaScript version of Sapl and all benchmark code can be found at [2]. We briefly repeat the description of the benchmark programs here:

1. **Prime Sieve** The prime number sieve program, calculating the 2000th prime number.
2. **Symbolic Primes** Symbolic prime number sieve using Peano numbers, calculating the 160th prime number.
3. **Interpreter** A small Sapl interpreter. As an example we coded the prime number sieve for this interpreter and calculated the 30th prime number.
4. **Fibonacci** The (naive) Fibonacci function, calculating `fib 35`.
5. **Match** Nested pattern matching (5 levels deep) repeated 160000 times.
6. **Hamming** The generation of the list of Hamming numbers (a cyclic definition) and taking the 1000th Hamming number, repeated 1000 times.
7. **Sorting** Tree Sort (3000 elements), Insertion Sort (3000 elements), Quick Sort (3000 elements), Merge Sort (10000 elements, merge sort is much faster, we therefore use a larger example)
8. **Queens** Number of placements of 11 Queens on a 11 x 11 chess board.

	Pri	Sym	Inter	Fib	Match	Ham	Qns	Kns	Sort	Plog	Parse
Sapl	1200	4100	500	8700	1700	2500	9000	3200	1700	1500	1100
Sapljs	2200	4000	220	280	2200	3700	11500	3950	2450	2750	4150
Sapljs nopt	4500	11000	1500	36000	6700	5500	36000	11000	4000	5200	6850
perc. mem.	58	68	38	0	21	31	37	35	45	53	41

Figure 1: Speed comparison (time in miliseconds).

9. **Knights** Finding a Knights tour on a 5 x 5 chess board.
10. **Prolog** A small Prolog interpreter based on unification only (no arithmetic operations), calculating ancestors in a four generation family tree, repeated 100 times.
11. **Parser Combinators** A parser for Prolog programs based on Parser Combinators parsing a 3500 lines Prolog program.

For sorting a list of size n a source list is used consisting of numbers 1 to n. The elements that are 0 modulo 10 are put before those that are 1 modulo 10, etc.

The benchmarks cover a wide range of aspects of functional programming: lists, laziness, deep recursion, higher order functions, cyclic definitions, pattern matching, heavy calculations, heavy memory usage. The programs were chosen to run at least for a second, if possible. This helps eliminating start-up effects and gives the JIT compiler enough time to do its work. In many cases the output was converted to a single number (e.g. by summing the elements of a list) to eliminate the influence of slow output routines.

4.1 Benchmark tests

We ran the tests for the following versions of Sapl:

- Sapl: the Java Applet version of Sapl;
- Sapljs: the Sapljs version including the normal form optimization, the inlining of arithmetic operations and the tail recursion optimization. The strictness optimization is only used for the fib benchmark;
- Sapljs nopt: the version not using these optimizations.

We also included the estimated percentage of time spent on memory management for the Sapljs version. The results can be found in Figure 1.

4.2 Evaluation of the benchmark tests

Before analysing the results we first make some general remarks about the performance of Java, JavaScript and the Sapl interpreter which are relevant for a better understanding of the results. In general it is difficult to give absolute figures when comparing the speeds of language implementations. They often also depend on the platform (processor), the operating system running on it and the particular benchmarks used to compare. Therefore, all numbers given should be interpreted as global indications.

According to the language shoot-out site [3] Java programs run between 3 and 5 times faster than similar JavaScript programs running on V8. So a reimplementation of the Sapl interpreter in JavaScript is expected to run much slower as the Sapl interpreter.

We could not run all benchmarks as long as we wished because of stack limitations for V8 JavaScript in Google Chrome. It supports a standard (not user modifiable) stack of only 30k at this moment. This is certainly enough for most JavaScript programs, but not for a number of our benchmarks that can be deeply recursive. This limited the size of the runs of the following benchmarks: Interpreter[1] all sorting benchmarks, and the Prolog and Parser Combinator benchmark. Another benchmark that we used previously, and that could not be ran at all in Sapljs is: `twice twice twice twice inc 0`.

For a lazy functional language the creation of thunks and the re-collection of them later on, often takes a substantial part of program run-times. It is therefore important to do some special tests that say something about the speed of memory (de-)allocation. The Sapl interpreter uses a dedicated memory management unit (see [8]) not depending on Java memory management. The better performance of the Sapl interpreter in comparison with the other interpreters partly depends on its fast memory management. For the JavaScript implementation we rely on the memory management of JavaScript itself. We did some dedicated tests that showed that memory allocation for the Java Sapl interpreter is about 5-7 times faster than the JavaScript implementation. Therefore, we included an estimation of the percentage of time spent on memory management for all benchmarks ran in Sapljs. The estimation was done by counting all memory allocations for a benchmark (all creations of thunks) and multiplying it with an estimation of the time to create a thunk, which was measured by a special application that only creates thunks.

[1]The latest version of Chrome has an even more restricted stack size. We can now run Interpreter only up to the 18th prime number.

Results The Fibonacci and Interpreter benchmarks run (30 and 2 times resp.) significantly faster in Sapljs than in the Sapl interpreter. Note that both these benchmarks profit significantly from the optimizations with Fibonacci being more than 100 times faster and Interpreter almost 7 times faster than the non-optimized version. The addition of the strictness annotation for Fibonacci contributes a factor of 3 to the speed-up. With this annotation the compiled Fibonacci program is equivalent to a direct implementation of Fibonacci in JavaScript and does not use feval anymore. The original Sapl interpreter does not apply any of these optimizations. The Interpreter benchmark profits much (almost a factor of 2) from the tail recursion optimization that applies for a number of often used functions that dominate the performance of this benchmark.

Symbolic Primes, Match, Queens and Knights run at a speed comparable to the Sapl interpreter. Hamming and Sort are 40 percent slower, Primes and Prolog are 80 percent slower. Parser Combinators is the worst performing benchmark and is almost 4 times slower than in Sapl.

All benchmarks benefit considerably from the optimizations (between 1.5 and 120 times faster), with Fibonacci as the most exceptional.

The Parser Combinators benchmark profits only modestly from the optimizations and spends relatively much time in memory management operations. It is also the most ‘higher order’ benchmark of all. Note that for the original Sapl interpreter this is one of the best performing benchmarks (see [8]), performing at a speed that is even competitive with compiler implementations. The original Sapl interpreter does an exceptionally good job on higher order functions.

We conclude that the Sapljs implementation offers a performance that is competitive with that of the Sapl interpreter and therefore with other interpreters for lazy functional programming languages.

Previously [8] we also compared Sapl with the GHC and Clean compilers. It was shown that the C version of the Sapl interpreter is about 3 times slower than GHC without optimizer. Extrapolating this result using the figures mentioned above we conclude that Sapljs is about 6-7 times slower than GHC (without optimizer). In this comparison we should also take into account that JavaScript applications run at least 5 times slower than comparable C applications. The remaining difference can be mainly attributed to the high price for memory operations in Sapljs.

4.3 Alternative memory management?

For many Sapljs examples a substantial part of their run-time is spent on memory management. They can only run significantly faster after a more efficient memory management is realized or after other optimizations are realized. It is tempting to implement a memory management similar to that of the Sapl interpreter. But this memory management relies heavily on representing graphs by binary trees, which does not fit with our model for turning thunks into JavaScript function calls which depends heavily on using arrays to represent thunks.

5 Applications

Developing rich client-side applications in Clean We can use the Sapljs compiler to create dedicated client-side applications in Clean that make use of JavaScript libraries. We can do this because JavaScript and code generated by Sapljs share the same namespace. In this way it is possible to call functions within Sapl programs that are implemented in JavaScript. The Sapljs compiler doesn't check the availability of a function, so one has to rely on the JavaScript interpreter to do this. Examples of such functions are the built-in core functions like `add` and `eq`, but they can be any application related predefined function.

Because we have to compile from Clean to Sapl before compiling to JavaScript, we need a way to use functions implemented in JavaScript within Clean programs. Clean does not allow that programs contain unknown functions, so we need a way to make these functions known to the Clean compiler. This can be realized in the following way. If one wants to postpone the implementation of a function to a later time, one can define its type and define its body to be `undef`. E.g., `example` is a function with 2 integer arguments and an integer result with an implementation only in JavaScript.

```
example :: Int Int → Int
example = undef
```

The function `undef` is defined in the StdMisc module. An `undef` expressions matches every type, so we can use this definition to check if the written code is syntactically and type correct. We adapted the Clean to Sapl compiler not to generate code for functions with an undefined body. In this way we have created a universal method to reference functions defined outside the Clean environment.

We used these techniques to define a library in Clean for manipulating the HTML DOM at the client side. The following Clean code gives a demonstration of its use:

```
import StdEnv, SaplHtml

onKeyUp :: !HtmlEvent !*HtmlDocument → *(*HtmlDocument, Bool)
onKeyUp e d
    # (d, str) = getDomAttr d "textarea" "value"
    # (d, str) = setDomAttr d "counter" "innerHTML" (toString (size str))
    = (d, True)

Start
    = toString (Div [] [] [TextArea [Id "textarea", Rows 15, Cols 50]
                                    [OnKeyUp onKeyUp],
                           Div [Id "counter"] [] []])
```

It is basically a definition of a piece of HTML using arrays and ADTs defined in the `SaplHtml` module. What is worth to notice here are the definitions of the event handler function and the DOM manipulating functions, `getDomAttr` and `setDomAttr`, which are also defined in `SaplHtml`, but are implemented in JavaScript using the above mentioned techniques. The two parameters of the event handler function are effectively the related JavaScript `Event` and `Document` objects, respectively.

Compiling the program to JavaScript and running it returns the following string, which is legal HTML:

```
<div><textarea id="textarea"
          rows="15"
          cols="50"
          onKeyUp="Sapl.execEvent(event, 'onKeyUp$eval')">
     </textarea>
     <div id="counter"></div>
</div>
```

The event handler call is wrapped by the `Sapl.execEvent` function which is responsible for passing the event related parameters to the actual event handler. Including this string into an HTML document along with the generated JavaScript functions we get a client side web application originally written in Clean. Despite this program is deliberately very simple, it demonstrates almost all the basics necessary to write any client side application. Additional interface functions, e.g. calling methods of a JavaScript object, can be found in the `SaplHtml` module.

iTask integration Another possible application is related to the iTask system [13]. iTask is a combinator library written in Clean, and is used for the realization of web-based dynamic workflow systems. An iTask application consists of a structured collection of tasks to be performed by users, computers or both.

To enhance the performance of iTask applications, the possibility to handle tasks on the client was added [14], accomplished by the addition of a simple `OnClient` annotation to a task. When this annotation is present, the iTask runtime automatically takes care of all communication between the client and server parts of the application. The client part is executed by the Sapl interpreter, which is available as a Java applet on the client.

However, the approachability of JavaScript is much better compared to Java. The Java runtime environment, the Java Virtual Machine might not even be available on certain platforms (on mobile devices in particular). Besides that, it exhibits significant latency during start-up. For these reasons, a new implementation of this feature is recommended using Sapljs instead of the Sapl interpreter written in Java. Several feature were made to foster this modification:

- The Sapl language was extended with some syntactic sugar to allow distinguishing between constructors and records.
- Automatic conversion of data types like records, arrays, etc, between Sapl and JavaScript was added. In this way full interaction between Sapl and existing libraries in JavaScript became possible.
- Automatic conversion of JSON data structures to enable direct interfacing with all kinds of web-services was added.

6 Related work

Client-side processing for Internet applications is a subject that has drawn much attention in the last years with the advent of Ajax based applications.

Earlier approaches using JavaScript as a client-side platform for the execution of functional programming languages are Hop [15, 10], Links [1] and Curry [7].

Hop is a dedicated web programming language with a HTML-like syntax build on top of Scheme. It uses two compilers, one for compiling the server-side program and one for compiling the client-side part. The client-side part is only used for executing the user interface. The application essentially runs on the client and may call services on the server. Syntactic constructions are used for indicating client and server part code. In [10] it is shown that a reasonably good performance for client-side functions in Hop can be obtained. However, contrary to Haskell and Clean, both Hop and the below mentioned Links are strict functional languages, which simplifies their translation to JavaScript considerably.

Links [1] and its extension Formlets is a functional language-based web programming language. Links compiles to JavaScript for rendering HTML pages, and SQL to communicate with a back-end database. Client-server communication is implemented using Ajax technology, like this is done in the iTask system.

Curry offers a much more restricted approach: only a very restricted subset of the functional-logic language Curry is translated to JavaScript to handle client-side verification code fragments only.

A more recent approach is the Flapjax language [12], an implementation of functional reactive programming in JavaScript. Flapjax can be used either as a programming language, compiling to JavaScript, or as a JavaScript library. Entire applications can be developed in Flapjax. Flapjax automatically tracks dependencies and propagates updates along dataflows, allowing for a declarative style of programming.

An approach to compile Haskell to JavaScript is YCR2JS [4] that compiles YHC Core to JavaScript, comparable to our approach compiling Sapl to JavaScript. Unfortunately, we could not find any performance figures for this implementation.

Another, more recent approach, for compiling Haskell to JavaScript is HS2JS [6], which integrates a JavaScript backend into the GHC compiler. A comparison of JavaScript programs generated by this implementation indicate that they run significantly slower than their Sapljs counterparts.

7 Conclusion and future work

In this paper we evaluated the use of JavaScript as a target language for lazy functional programming languages like Haskell or Clean using the intermediate language Sapl. The implementation has the following characteristics:

- It achieves a speed for compiled benchmarks that is competitive with that of the Sapl interpreter and is faster than interpreters like Amanda, Helium, Hugs and GHCi. This is despite the fact that JavaScript has a 3-5 times slower execution speed than the platforms used to implement these interpreters.

- The execution time of benchmarks is often dominated by memory operations. But in many cases this overhead could be significantly reduced by a simple optimization on the creation of thunks.
- The implementation tries to map Sapl to corresponding JavaScript constructs as much as possible. Only when the lazy semantics of Sapl requires this, an alternative translation is made. This opens the way for additional optimizations based on compile time analysis of programs.
- The implementation supports the full Clean (and Haskell) language, but not all libraries are supported. We tested the implementation against a large number of Clean programs compiled with the Clean to Sapl compiler.

7.1 Future work

We have planned the following future work:

- Implement a web-based Clean to Sapl (or to JavaScript) compiler (experimental version already made).
- Experimenting with supercompilation optimization by implementing a Sapl to Sapl compiler based on whole program analysis.
- Encapsulate JavaScript libraries in a functional way, e.g. using generic programming techniques.
- Attach client-side call-backs written in Clean to iTask editors. It can be implemented using Clean-Sapl dynamics [9] which make it possible to serialize expressions at the server side and execute them at the client side.
- Use JavaScript currying instead of building thunks. Our preliminary results indicate that using JavaScript currying would be significantly slower, but further investigation is needed for proper analysis.

Acknowledgements

The research of the first author was supported by the European Union and the European Social Fund under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003.

References

[1] E. Cooper, S. Lindley, P. Wadler, J. Yallop, Links: web programming without tiers, *Proc. 5th International Symposium on Formal Methods for Components and Objects (FMCO '06)* , *Lecture Notes in Comput. Sci.*, **4709** (2006) 266–296. ⇒94, 95

[2] L. Domoszlai, E. Bruël, J. M. Jansen, The Sapl home page, `http://www.nlda-tw.nl/janmartin/sapl`. ⇒88

[3] B. Fulgham, The computer language benchmark game, `http://shootout.alioth.debian.org`. ⇒90

[4] D. Golubovsky, N. Mitchell, M. Naylor, Yhc.Core – from Haskell to Core, *The Monad.Reader*, **7** (2007) 236–243. ⇒95

[5] J. van Groningen, T. van Noort, P. Achten, P. Koopman, R. Plasmeijer, Exchanging sources between Clean and Haskell – a double-edged front end for the Clean compiler, *Haskell Symposium,* Baltimore, MD, 2010. ⇒79

[6] T. Hallgren, HS2JS test programs, `http://www.altocumulus.org/~hallgren/hs2js/tests/`. ⇒95

[7] M. Hanus, Putting declarative programming into the web: translating Curry to JavaScript, *Proc. 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '07),* Wroclaw, Poland, 2007, ACM, pp. 155–166. ⇒94

[8] J. M. Jansen, P. Koopman, R. Plasmeijer, Efficient interpretation by transforming data types and patterns to functions, *Proc. Seventh Symposium on Trends in Functional Programming (TFP 2006),* Nottingham, UK, 2006. ⇒77, 78, 88, 90, 91

[9] J. M. Jansen, P. Koopman, R. Plasmeijer, iEditors:extending iTask with interactive plug-ins, *Proc. 20th International Symposium on the Implementation and Application of Functional Languages (IFL 2008),* Hertfordshire, UK, 2008, pp. 170–186. ⇒96

[10] F. Loitsch, M. Serrano, Hop client-side compilation, *Trends in Functional Programming (TFP 2007)*, New York, 2007, pp. 141–158. ⇒94

[11] E. Meijer, D. Leijen, J. Hook, Client-side web scripting with HaskellScript, *First International Workshop on Practical Aspects of Declarative Languages (PADL '99),* San Antonio, Texas, 1999, *Lecture Notes in Comput. Sci.*, **1551** (1999) 196–210. ⇒77

[12] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, Flapjax: a programming language for Ajax applications, *SIGPLAN Not.*, **44** (2009) 1–20. ⇒95

[13] R. Plasmeijer, P. Achten, P. Koopman, iTasks: executable specifications of interactive work flow systems for the web, *Proc. 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007),* Freiburg, Germany, 2007, ACM, pp. 141–152. ⇒94

[14] R. Plasmeijer, J. M. Jansen, P. Koopman, P. Achten, Declarative Ajax and client side evaluation of workflows using iTasks, *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '08),* Valencia, Spain, 2008. ⇒77, 94

[15] M. Serrano, E. Gallesio, F. Loitsch, Hop: a language for programming the web 2.0, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006),* Portland, Oregon, 2006, pp. 975–985. ⇒94

Received: January 31, 2011 • Revised: March 23, 2011

Acta Univ. Sapientiae, Informatica, **3**, 1 (2011) 127–136

On scattered subword complexity

Zoltán KÁSA
Sapientia Hungarian University of Transylvania
Department of Mathematics and Informatics,
Tg. Mureş, Romania
email: kasa@ms.sapientia.ro

Abstract. Special scattered subwords, in which the gaps are of length from a given set, are defined. The scattered subword complexity, which is the number of such scattered subwords, is computed for rainbow words.

1 Introduction

Sequences of characters called *words* or *strings* are widely studied in combinatorics, and used in various fields of sciences (e.g. chemistry, physics, social sciences, biology [2, 3, 4, 11] etc.). The elements of a word are called *letters*. A contiguous part of a word (obtained by erasing a prefix or/and a suffix) is a *subword* or *factor*. If we erase arbitrary letters from a word, what is obtained is a *scattered subword*. Special scattered subwords, in which the consecutive letters are at distance at most d ($d \geq 1$) in the original word, are called d-*subwords* [7, 8]. In [9] the *super*-d-*subword* is defined, in which case the distances are of length at least d. The super-d-complexity, as the number of such subwords, is computed for rainbow words (words with pairwise different letters).

In this paper we define special scattered subwords, for which the distance in the original word of length n between two letters which will be consecutive in the subword, is taken from a subset of $\{1, 2, \ldots, n-1\}$.

Computing Classification System 1998: G.2.1, F.2.2
Mathematics Subject Classification 2010: 68R15
Key words and phrases: word complexity, scattered subword, d-complexity, super-d-complexity

The *complexity of a word* is defined as the number of all its different subwords. Similar definitions are for d-*complexity*, *super*-d-*complexity* and *scattered subword complexity*.

The scattered subword complexity is computed in the special case of rainbow words. The idea of using scattered words with gaps of length between two given values is from József Bukor [1].

Another point of view of scattered complexity in the case of non-primitive words is given is [5].

2 Definitions

Let Σ be an alphabet, Σ^n, as usually, the set of all words of length n over Σ, and Σ^* the set of all finite word over Σ.

Definition 1 *Let n and s be positive integers, $M \subseteq \{1, 2, \dots, n-1\}$ and $u = x_1x_2 \dots x_n \in \Sigma^n$. An M-subword of length s of u is defined as $v = x_{i_1}x_{i_2} \dots x_{i_s}$ where*
$i_1 \geq 1$,
$i_{j+1} - i_j \in M$ *for* $j = 1, 2, \dots, s-1$,
$i_s \leq n$.

Definition 2 *The number of M-subwords of a word u for a given set M is the scattered subword complexity, simply M-complexity.*

The M-subword in the case of $M = \{1, 2, \dots, d\}$ is the d-*subword* defined in [7], while in the case of $M = \{d, d+1, \dots, n-1\}$ is the *super*-d-*complexity* defined in [9].
Examples. The word $abcd$ has 11 $\{1, 3\}$-subwords: a, ab, abc, $abcd$, ad, b, bc, bcd, c, cd, d. The $\{2, 3 \dots, n-1\}$-subwords of the word $abcdef$ are the following: a, ac, ad, ae, af, ace, acf, adf, b, bd, be, bf, bdf, c, ce, cf, d, df, e, f.

Hereinafter instead of $\{d_1, d_1+1, \dots, d_2-1, d_2\}$-subword we will use the simple notation (d_1, d_2)-subword.

3 Computing the scattered complexity for rainbow words

Words with pairwise different letters are called *rainbow words*. The M-complexity of a rainbow word of length n does not depend on what letters it contains, and is denoted by $K(n, M)$.

Let us recall two results for special scattered words, as d-subwords and super-d-subwords.

For a rainbow word of length n the super-d-compexity [9] is equal to

$$K\big(n, \{d, d+1, \ldots, n-1\}\big) = \sum_{k\geq 0} \binom{n-(d-1)k}{k+1}, \tag{1}$$

and the $(n-d)$-complexity [8] is

$$K\big(n, \{1, 2, \ldots, n-d\}\big) = 2^n - (d-2)\cdot 2^{d-1} - 2, \text{ for } n \geq 2d-2.$$

For special cases the following propositions can be easily proved.

Proposition 3 *For* $n, d_1 \leq d_2$ *positive integers*

$$K\big(n, \{d_1, d_1+1, \ldots, d_2\}\big) \leq n + \sum_{k\geq 1} \binom{n-(d_1-1)k}{k+1} - \sum_{k\geq 1} \binom{n-d_2k}{k+1}.$$

Proof. This can be obtained from (1) and the formula

$$\begin{aligned} K\big(n, \{d_1, d_1+1, \ldots, d_2\}\big) &\leq K\big(n, \{d_1, d_1+1, \ldots, n-1\}\big) \\ &- K\big(n, \{d_2+1, d_2+2, \ldots, n-1\}\big) + n. \end{aligned}$$

□

For example, $K(7, \{2, 3, 4, 5, 6\}) = 33$, $K(7, \{4, 5, 6\}) = 13$, and from the proposition $K(7, \{2, 3\}) \leq 27$. The exact value is $K(7, \{2, 3\}) = 25$, the two words acg and aeg are not eliminated (here the original distances are 2 and 4 in acg, and 4 and 2 in aeg).

Proposition 4 *For the integers* $n, d \geq 1$, *where* $n = hd + m$

$$K(n, \{d\}) = \frac{(h+1)(n+m)}{2}.$$

Proof.

$$\begin{aligned} K(n, \{d\}) &= n + \sum_{i=1}^{n-d} \left\lfloor \frac{n-i}{d} \right\rfloor = n + d(1+2+\ldots+h-1) + mh \\ &= n + \frac{dh(h-1)}{2} + mh = \frac{(h+1)(n+m)}{2}. \end{aligned}$$

□

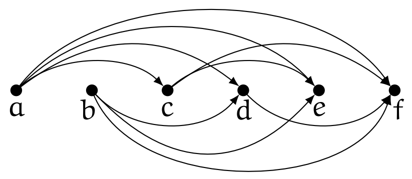

Figure 1: Graph for $(2, n-1)$-subwords when $n = 6$.

To compute the M-complexity of a rainbow word of length n we will use graph theoretical results. Let us consider the rainbow word $a_1a_2\ldots a_n$ and the correspondig digraph $G = (V, E)$, with

$V = \{a_1, a_2, \ldots, a_n\}$,

$E = \{(a_i, a_j) \mid j - i \in M,\ i = 1, 2, \ldots, n, j = 1, 2, \ldots, n\}$.

For $n = 6, M = \{2, 3, 4, 5\}$ see Figure 1.

The adjacency matrix $A = \left(a_{ij}\right)_{i=\overline{1,n},j=\overline{1,n}}$ of the graph is defined by:

$$a_{ij} = \begin{cases} 1, & \text{if } j - i \in M, \\ 0, & \text{otherwise,} \end{cases} \qquad \text{for } i = 1, 2, \ldots, n, j = 1, 2, \ldots, n.$$

Because the graph has no directed cycles, the entry in row i and column j in A^k (where $A^k = A^{k-1}A$, with $A^1 = A$) will represent the number of directed paths of length k from a_i to a_j. If I is the identity matrix (with entries equal to 1 only on the first diagonal, and 0 otherwise), let us define the matrix $R = (r_{ij})$:

$$R = I + A + A^2 + \cdots + A^k, \text{ where } A^{k+1} = O \text{ (the null matrix).}$$

The M-complexity of a rainbow word is then

$$K(n, M) = \sum_{i=1}^{n} \sum_{j=1}^{n} r_{ij}.$$

Matrix R can be better computed using a variant of the well-known Warshall algorithm (for the original Warshall algorithm see for example [12]):

```
WARSHALL($A, n$)
1 $W \leftarrow A$
2 for $k \leftarrow 1$ to $n$
3     do for $i \leftarrow 1$ to $n$
4            do for $j \leftarrow 1$ to $n$
5                   do $w_{ij} \leftarrow w_{ij} + w_{ik}w_{kj}$
6 return $W$
```

From W we obtain easily $R = I + W$.
For example let us consider the graph in Figure 1. The corresponding adjacency matrix is:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

After applying the Warshall algorithm:

$$W = \begin{pmatrix} 0 & 0 & 1 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad R = \begin{pmatrix} 1 & 0 & 1 & 1 & 2 & 3 \\ 0 & 1 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

and then $K\big(6, \{2,3,4,5\}\big) = 20$, the sum of elements in R.

The Warshall algorithm combined with the Latin square method can be used to obtain all nontrivial (with length at least 2) M-subwords of a given rainbow word $a_1a_2\cdots a_n$. Let us consider a matrix $\mathcal{A}$ with the entries A_{ij}, which are set of words. Initially this matrix is defined as:

$$A_{ij} = \begin{cases} \{a_ia_j\}, & \text{if } j - i \in M, \\ \emptyset, & \text{otherwise,} \end{cases} \qquad \text{for } i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, n.$$

If $\mathcal{A}$ and $\mathcal{B}$ are sets of words, $\mathcal{AB}$ will be formed by the set of concatenation of each word from $\mathcal{A}$ with each word from $\mathcal{B}$:

$$\mathcal{AB} = \big\{ab \,\big|\, a \in \mathcal{A}, b \in \mathcal{B}\big\}.$$

If $s = s_1s_2\cdots s_p$ is a word, let us denote by $'s$ the word obtained from s by erasing the first character: $'s = s_2s_3\cdots s_p$. Let us denote by $'A_{ij}$ the set A_{ij}

in which we erase the first character from each element. In this case $'\mathcal{A}$ is a matrix with entries $'A_{ij}$.

Starting with the matrix $\mathcal{A}$ defined as before, the algorithm to obtain all nontrivial M-subwords is the following:

```
Warshall-Latin($\mathcal{A}$, $n$)
1  $\mathcal{W} \leftarrow \mathcal{A}$
2  for $k \leftarrow 1$ to $n$
3      do for $i \leftarrow 1$ to $n$
4             do for $j \leftarrow 1$ to $n$
5                    do if $W_{ik} \neq \emptyset$ and $W_{kj} \neq \emptyset$
6                          then $W_{ij} \leftarrow W_{ij} \cup W_{ik}\, 'W_{kj}$
7  return $\mathcal{W}$
```

The set of nontrivial M-subwords is $\bigcup_{i,j \in \{1,2,\ldots,n\}} W_{ij}$.

For $n = 8$, $M = \{3,4,5,6,7\}$ the initial matrix is:

$$\begin{pmatrix}
\emptyset & \emptyset & \emptyset & \{ad\} & \{ae\} & \{af\} & \{ag\} & \{ah\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \{be\} & \{bf\} & \{bg\} & \{bh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{cf\} & \{cg\} & \{ch\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{dg\} & \{dh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{eh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}.$$

The result of the algorithm Warshall-Latin in this case is:

$$\begin{pmatrix}
\emptyset & \emptyset & \emptyset & \{ad\} & \{ae\} & \{af\} & \{ag, adg\} & \{ah, adh, aeh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \{be\} & \{bf\} & \{bg\} & \{bh, beh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{cf\} & \{cg\} & \{ch\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{dg\} & \{dh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \{eh\} \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\
\emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset
\end{pmatrix}.$$

The algorithm Warshall-Latin can be used for nonrainbow words too, with the remark that repeating subwords must be eliminated. For the word $aabbbaaa$ and $M = \{3,4,5,6,7\}$ the result is: aa, ab, aba, ba.

4 Computing the (d_1, d_2)-complexity

Let us denote by a_i the number of (d_1, d_2)-subwords which terminate at position i in a rainbow word of length n. Then

$$a_i = 1 + a_{i-d_1} + a_{i-d_1-1} + \cdots + a_{i-d_2}, \tag{2}$$

with the remark that for $i \leq 0$ we have $a_i = 0$. Subtracting a_{i-1} from a_i we get the following simpler equation.

$$a_i = a_{i-1} + a_{i-d_1} - a_{i-1-d_2}.$$

The (d_1, d_2)-complexity of a rainbow word of length n is

$$K\big(n, \{d_1, d_1+1, \ldots, d_2\}\big) = \sum_{i=1}^{n} a_i \tag{3}$$

For example, if $d_1 = 2, d_2 = 4$, the following values are obtained

n	1	2	3	4	5	6	7	8	9	10	11	12	13
a_n	1	1	2	3	5	7	11	16	24	35	52	76	112
$K(n, \{2,3,4\})$	1	2	4	7	12	19	30	46	70	105	157	233	345

If we denote by $A(z) = \sum_{n\geq 1} a_n z^n$ the generating function of the sequence a_n, then from (2) we obtain

$$\sum_{n\geq 1} a_n z^n = \sum_{n\geq 1} z^n + \sum_{n\geq 1} a_{n-d_1} z^{n-d_1} + \cdots + \sum_{n\geq 1} a_{n-d_2} z^{n-d_2},$$

and

$$A(z) = \frac{z}{1-z} + z^{d_1} A(z) + \cdots + z^{d_1} A(z).$$

From this we obtain

$$A(z) = \frac{z}{z^{d_2+1} - z^{d_1} - z + 1}. \tag{4}$$

For $d_1 = 2, d_2 = 4$ the sequence $(a_n)_{n\geq 0}$ ([10] sequence A023435) corresponds to a variant of the dying rabbits problem [6].

To compute the generating function for the complexity $K\big(n, \{d_1, d_1+1, \ldots, d_2\}\big)$, let us denote this complexity simply by K_n only, and its generating function by $K(z) = \sum_{n\geq 1} K_n z^n$. We remark that $K_n = 0$ for $n \leq 0$, and $K_1 = 1$.

From (3) and (4) we can immediately conclude that

$$K(z) = \frac{1}{1-z}A(z) = \frac{z}{(1-z)(z^{d_2+1} - z^{d_1} - z + 1)}\,.$$

5 Correspondence between $(d, n+d-1)$-subwords and $\{1, d\}$-subwords

The following result is inspired from the sequence A050228[1] of [10].

Proposition 5 *The number of* $\{1, d\}$*-subwords of a rainbow word of length* n *is equal to the number of* $\{d, d+1, \ldots, n+d-1\}$*-subwords of length at least 2 of a rainbow word of length* $n+d$*.*

Proof. By the generalization of the sequence A050228 [10] the number of the $\{1, d\}$-subwords of a rainbow word of length n is equal to

$$K\big(n, \{1, d\}\big) = \sum_{k\geq 0} \binom{n+1-(d-1)k}{k+2}.$$

From (1) we have

$$K\big(n+d, \{d, d+1, \ldots, n+d-1\}\big) - (n+d) = \sum_{k\geq 1} \binom{n+d-(d-1)k}{k+1}.$$

By changing k to $k+1$ in the sum, we obtain $\sum_{k\geq 0} \binom{n+1-(d-1)k}{k+2}$, and this proves the theorem. □

Example. For $abcde$ the 19 $\{1, 3\}$-subwords are:
$a, b, c, d, e, ab, abc, abcd, ad, ade, abcde, abe, bc, bcd, bcde, be, cd, cde, de$.

For $abcdefgh$ the 19 $\{3, 4, 5, 6, 7\}$-subwords of length at least 2 are:
$ad, ae, af, ag, adg, ah, adh, aeh, be, bf, bg, bh, beh, cf, cg, ch, dg, dh, eh$.

[1]A050228: a_n is the number of subsequences $\{s_k\}$ of $\{1, 2, 3, \ldots n\}$ such that $s_{k+1} - s_k$ is 1 or 3.

Conclusions

A special scattered subword, the so-called M-subword is defined, in which the distances (gaps) between letters are from the set M. The number of the M-subwords of a given word is the M-complexity. Graph algorithms are used to compute the M-complexity and to determine all M-subwords of a rainbow word. This notion of M-complexity is a generalization of the d-complexity [7] and of the super-d-complexity [9]. If M consists of successive numbers from d_1 to d_2 then the so-called (d_1, d_2)-complexity is computed by recursive equations and generating functions.

Acknowledgements

This work was supported by the project under the grant agreement no. TÁMOP 4.2.1/B-09/1/KMR-2010-0003 (Eötvös Loránd University, Budapest) financed by the European Union and the European Social Fund.

References

[1] J. Bukor, Personal communication at the *8th Joint Conference on Mathematics and Computer Science*, Komárno (Slovakia), July 14–17, 2010. ⇒128

[2] W. Ebeling, R. Feistel, *Physik der Selbstorganisation und Evolution,* Akademie-Verlag, Berlin, 1982. ⇒127

[3] C. Elzinga, S. Rahmann, H. Wang, Algorithms for subsequence combinatorics, *Theor. Comput. Sci.* **409,** 3 (2008) 394–404. ⇒127

[4] C. H. Elzinga, Complexity of categorial time series, *Sociological Methods & Research* **38,** 3 (2010) 463–481. ⇒127

[5] Sz. Zs. Fazekas, B. Nagy, Scattered subword complexity of non-primitive words, *J. Autom. Lang. Comb.* **13,** 3–4 (2008) 233–247. ⇒128

[6] V. E. Hoggatt Jr., D. A. Lind, The dying rabbit problem, *Fib. Quart.* **7,** 5 (1969), 482–487. ⇒133

[7] A. Iványi, On the d-complexity of words, *Ann. Univ. Sci. Budapest., Sect. Comput.* **8** (1987) 69–90. ⇒127, 128, 135

[8] Z. Kása, On the **d**-complexity of strings, *Pure Math. Appl.* **9,** 1–2 (1998) 119–128. ⇒127, 129

[9] Z. Kása, Super-**d**-complexity of finite words, *MACS 2010: 8th Joint Conference on Mathematics and Computer Science*, Selected Papers, Komárno (Slovakia), July 14–17, 2010, pp. 251–261. ⇒ 127, 128, 129, 135

[10] N. J. A. Sloane, The on-line encyclopedia of integer sequences, `http://www.research.att.com/~njas/sequences/`. ⇒133, 134

[11] O. G. Troyanskaya, O. Arbell, Y. Koren, G. M. Landau, A. Bolshoy, Sequence complexity profiles of prokaryotic genomic sequences: A fast algorithm for calculating linguistic complexity, *Bioinformatics* **18,** 5 (2002) 679–688. ⇒127

[12] S. Warshall, A theorem on Boolean matrices, *J. ACM* **9,** 1 (1962) 11–12. ⇒130

Received: December 4, 2010 • Revised: March 12, 2011

Acta Universitatis Sapientiae

The scientific journal of Sapientia Hungarian University of Transylvania publishes original papers and surveys in several areas of sciences written in English. Information about each series can be found at `http://www.acta.sapientia.ro`.

Acta Universitatis Sapientiae, Informatica

Each volume contains two issues.

Sapientia University Scientia Publishing House

ISSN 1844-6086
http://www.acta.sapientia.ro

Information for authors

Acta Universitatis Sapientiae, Informatica publishes original papers and surveys in various fields of Computer Science. All papers are peer-reviewed.

Papers published in current and previous volumes can be found in Portable Document Format (pdf) form at the address: `http://www.acta.sapientia.ro`.

The submitted papers should not be considered for publication by other journals. The corresponding author is responsible for obtaining the permission of coauthors and of the authorities of institutes, if needed, for publication, the Editorial Board is disclaiming any responsibility.

Submission must be made by email (`acta-inf@acta.sapientia.ro`) only, using the LaTeX style and sample file at the address `http://www.acta.sapientia.ro`. Beside the LaTeX source a pdf format of the paper is needed too.

Prepare your paper carefully, including keywords, ACM Computing Classification System codes (`http://www.acm.org/about/class/1998`) and AMS Mathematics Subject Classification codes (`http://www.ams.org/msc/`).

References should be listed alphabetically based on the Intructions for Authors given at the address `http://www.acta.sapientia.ro`.

Illustrations should be given in Encapsulated Postscript (eps) format.

One issue is offered each author free of charge. No reprints will be available.

Contact address and subscription:
Acta Universitatis Sapientiae, Informatica
RO 400112 Cluj-Napoca
Str. Matei Corvin nr. 4.
Email: acta-inf@acta.sapientia.ro

Printed by Gloria Printing House
Director: Péter Nagy

ISSN 1844-6086
http://www.acta.sapientia.ro