# Turchin's Relation for Call-by-name Computations: a Formal Approach

Antonina Nepeivoda
Program Systems Institute of Russian Academy of Sciences*
Pereslavl-Zalessky, Russia
a_nevod@mail.ru

Supercompilation is a program transformation technique that was first described by V.F. Turchin in the 1980s. In supercompilation, Turchin's relation as a similarity relation on call stack configurations is used both for call-by-value and call-by-name semantics to terminate unfolding of the program being transformed. In this paper, we give a formal grammar model of call-by-name stack behaviour. We classify the model in terms of the Chomsky hierarchy and then formally prove that Turchin's relation can terminate all computations generated by the model and estimate how long may be a sequence of call stacks generated by a program until it is terminated via Turchin's relation.

## 1 Introduction

In recent years, general-purpose program transformation tools such as supercompilers are proved to be useful in some verification tasks, e. g., verification of cache-coherence protocols [10] and search of attacks on some cryptographic protocols [12]. In this domain, two different approaches to verification are used. The first approach is mostly case-studying. It describes program transformation tools that are successful for solving some verification problems ([1]). The problems themselves can be very complicated (e. g., the missionaries and cannibals puzzle which was solved in a generic case by interactive supercompilation [9]). So the approach focuses on searching for "stories of success" without describing uniform modelling algorithms for similar problems or displaying formal proofs. Although showing how the program transformation can be efficient when solving real-life tasks [8, 5], this approach can tempt a researcher to produce some new program transformation algorithms which are perfectly suited for solving a particular problem.

The second approach is much more formal: its aim is to describe some problem classes for which verification can be proved to be solvable by a given program transformation method. These works are usually inspired by the "stories of success" found by the first approach. Many problems that are known to be solvable are already successfully automatically solved by specialized verifiers, and generic purpose program transformation tools cannot compare to them with respect to efficiency and convenience. However, research in this direction can indicate some possibly successful applications of the program transformation tools by proving that these applications can be done in theory. It may be useful first to study computational power of some program transformation technique and then to use this knowledge for searching of problems that can be handled in the scope of this computational power. This way of thinking already proved itself to be useful: after studying properties of the homeomorphic embedding relation on prefix-grammar-generated traces, it became possible to use prefix grammars as models for a set of ping-pong cryptographic protocols [3] and to prove that supercompiling the corresponding program models is

*The reported study was partially supported by RFBR, research project No. 14-07-00133 a

Submitted to:
VPT 2016

equivalent to verifying the corresponding protocol models in the classical case [12]. But the prefix grammar model makes sense only for program languages based on call-by-value semantics, although most program transformation tools analyze call-by-name programs. So it is interesting to model behaviour of the function call stacks for the call-by-name semantics by the way similar to prefix grammars and find out whether this new model is able to solve more formal problems than the prefix grammars model. The theoretical answer to this question is definitely yes.

The paper is organized as follows. First, we recall notions from [13] to which we refer further. Then we define a class of multi-layer prefix grammars and describe its computational power. Then we show how the defined grammars can be used to model the call stack behaviour for call-by-name computations. Finally, we refine the definition of Turchin's relation for the new class of grammars and give some estimations of bad sequence length.

Our contributions are the following:

1. We give a formal model of the call-by-name stack behaviour and estimate its computational power.
2. We formally prove that the Turchin relation on the call stack sequences generated by the model is well binary.
3. We give some bounds on the maximal number of steps of a computation path before the Turchin relation terminates the path.

## 2 Preliminaries

There we recall some simple definitions from [13], to which we refer further in this paper.

**Definition 1.** *Given a set $S$ and a relation $R \subset S \times S$, $R$ is called* a well binary relation*, if every sequence $\{\Phi_n\}$ of elements from $S$ such that $\forall i, j (i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is finite. So, a well binary relation is "a well quasi-order without the order" (i. e., it is not necessarily transitive).*

*A sequence $\{\Phi_n\}$ satisfying the property $\forall i, j (i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is called* a bad sequence *with respect to $R$.*

**Definition 2.** *A tuple $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, where $\Upsilon$ is an alphabet, $\Gamma_0 \in \Upsilon^+$ is an initial word, and $\mathbf{R} \subset \Upsilon^+ \times \Upsilon^*$ is a finite set of rewriting rules, is called* a prefix grammar *if $R : R_l \to R_r \in \mathbf{R}$ can be applied only to words of the form $R_l\Phi$ (where $R_l$ is a prefix of the word $R_l\Phi$ and $\Phi \in \Upsilon^*$ is arbitrary) and generates only words of the form $R_r\Phi$.*

*If the left-hand sides $R_l$ of all rules $R : R_l \to R_r \in \mathbf{R}$ have the length 1 (only the first letter of a word is changed by any rule) then the prefix grammar is called* an alphabetic prefix grammar.

A trace *of a prefix grammar $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ is a word sequence $\{\Phi_i\}$ (finite or infinite) where $\Phi_1 = \Gamma_0$ and for all $i$ $\exists R(R : R_l \to R_r \ \&\ R \in \mathbf{R} \ \&\ \Phi_i = R_l\Theta \ \&\ \Phi_{i+1} = R_r\Theta)$ (where $\Theta$ is a suffix). In other words, the elements of a trace are derived from their predecessors by applying the rewriting rules from $\mathbf{G}$.*

Finally, we recall how the Turchin relation is defined for traces generated by a prefix grammar. We say that a letter in a word in a trace was changed in the trace segment, if the letter is generated by some rule that was applied to some word in the trace segment.

**Example 1.** *Let $\Gamma_0$ be $aa$, and let us apply rule $a \to ba$ to $\Gamma_0$. In the trace segment*

$$\Gamma_0 : aa \to \Gamma_1 : baa$$

*the first letter $a$ in $\Gamma_1$ does not coincide with the first letter $a$ in $\Gamma_0$ (it is rewritten by $a \to ba$) and the second letter $a$ in $\Gamma_1$ is unchanged with respect to $\Gamma_0$.*

The following definition formalizes the definition of the Turchin relation for function call stacks given in [11].

**Definition 3.** *Let $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ be a prefix grammar. We say that two words $\Gamma_i$, $\Gamma_j$ in a trace $\{\Gamma_k\}$ form* a Turchin pair *(denoted as $\Gamma_i \preceq \Gamma_j$) if $\Gamma_i = \Phi\Theta_0$, $\Gamma_j = \Phi\Psi\Theta_0$ and the suffix $\Theta_0$ is not changed in the trace segment from $\Gamma_i$ to $\Gamma_j$.*

## 3 Multi-layer Prefix Grammars

Let $\Upsilon$ be an alphabet. Let $\mathbf{S}$ be *a label set* and $\lhd$ be a strict (non-reflexive) partial order relation over $\mathbf{S}$. We denote labels from $\mathbf{S}$ by the letters $s$, $t$ (maybe with subscripts). Let us say that $s_1$ is a child of $s_0$ w.r.t. $\mathbf{S}' \subseteq \mathbf{S}$ (denoted by $s_1 = \text{child}(s_0)[\mathbf{S}']$) if $s_0 \lhd s_1$, $s_0 \in \mathbf{S}'$, $s_1 \in \mathbf{S}'$ and there is no such $s_2 \in \mathbf{S}'$ that $s_0 \lhd s_2$ and $s_2 \lhd s_1$. The inverse for the child relation is the parent relation. Given a set $\mathbf{S}' \subseteq \mathbf{S}$, if $\mathbf{S}'$ contains no children or parents of a label $t$ then we call *$t$ a free label* w.r.t. $\mathbf{S}'$[1].

Informally, the labels can be considered as nodes of trees with unbounded branching, then the child–parent relation has its usual meaning.

Henceforth, Greek capitals $\Gamma$, $\Delta$, $\Phi$, $\Psi$, $\Xi$, $\Theta$ denote words from $\{\langle a, s_i \rangle | a \in \Upsilon \;\&\; s_i \in \mathbf{S}\}^*$. Finite sequences of such pairs are called *layered words*. If $\langle a_1, s_1 \rangle \dots \langle a_n, s_n \rangle$ is a layered word, the corresponding *plain word* is defined as $a_1 \dots a_n$. The empty word is denoted by $\Lambda$.

If $\Phi$ is a layered word, $|\Phi|$ stands for the number of pairs in $\Phi$ and $\Phi[i]$ stands for the $i$-th pair. For the sake of brevity, layered word $\langle a_1, s_0 \rangle \dots \langle a_n, s_0 \rangle$ can be also written as $\langle a_1 \dots a_n, s_0 \rangle$ (thus, $a\langle s_0 \rangle$ is an equivalent form for $\langle a, s_0 \rangle$).

$\Phi\langle s_0 \rangle$ denotes the maximal subsequence of $\Phi$ containing only pairs with the label $s_0$. $\Phi\langle \hat{s_0} \rangle$ denotes the maximal subsequence of $\Phi$ not containing pairs with the label $s_0$. The set of all labels in $\Phi$ is denoted by $\mathbf{S}_\Phi$.

**Example 2.** *Let $\Phi = \langle a_1, s_1 \rangle \langle a_2, s_1 \rangle \langle a_3, s_2 \rangle \langle a_4, s_4 \rangle \langle a_5, s_1 \rangle \langle a_6, s_3 \rangle \langle a_7, s_4 \rangle$. Then $\Phi\langle s_1 \rangle = \langle a_1 a_2 a_5, s_1 \rangle$, $\Phi\langle \hat{s_1} \rangle = \langle a_3, s_2 \rangle \langle a_4, s_4 \rangle \langle a_6, s_3 \rangle \langle a_7, s_4 \rangle$.*

*If $s_1 \lhd s_2$, $s_2 \lhd s_3$, $s_1 \lhd s_4$ (and not $s_2 \lhd s_4$), then the layered word $\Phi$ can be represented as the following tree:*

$$\begin{tikzcd}
& s_1 : a_1 a_2 a_5 \arrow[dl] \arrow[dr] & \\
s_2 : a_3 \arrow[d] & & s_4 : a_4 a_7 \\
s_3 : a_6 & &
\end{tikzcd}$$

*The order of the letters in $\Phi$ is significant for the tree representation only if the letters have the same label*[2].

Given a label $s_i$ and natural numbers $K_1$ and $K_2$, we define a set of *layer functions* $\mathfrak{F}^{s_i}_{K_1,K_2} : \{\langle a, t \rangle | a \in \Upsilon \;\&\; t \in \mathbf{S}\}^* \to \{\langle a, t \rangle | a \in \Upsilon \;\&\; t \in \mathbf{S}\}^*$ as a minimal set of functions containing all compositions of $K_1$ elementary functions, which are:

[1]In most cases, we assume that $\mathbf{S}'$ is a set of all previously used labels, hence there is no need to write it in the square brackets in expressions like $\text{child}(s_0)[\mathbf{S}']$.

[2]Hence, the tree above is a representation not only of the word $\Phi$, but also, for example, of the word $\langle a_6, s_3 \rangle \langle a_3, s_2 \rangle \langle a_1, s_1 \rangle \langle a_4, s_4 \rangle \langle a_7, s_4 \rangle \langle a_2, s_1 \rangle \langle a_5, s_1 \rangle$

1. Appending $\mathrm{App}^{s_j}[\Psi]$ ($s_j \in \mathbf{S}$, $\Psi \in \Upsilon^*$): given a layered word $\Phi$, $\mathrm{App}^{s_j}[\Psi](\Phi)$ is the sequence $\Phi\Psi\langle s_j\rangle$ such that $s_j$ is a child of $s_i$ w.r.t. $\mathbf{S}_\Phi \cup \{s_j\}$, $s_j$ is free w.r.t. $\mathbf{S}_\Phi \setminus \{s_i, s_j\}$, and $|\Psi| \leq K_2$.
   For example, if $\mathrm{App}^{s_1}[a] \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_1$, then
   $$\overset{s_1}{\mathrm{App}}[a](\langle a, s_0\rangle\langle b, s_1\rangle) = \langle a, s_0\rangle\langle b, s_1\rangle\langle a, s_1\rangle$$
   On tree representations, the appending operation appends some new letters to an existing node.
2. Lifting $\mathrm{Lift}^{s_j}[\Psi\langle s_k\rangle](s_j, s_k \in \mathbf{S}, \Psi \in \Upsilon^*)$: given $\Phi$ with a non-empty $\Phi\langle s_j\rangle$, where $s_j$ is a child of $s_i$ w.r.t. $\mathbf{S}_\Phi$, $\mathrm{Lift}^{s_j}[\Psi\langle s_k\rangle](\Phi)$ is the sequence $\Phi\Psi\langle s_k\rangle$ where $|\Psi| \leq K_2$ and $s_k$ is a child of $s_i$ w.r.t. $\mathbf{S}_\Phi \cup \{s_k\}$, $s_k$ is free w.r.t $\mathbf{S}_\Phi \setminus \{s_i\}$ and $s_j$ is a child of $s_k$ w.r.t $\mathbf{S}_\Phi \cup \{s_k\}$.
   For example, if $\mathrm{Lift}^{s_1}[a\langle s_2\rangle] \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_1$, then
   $$\overset{s_1}{\mathrm{Lift}}[\langle a, s_2\rangle](\langle a, s_0\rangle\langle b, s_1\rangle) = \langle a, s_0\rangle\langle b, s_1\rangle\langle a, s_2\rangle$$
   The lifting operation differs from the appending operation only by introduction of an unused child label $s_k$, which marks the newly appended word $\Psi$. On tree representations, the lifting operation inserts a new node between the nodes labelled by $s_i$ and $s_j$.
3. Deleting $\mathrm{Del}^{s_j}$ ($s_j \in \mathbf{S}$): given $\Phi$ with a non-empty $\Phi\langle s_j\rangle$, $s_j = \mathrm{child}(s_i)$ w.r.t. $\mathbf{S}_\Phi$, $\mathrm{Del}^{s_j}$ erases $\Phi\langle s_j\rangle$ from $\Phi$ together with all $\Phi\langle t\rangle$ for which $s_j \lhd t$.
   For example, if $\mathrm{Del}^{s_1} \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_1$, $s_2 = \mathrm{child}(s_1)$, then
   $$Delete^{s_1}(\langle a, s_0\rangle\langle b, s_1\rangle\langle c, s_2\rangle) = \langle a, s_0\rangle$$
   On tree representations, the deleting operation deletes the subtree, which uppermost node is labelled by $s_j$.
4. Copying $\mathrm{Copy}^{s_j}$ ($s_j \in \mathbf{S}$): given $\Phi$ with a non-empty $\Phi\langle s_j\rangle$, $s_j = \mathrm{child}(s_i)$ w.r.t. $\mathbf{S}_\Phi$, $\mathrm{Copy}^{s_j}$ appends $\Phi\langle s_k\rangle$ to $\Phi$, where $s_k$ is a child of $s_i$ w.r.t. $\mathbf{S}_\Phi \cup \{s_j\}$, $s_j$ is free w.r.t. $\mathbf{S}_\Phi \setminus \{s_i\}$ and then appends all subsequences $\Phi\langle s_l\rangle$ labelled by incomparable children of $s_j$ and labels them by incomparable children of $s_l$ and so on until all the sequences $\Phi\langle t\rangle$, where $s_j \lhd t$, are copied exactly once.
   For example, if $\mathrm{Copy}^{s_1} \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_1$, $s_2 = \mathrm{child}(s_1)$, then
   $$\overset{s_1}{\mathrm{Copy}}(\langle a, s_0\rangle\langle b, s_1\rangle\langle c, s_2\rangle) = \langle a, s_0\rangle\langle b, s_1\rangle\langle c, s_2\rangle\langle b, s_3\rangle\langle c, s_4\rangle,$$
   where $s_3$ and $s_4$ are incomparable with $s_1$ and $s_2$, $s_4 = \mathrm{child}(s_3)$, $s_3 = \mathrm{child}(s_0)$.
   On tree representations, the copying operation creates a copy of the subtree, which uppermost node is labelled by $s_j$.

**Example 3.** *Let $\Upsilon = \{a, b\}$, $\{\mathbf{S}, \lhd\} = \{\mathbb{Q}, <\}$. Let $\mathfrak{F}^1_{2,1}$ be determined by the following basic functions:*

1. *$App^3[a]$, $App^4[b]$.*
2. *$Lift^3[c\langle 2\rangle]$.*

*Consider the word $a\langle 1\rangle$. The set of all images of $\mathfrak{F}^1_{2,1}(a\langle 1\rangle)$ is $\{a\langle 1\rangle aa\langle 3\rangle, a\langle 1\rangle bb\langle 4\rangle, a\langle 1\rangle c\langle 2\rangle a\langle 3\rangle\}$.*

*Because the label 3 is not free w.r.t. to word $a\langle 1\rangle b\langle 4\rangle$, and the label 4 is not free w.r.t. to word $a\langle 1\rangle a\langle 3\rangle$, we cannot apply $App^3[a]$ when $App^4[b]$ is applied, and vice versa.*

In Section 4, it is shown how the described layer functions model call stack transformations.

**Definition 4.** *Let us consider a tuple* $\mathbf{G} = \langle \Upsilon, \mathbf{S}, \mathbf{R}, \mathfrak{F}^x_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *where* $\Gamma_0$ *and* $\Delta_0$ *are layered words over* $\Upsilon \times \mathbf{S}$ *such that for every* $\Gamma_0[i] = \langle a_i, s_i\rangle$ *and* $\Gamma_0[j] = \langle a_j, s_j\rangle$*, if* $i > j$ *then* $s_j \lhd s_i$ *or* $s_j = s_i$*,* \$ *is a special symbol not belonging to* $\Upsilon$ *and* $\mathfrak{F}^x_{K_1,K_2}$ *is a finite set of layer function forms where x runs over* $\mathbf{S}$*. For every* $\mathbf{G}$*-word* $\Gamma\$\Delta$*, where* $\Gamma$ *and* $\Delta$ *are words over* $\Upsilon \times \mathbf{S}$*, we call* $\Gamma$ the visible layer*, and we call* $\Delta$ the invisible layer *of* $\Gamma\$\Delta$*.*

*Let all rewriting rules from* $\mathbf{R}$ *have one of the following forms:*

1. $\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Phi\langle s_i\rangle\Theta\$F^{s_i}(\Psi)$, $|\Phi| \le K_2$, $|\Xi| \le K_2 - 1$, $F^{s_i}(\Psi) \in \mathfrak{F}^{s_i}_{K_1,K_2}$.
2. *For some* $s_j = child(s_i) \in \mathbf{S}$,

$$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Psi\langle s_j\rangle\Phi\langle s_i\rangle\Theta\$F^{s_i}(\Psi\langle \hat{s_j}\rangle),$$

   $|\Phi| \le K_2$, $|\Xi| \le K_2 - 1$, $F^{s_i}(\Psi) \in \mathfrak{F}^{s_i}_{K_1,K_2}$*. The rules having this form are called* pop rules*. A pop rule cannot "look into" containment of the invisible layer of a transformed word: if there are no children of* $s_i$*,* $\Psi\langle child(s_i)\rangle$ *will be* $\Lambda$*; otherwise we may specify* $s_j$*, but there are no ways to specify* $\Psi\langle s_j\rangle$*.*

*Such a grammar* $\mathbf{G}$ *is called* a multi-layer prefix grammar*.* $K_2$ *is called* the maximal rewrite depth*,* $K_1$ *is called* the maximal replication index*. A sequence of* $\mathbf{G}$*-words starting with* $\Gamma_0\$\Delta_0$ *that is transformed by the rules from* $\mathbf{R}$ *is called* a trace *over* $\mathbf{G}$*.*

*If any rule of such a grammar changes only one letter of the visible layer, then the multi-layer prefix grammar is* alphabetic*.*

**Definition 5.** *Let* $\Phi\Theta\$\Delta_i$ *be the i-th* $\mathbf{G}$*-word in a trace* $\{\Gamma_k\$\Delta_k\}$ *generated by an alphabetic multi-layer prefix grammar* $\mathbf{G}$*. If* $\Gamma_j\$\Delta_j = \Psi\Theta\$\Delta_j$ *(*$j > i$*) then we say that* $\Psi$ *is* a derivative prefix *(or simply* a derivative*) of* $\Phi$ *(denoted by deriv*$(\Phi)$*).*

Now we can prove some simple propositions about the multi-layer grammars.

**Proposition 1.** *The three following properties hold.*

1. *If* $\mathbf{G}$ *is an alphabetic multi-layer prefix grammar and* $\Gamma\$\Delta$ *is a* $\mathbf{G}$*-word generated by* $\mathbf{G}$*, then for every* $\Gamma[i] = \langle a_i, s_i\rangle$*,* $\Gamma[j] = \langle a_j, s_j\rangle$ *if* $i < j$ *then either* $s_i = s_j$ *or* $s_i \lhd s_j$*.*
2. *Let an alphabetic multi-layer grammar* $\mathbf{G}$ *generate* $\Gamma\$\Psi\Psi\langle s_1\rangle\Psi\langle s_2\rangle$ *such that* $s_1$ *and* $s_2$ *are not equal and incomparable. Then derivatives of* $\Psi\langle s_1\rangle$ *and* $\Psi\langle s_2\rangle$ *cannot occur in the visible part of the same word.*
3. *Let* $\mathbf{G}$ *generate a word* $\langle a, s_i\rangle\Gamma\langle b, s_j\rangle\Theta\$\Psi$*. In a trace containing this word, all the words have occurrences of deriv*$(\langle a, s_i\rangle)$ *before occurrences of deriv*$(\langle b, s_j\rangle)$ *in the visible layer.*

*Proof.* 1. By the definition, the initial word $\Gamma_0\$\Delta_0$, satisfies the stated property. If some word $\langle a, s_i\rangle\Gamma\$\Delta$ satisfies the proposition statement, a non-pop rule can only prepend children of $\langle a, s_i\rangle$ to $\Gamma$, and a pop rule can only append descendants of $\langle a, s_i\rangle$ to $\Gamma$. Therefore, all the words generated from $\langle a, s_i\rangle\Gamma\$\Delta$ must satifsy the stated property.

2. If $s_1$ and $s_2$ are incomparable then the labels of the derivatives of $\Psi\langle s_1\rangle$ and $\Psi\langle s_2\rangle$ are also incomparable. And the stated property follows from the case (1).

3. In a trace generated by $\mathbf{G}$, deriv$(\langle b, s_j\rangle)$ can be generated only after transformations of the word $\langle b, s_j\rangle\Theta\$\Psi'$, and this word can contain deriv$(\langle a, s_i\rangle)$ only in the invisible layer. All deriv$(\langle b, s_j\rangle)$ are appended to the end of the invisible layer (by the definitions of $App^{s_k}[\Psi]$, $\mathrm{Lift}^{s_k}[\Psi\langle s_u\rangle]$, $\mathrm{Copy}^{s_k}$). They are marked by the labels, which are not less (but may be incomparable) than the labels of deriv$(\langle a, s_i\rangle)$. Hence, after applying the pop rules, deriv$(\langle b, s_j\rangle)$ will occur in the visible layer after deriv$(\langle a, s_i\rangle)$. □

**Definition 6.** *Let* $\mathbf{G} = \langle \Upsilon, \mathbf{S}, R, \mathfrak{F}^{x}_{K_1,K_2}, \Gamma_0\$\Delta_0 \rangle$ *be a multi-layer prefix grammar with* $|\Delta_0| > 0$*. The set of all the words* $A \in \Upsilon$ *s.t.* $A$ *is a plain word corresponding to the visible layer* $\Delta$ *of some* $\mathbf{G}$*-word* $\Delta\$\Lambda$ *produced by a finite trace of* $\mathbf{G}$ *is called* a language generated by $\mathbf{G}$.

**Theorem 1.** *Every recursively enumerable set can be generated by a multi-layer prefix grammar.*

*Proof.* It is sufficient to prove that, given an input word, every Turing machine on the input can be emulated by a multi-layer prefix grammar treating the input word as its initial word.

Consider an arbitrary Turing machine $\langle Q, \Upsilon_A, b, \sigma, q_0, F \rangle$, where $Q$ is a state alphabet, $\Upsilon_A$ is a tape alphabet, $b$ is the blank symbol, $q_0 \in Q$ is the initial state, $F \subset Q$ is a set of final states, and $\sigma \subset Q \times \Upsilon_A \times Q\Upsilon_A \times \{L, R\}$ is a set of transition rules, and an input $I \in \Upsilon_A$. Let us introduce a multi-layer prefix grammar with the alphabet $\Upsilon = \Upsilon_A \cup Q^R \cup Q^L \cup \{Blank^L, Blank^R, b\}$, where $Q^R$ and $Q^L$ are the state alphabets $Q$ marked by the superscripts meaning "a state after moving to the right cell" and "a state after moving to the left cell" correspondingly. $Blank^R$ and $Blank^L$ are special "end-marks" referring to blanks on the tape after the rightmost and leftmost cells reached in a computation. All blanks on the tape between them are denoted in the model grammar by usual $b$ symbols.

Let us assume in this proof that all labels in the model grammar have the form $s_i$, where $i$ is a rational number ($i \in \mathbb{Q}$), and $s_i \lhd s_j$ iff $i < j$ ($i \in \mathbb{Q}$, $j \in \mathbb{Q}$).

The initial word in the model grammar is

$$\Gamma_0\$\Delta_0 = \langle q_0^R, s_0 \rangle I \langle s_0 \rangle \langle Blank^R, s_0 \rangle \$ \langle Blank^L, s_1 \rangle$$

In order to emulate a rule $(q_1, a_1) \to (q_2, a_2, R) \in \sigma$, we use one of the following two rewrite schemes ($x$ is a letter variable):

$$\langle q_1^R, s_i \rangle \langle a_1, s_j \rangle \Phi\$\Psi \langle x, s_k \rangle \to \langle q_2^R, s_j \rangle \Phi\$\Psi \langle x, s_k \rangle \langle a_2, s_{\frac{j+k}{2}} \rangle$$

$$\langle a_1, s_j \rangle \langle q_1^L, s_i \rangle \Phi\$\Psi \langle x, s_k \rangle \to \langle q_2^R, s_j \rangle \Phi\$\Psi \langle x, s_k \rangle \langle a_2, s_{\frac{j+k}{2}} \rangle$$

In order to emulate a rule $(q_1, a_1) \to (q_2, a_2, L)$, we use one of the following two schemes:

$$\langle q_1^R, s_i \rangle \Phi\$\Psi \langle x, child(s_i) \rangle \to \langle x, child(s_i) \rangle \langle q_2^L, child(s_i) \rangle \langle a_2, child(s_i) \rangle \Phi\$\Psi$$

$$\langle a_1, s_j \rangle \langle q_1^L, s_i \rangle \Phi\$\Psi \langle x, child(s_i) \rangle \to \langle x, child(s_i) \rangle \langle q_2^L, child(s_i) \rangle \langle a_2, child(s_i) \rangle \Phi\$\Psi$$

So we model the tape part to the right of the machine head by the visible layer, and the tape part to the left of the machine head by the invisible layer. The letters from $Q^R \cup Q^L$ can only appear in the visible layer, and what is more, a letter from $Q^R$ may be only the first letter of the visible layer, and a letter from $Q^L$ — the second letter. □

In the proof above we construct a grammar, every rule of which changes the two first letters of the visible part of a $\mathbf{G}$-word. If we model function call stacks by multi-layer prefix grammars, this can be possible only if function definitions contain nested functions in the patterns. In the case of the plain prefix grammars, the class of generating alphabetic prefix grammars defines the regular languages as well as the class of all prefix grammars [2]. In the case of the alphabetic multi-layer prefix grammars, the situation changes drastically. We can informally compare their power to the 1-state Turing machines, although the grammars modelling Turing machines use only the lifting layer functions, while copying and appending are left aside.

In the case of the multi-layer prefix grammars, the state of the art is presented by the following proposition.

**Proposition 2.** *Alphabetic multi-layer prefix grammars are strictly stronger*[3] *than context-free grammars and one-state Turing machines.*

*Proof.* First, we prove that the described class of the multi-layer prefix grammars is not weaker than the class of context-free grammars and the class of one-state Turing machines.

For the one-state Turing machines, the corresponding model is constructed in the proof of Theorem 1.

Given a context-free language, we consider its generating context-free grammar in the Greibach normal form [4]. Let the set of non-terminals of the grammar be $Q$ and the set of terminals be $T$. We construct a multi-layer grammar with the alphabet $\Upsilon = T \cup Q \cup \{\text{'Nil'}\} \cup \{\text{'Pop'}\}$. The initial word is $\langle S, s_0\rangle\langle\text{'Pop'}, s_0\rangle\$\langle\text{'Nil'}, s_1\rangle$ where $S$ is the initial symbol of the context-free grammar. For every rule $q_1 \to t q_2 q_3$, $q_i \in Q$, $t \in T$, we construct the rewriting rule

$$\langle q_1, s_0\rangle\Phi\$\Psi \to \langle q_2 q_3, s_0\rangle\Phi\$\Psi\langle t, s_1\rangle$$

For a rule $q_1 \to \Lambda$ we construct the rule

$$\langle q_1, s_0\rangle\Phi\$\Psi \to \Phi\$\Psi$$

Finally, we add the rule

$$\langle\text{'Pop'}, s_0\rangle\Phi\$\Psi \to \Psi\langle s_1\rangle\Phi\$\Psi\langle\hat{s_1}\rangle$$

Because all invisible letters are labelled by $s_1$ and 'Pop' is never generated elsewhere than in the initial word the last rule actually looks as $\langle\text{'Pop'}, s_0\rangle\$\Psi\langle child(s_0)\rangle \to \Psi\langle child(s_0)\rangle\$\Lambda$ and its application halts the computation.

Since alphabetic multi-layer grammars generate all context-free languages and all languages generated by one-state Turing machines, they are stronger than the one-state Turing machines (some regular languages cannot be generated by such a machine [14]).

In order to show that the alphabetic multi-layer prefix grammars are stronger than the context-free grammars, it is sufficient to show that the language $\{b^{2^n} | n \in \mathbb{N}\}$ can be generated by such a grammar (such a language cannot be generated by a context-free grammar [6]).

Let the initial word of the grammar be $\langle a, s_0\rangle\$\langle bb, s_1\rangle\langle\text{'Nil'}, s_2\rangle$, where $s_1$ and $s_2$ are children of $s_0$ incomparable with each other.

$R^{[1]} : \langle a, s_0\rangle\Phi\$\Psi\langle t\rangle\Psi\langle s_2\rangle \to \Psi\langle t\rangle\langle a, s_0\rangle\Phi\$\Psi\langle s_2\rangle$
$R^{[2]} : \langle a, s_0\rangle\$\Psi\langle t\rangle\Psi\langle s_2\rangle \to \Psi\langle t\rangle\$\Lambda$
$R^{[3]} : \langle b, t\rangle\Phi\$\Psi'\Psi\langle \text{child}(t)\rangle \to \Phi\$\Psi'\Psi\langle \text{child}(t)\rangle\langle bb, \text{child}(t)\rangle$

Consider the tree of the possible traces generated by the grammar.

[3]They generate all the languages these models can generate, and some languages that cannot be generated by these two models.

$$\begin{tikzcd}[column sep=small]
 & \langle a,s_0\rangle\$\langle bb,s_1\rangle\langle\text{'Nil'},s_2\rangle \arrow[dl] \arrow[dr] & & \\
\langle bb,s_1\rangle\$\Lambda & & \langle bb,s_1\rangle\langle a,s_0\rangle\$\langle\text{'Nil'},s_2\rangle \arrow[d] & \\
 & & \langle b,s_1\rangle\langle a,s_0\rangle\$\langle bb,\text{child}(s_1)\rangle\langle\text{'Nil'},s_2\rangle \arrow[d] & \\
 & & \langle a,s_0\rangle\$\langle bbbb,\text{child}(s_1)\rangle\langle\text{'Nil'},s_2\rangle \arrow[dl] \arrow[d] & \\
 & \langle bbbb,\text{child}(s_1)\rangle\$\Lambda & \dots \arrow[d] & \\
 & & \langle a,s_0\rangle\$\langle b^8,\text{child}(\text{child}(s_1))\rangle\langle\text{'Nil'},s_2\rangle \arrow[dl] \arrow[dr] & \\
 & \dots & & \dots
\end{tikzcd}$$

The resulting grammar $\{b^{2^n} | n \in \mathbb{N}\}$ is not even mildly context-sensitive[4] [6]. $\square$

Given a word $w = w[1]w[2]\dots w[N]$, *the inverse word of* $w$ is the word $w[N]\dots w[2]w[1]$ (denoted by $\text{inv}(w)$).

**Proposition 3.** *No rule set* $\mathbf{R}$ *exists such that all alphabetic multi-layer prefix grammars with the rule set* $\mathbf{R}$ *and the initial word arbitrarily chosen from* $w \in \{a,b\}^*$ *constructs inv*$(w)$.

*Proof.* Let $w$ be modelled by the initial word $\Phi_0\langle x,s_i\rangle\Phi_1\langle y,s_j\rangle\$\Lambda$. The word $\text{inv}(w)$ is be modelled by $\langle y,t_j\rangle\,\text{inv}(\Phi_1)\langle x,t_i\rangle\,\text{inv}(\Phi_0)\$\Lambda$. According to Proposition 1, because $\langle y,t_j\rangle = \text{deriv}(\langle y,s_j\rangle)$, $\langle x,t_i\rangle = \text{deriv}(\langle x,s_i\rangle)$, such a word cannot appear in any trace generated by any alphabetic multi-layer prefix grammar. $\square$

This proof shows that the class of the alphabetic multi-layer prefix grammars does not coincide with the classes of tree automata grammars or linear indexed grammars [6]. Informally, this class contains grammars that are able to generate very long words with a rather simple structure.

## 4 Modelling Call Stack Behaviour by Multi-layer Grammars

### 4.1 Language

In this section, we informally describe the syntax and semantics of the simple functional language $\mathbb{L}$ which is used to demonstate the modelling mechanism.

The language $\mathbb{L}$ has the call-by-name semantics. The names of the variables in $\mathbb{L}$ are the words starting with the letter `x`; other identifiers are the names of the constants (null-ary constructors) or the functions. The identifier `cons` cannot be used for a constant or a function name. There are two data types: the list set and the natural number set. The increment (+1) and decrement functions (-1) are available for the natural numbers.

[4]The class of mildly context-sensitive grammars is a special subclass of context-sensitive grammars that includes not only all context-free grammars, but also, e. g., tree adjoining grammars [7].

$\mathbb{L}$ has the two constructors: the tuple constructor, denoted as `<_,...,_>`, the ordinary list constructor `cons` with the two arguments: the first is the list head, and the second is the tail.

A definition of a function `f(x1,...,xn)` in $\mathbb{L}$ is a sequence of sentences of the form

$$\mathtt{f(T1,...,Tn)} = \mathtt{T0};$$

or

$$\mathtt{f(T1,...,Tn)} = \mathtt{h(S1,...,Sm)};$$

Here `Ti` may be a constant, a variable, a tuple of constants and variables, or `cons(Qi,Tj)` where `Qi` is a constant, a variable or a tuple containing constants or variables, and $j > n$. `Si` are expressions that can contain function calls but cannot contain the variables that are absent in `f(T1,...,Tn)`. For every left-hand side of the definition `f(T1,...,Tn)`, no variable can appear in `f(T1,...,Tn)` more than once.

The program sentences in $\mathbb{L}$ are rewriting rules. The rewriting rules in the programs are ordered from top to bottom and they should be matched in this order (such pattern matching is used in the Haskell, Refal [17] and Prolog languages).

## 4.2 Multi-layer grammars as call stack behaviour models

We borrow the notions of $f$-function and $g$-function from [15] and use them in the following sense. An $f$-function is a function whose definition contains only variables in the patterns (e. g., if `h1` is defined as $\mathtt{h1(x1,x2)} = \mathtt{x2} + \mathtt{h2(x1} + \mathtt{1)}$ then `h1` is an $f$-function). A $g$-function is a function with non-trivial patterns in the definition (e. g., $\mathtt{h1(x1} + \mathtt{1)} = \mathtt{h1(x1)} + \mathtt{h1(x1)}$ is a definition of the $g$-function).

In order to get a grammar from a program, we treat every configuration which appears while execution of the program as a tree, where nodes are named by function or constructor names and leaves contain no function calls.

All the nodes with the function calls are marked by labels from the label set **S**. If one such node in the configuration tree is a child of another, the parent node has no other children containing function calls, and the function call in the parent node is a call of $g$-function, then the parent and the child are given the same label. If some node $T$ is a descendant of a node $W$, the label of $T$ is greater (or equal) than the label of $W$. Otherwise the labels are incomparable.

For example, assuming that `h3` is a $g$-function[5], configuration $\mathtt{h1}(<\mathtt{h2(A,B)},\mathtt{h3(h2(x))}>)$ can be labelled as

$$\begin{tikzcd}
& s_0 : h_1 \arrow[d] & \\
& < \cdots > \arrow[dl] \arrow[dr] & \\
s_1 : h_2 \arrow[d] \arrow[dr] & & s_2 : h_3 \arrow[d] \\
A & B & s_2 : h_2 \arrow[d] \\
& & x
\end{tikzcd}$$

There $s_0 \lhd s_1$, $s_0 \lhd s_2$, $s_1$ and $s_2$ are incomparable.

[5]Otherwise, the innermost call of `h2` must be given a fresh label, which is a child for $s_2$.

After that, we delete all the unlabelled nodes (which have the names of static constructors) from the tree[6] and merge the nodes with the same labels. When the nodes are merged, the function names from the ancestor nodes is placed after the function names from the descendant nodes. For our configuration, after the merging we get the following diagram:

$$\begin{tikzcd}
 & s_0 : h_1 \arrow[dl] \arrow[dr] & \\
s_1 : h_2 & & s_2 : h_2 h_3
\end{tikzcd}$$

This diagram is a sketch for **G**-word representing the current call stack configuration. After the sketch is constructed, in the corresponding **G**-word the node marked by the function call to be evaluated is placed to the beginning on the visible layer, and all its ancestors (starting from the bottom) are placed after it. All the other nodes with labels incomparable with the label of the active function call are placed into invisible layer. If the name of the active call is `h1`, then the corresponding **G**-word is $\langle h1, s_0\rangle\$\langle h2, s_1\rangle\langle h2h3, s_2\rangle$. If the active call is, say, the call of `h2` in the first argument of `h1`, the corresponding **G**-word will be $\langle h2, s_1\rangle\langle h1, s_0\rangle\$\langle h2h3, s_2\rangle$.

So the visible layer of the word represents the active part of the call stack, the invisible layer represents the tree of the passive calls.

Layer functions, that are used in the multi-layer grammars (introduced in Section 3), describe one-step actions transforming passive parts of function call stacks, that can be made in $\mathbb{L}$ (and in most of other programming languages of the same sort).

1. Appending $\mathrm{App}^{s_j}[\Psi]$ models adding a $g$-function call $\Psi$ (or a sequence of such function calls) to the passive part of the configuration. None of the function calls from $\Psi$ can be evaluated immediately. E. g., if the definition of `h1` is

$$\mathtt{h1(x1+1,x2) = h1(x1,h2(h2(x2)));}$$

and the definition of `h2` is

$$\mathtt{h2(x+1) = x;}$$

then the **G**-word $\langle h_1, s_0\rangle\$\langle h_3, s_1\rangle\langle h_1, s_2\rangle$ corresponding to the configuration $\mathtt{h1(h3(x1)+1,h2(0))}$ will be transformed to $\langle h_3, s_1\rangle\langle h_1, s_0\rangle\$\langle h_2h_2h_2, s_2\rangle$, which corresponds to the pop of $\Psi\langle s_1\rangle$ from the invisible layer and an application of $\mathrm{App}^{s_2}[h_2h_2]$ to it.

2. Lifting $\mathrm{Lift}^{s_j}[\Psi\langle s_k\rangle]$ models adding a function call with a trivial pattern in the definition (an $f$-function), or a call of a $g$-function, which can be evaluated immediately, to the invisible layer. E. g., given the following definition

$$\mathtt{h1(x1+1,x2) = h1(x1,h2(x2+1));}$$

the call of `h2` can be evaluated immediately. Hence, the word $\langle h_1, s_0\rangle\$\langle h_3, s_1\rangle\langle h_2, s_2\rangle$ will be transformed by the call of `h1` to the word $\langle h3, s_1\rangle\langle h_1, s_0\rangle\$\langle h_2, s_2\rangle\langle h_2, s_3\rangle$, where $s_3 \lhd s_2$ and $s_1 \lhd s_3$. This word transformation corresponds to the pop of $\Psi\langle s_1\rangle$ from the invisible layer and an application of $\mathrm{Lift}^{s_2}[\langle h_2, s_3\rangle]$ to it.

[6]In some cases, this action can transform the tree into a forest. For example, that can happen if the configuration is `cons(h1(x),cons(h2(x),Nil))`. To avoid these cases, we always assume that the transformed tree has a root, but the root is a "virtual" function call, which is always present in the **G**-word corresponding to the tree and is denoted by $.

3. Deleting $\mathrm{Del}^{s_j}$ corresponds to replacing of one argument of the called function by an expression without function calls. E. g., let `h1` be defined as follows

$$\mathtt{h1}(\mathtt{x1}+1,\mathtt{x2}) = \mathtt{h1}(\mathtt{x1},0);$$

The word $\langle h_1, s_0\rangle\$\langle h_3, s_1\rangle\langle h_2, s_2\rangle$ will be transformed by the call of the function `h1` to the word $\langle h_3, s_1\rangle\langle h_1, s_0\rangle\$\Lambda$. This word transformation corresponds to the pop of $\Psi\langle s_1\rangle$ from the invisible layer and an application of $\mathrm{Del}^{s_2}$ to it.

4. Copying $\mathrm{Copy}^{s_j}$ corresponds to copying one argument of the called function into another. E. g., let `h1` be defined as follows

$$\mathtt{h1}(\mathtt{x1}+1,\mathtt{x2}) = \mathtt{h1}(\mathtt{x1},\mathtt{x1});$$

The word $\langle h1, s_0\rangle\$\langle h3, s_1\rangle\langle h2, s_2\rangle$ will be transformed by the call of the function `h1` to the word $\langle h_3, s_1\rangle\langle h_1, s_0\rangle\$\langle h_3, s_3\rangle$. This word transformation corresponds to the pop of $\Psi\langle s_1\rangle$ from the invisible layer and an application of $\mathrm{Copy}^{s_1}\,\mathrm{Del}^{s_2}$ to it.

Because every definition is finite, only a finite number of appendings, deletings, copyings and liftings can be done in one step of a program execution. That guarantees finiteness of the constants $K_1$ and $K_2$ in the corresponding multi-layer grammar. Absence of function calls in the patterns of function definitions in $\mathbb{L}$ makes the multi-layer grammar, that describes an $\mathbb{L}$-program, alphabetic.

## 5 Turchin's Relation and Multi-Layer Grammars

### 5.1 Turchin's theorem for multi-layer grammars

**Definition 7.** *Let* $\mathbf{G}$ *be a multi-layer prefix grammar, and* $\{\Phi_l\$\Delta_l\}$ *be a trace of* $\mathbf{G}$*-words. Let us call a suffix* $\Theta$ a permanently stable *suffix w.r.t. the segment* $\langle i,j\rangle$ *if all the words* $\Phi_k\$\Delta_k$ *in the trace, such that* $k > i$ *and* $k < j$*, are of the form* $\Phi'_k\Theta\$\Delta_k$*. If* $j$ *is not bounded,* $\Theta$ *is called* a permanently stable suffix *w.r.t.* $i$*.*

Informally, a permanently stable suffix is a suffix of the visible layer that is never changed in the trace segment starting by the $i$-th and ending by the $j$-th $\mathbf{G}$-word in the trace. In the terms of call stack behaviour, a permanently stable suffix corresponds to an unchanged context of the computation.

**Example 4.** *Let a trace of some multi-layer grammar be:*

$$\langle i, s_0\rangle\$\Lambda \rightarrow \langle abcd, s_0\rangle\$\Lambda \rightarrow \langle bcd, s_0\rangle\$\langle a, s_1\rangle \rightarrow \langle cd, s_0\rangle\$\langle ab, s_1\rangle \rightarrow \langle ab, s_1\rangle\langle d, s_0\rangle\$\Lambda$$

*Suffix* $\langle d, s_0\rangle$ *is permanently stable w.r.t. the trace segment starting from* $\langle abcd, s_0\rangle\$\Lambda$*. Suffix* $\langle b, s_1\rangle$ *is not permanently stable w.r.t. the trace segment consisting of the last two words in the trace, because in the word* $\langle cd, s_0\rangle\$\langle ab, s_1\rangle$ *it occurs in the invisible layer.*

**Definition 8.** *Let* $\mathbf{G} = \langle\Upsilon, \mathbf{S}, R, \mathfrak{F}^x_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *be a multi-layer prefix grammar. Given two* $\mathbf{G}$*-words* $\Xi_i = \Gamma_i\$\Delta_i$*,* $\Xi_j = \Gamma_j\$\Delta_j$ *in a trace* $\{\Gamma_k\$\Delta_k\}$*, we say that the words form* a Turchin pair *(denoted as* $\Xi_i \preceq \Xi_j$*) if* $\Gamma_i = \Phi\Theta_0$*,* $\Gamma_j = \Phi'\Psi\Theta_0$*,* $\Phi$ *is equal to* $\Phi'$ *as a plain word (up to the layer labels) and the suffix* $\Theta_0$ *is permanently stable on the trace segment from* $\Gamma_i\$\Delta_i$ *to* $\Gamma_j\$\Delta_j$*.*

**Theorem 2.** *Let* $\mathbf{G} = \langle\Upsilon, \mathbf{S}, \mathbf{R}, \mathfrak{F}^x_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *be a multi-layer prefix grammar. Every infinite trace generated by grammar* $\mathbf{G}$ *contains an infinite subsequence* $\Gamma_{i_n}\$\Delta_{i_n}$ *such that for every* $\Gamma_{i_{k_1}}\$\Delta_{i_{k_1}}$*,* $\Gamma_{i_{k_2}}\$\Delta_{i_{k_2}}$*,* $k_1 < k_2$ *implies* $\Gamma_{i_{k_1}} \preceq \Gamma_{i_{k_1}}$*.*

*Proof.* The idea of the proof is borrowed from the original V. Turchin's work [18]. Afterwards, in informal discussions, Andrei Klimov suggested its more formal refinement, a modification of which is presented in this paper.

Let $N$ be the maximal number of letters of the visible layer that can be changed by rewriting rules from $\mathbf{R}$. We consider the following two cases.

Let $\{\Phi_i\$\Delta_i\}$ be an infinite trace under the theorem conditions. If some word $\Gamma_i'\Theta_i\$\Delta_i$ contains a suffix $\Theta_i$ that is permanently stable w.r.t. $i$ in the trace, and no word $\Gamma_j\$\Delta_j$, $j > i$, contains a permanently stable (w.r.t. $j$) suffix longer than $\Theta_i$, then there are infinitely many words $\Phi\$\Psi$ such that $|\Phi| \leq |\Theta_i| + N$ in the trace. Thus some word $\Phi$ repeats itself as a plain word infinitely many times in the visible layer, so we can consider the subsequence of words, which have $\Phi$ as the visible part, in the trace as a subsequence, every two words of which form a Turchin pair.

Let $\{\Phi_i\$\Delta_i\}$ be an infinite trace with no upper bound on the permanently stable suffixes' length. So there is an infinite sequence of suffixes $\{\hat{\Phi}_{i_n}\}$ that are permanently stable w.r.t to positions $i_n$ of the $\mathbf{G}$-words $\{\Phi_{i_n}\$\Delta_{i_n}\}$ where these suffixes first appear. The letter of $\Phi_{i_n}$ preceding $\hat{\Phi}_{i_n}$ is not permanently stable, so it is erased somewhere in the trace $\{\Phi_i\$\Delta_i\}$. The $\mathbf{G}$-word in which it is erased looks as $\Psi_{i_n'}\hat{\Phi}_{i_n}\$\Delta_{i_n'}$, $|\Psi_{i_n'}| \leq N$. We consider the sequence $\{\Gamma_{i_n}\$\Delta_{i_n}\}$ of such $\mathbf{G}$-words. Since $|\Psi_{i_n'}|$ is bounded, there are infinitely many $\mathbf{G}$-words $\Xi_{i_n}$ with the same prefixes $\Psi_{i_n'}$ up to the layer labels. Let us consider two of the words having the same prefix, say $\Psi\Theta_{i_n}\$\Delta_{i_n}'$ and $\Psi'\Theta_{i_p}\$\Delta_{i_p}'$, where $i_n < i_p$. Both $\Theta_{i_n}$ and $\Theta_{i_p}$ are permanently stable, so $\Theta_{i_n}$ is a suffix of $\Theta_{i_p}$ (and $\Theta_{i_p} = \Xi\Theta_{i_n}$). Hence, these words have the form $\Psi\Theta_{i_n}\$\Delta_{i_n}'$ and $\Psi'\Xi\Theta_{i_n}\$\Delta_{i_p}'$, and $\Psi$ and $\Psi'$ coincide as the plain words, and $\Theta_{i_n}$ is never changed on the trace starting by $\Psi\Theta_{i_n}\$\Delta_{i_n}'$. So every two words in the subsequence $\Xi_{i_n}$ form a Turchin pair.

□

The proof of Theorem 2 implies the following corollary.

**Proposition 4.** *A product of the Turchin relation and an arbitrary well binary relation $R$ is a well binary relation on the traces generated by multi-layer prefix grammars.*

*Proof.* Every infinite trace generated by a multi-layer prefix grammar contains an infinite subsequence, every two words of which form a Turchin pair. Due to well-binariness of $R$, this subsequence also contains two words $\Gamma$ and $\Delta$ such that $\Gamma$ precedes $\Delta$ and $\langle\Gamma,\Delta\rangle \in R$. □

The proof of Theorem 2 also implies the following corollary.

**Proposition 5.** *Let $\mathbf{G} = \langle\Upsilon, \mathbf{S}, R, \mathfrak{F}^x_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ be a multi-layer prefix grammar. Every infinite trace generated by $\mathbf{G}$ contains two words $\Gamma_i\$\Delta_i$, $\Gamma_j\$\Delta_j$ ($i < j$) such that $\Gamma_i\$\Delta_i \preceq \Gamma_j\$\Delta_j$ and $|\Gamma_i| < |\Gamma_{i+1}|$ and $|\Gamma_j| < |\Gamma_{j+1}|$.*

Informally Proposition 5 states that when we search for Turchin pairs in a trace, we can skip terms before shortenings of the visible layer. However one can notice that a refinement that was suggested in [13] for the plain prefix grammars can generate infinite bad sequences in computations over multi-layer prefix grammars. The refinement tests for the Turchin relation only the pairs of the words generated by the rules not shortening the visible layer of a word.

**Example 5.** *Let $\mathbf{G}$ be determined by the initial word $\langle tpqe, s_0\rangle\$\Lambda$ and the rewriting rules*

$R^{[1]} : \langle p, s_i\rangle\Phi\$\Psi'\Psi\langle child(s_i)\rangle \rightarrow \Phi\$\Psi'\Psi\langle pp, child(s_i)\rangle$
$R^{[2]} : \langle q, s_i\rangle\Phi\$\Psi'\Psi\langle child(s_i)\rangle \rightarrow \Phi\$\Psi'\Psi\langle qq, child(s_i)\rangle$
$R^{[3]} : \langle e, s_i\rangle\Phi\$\Psi'\Psi\langle child(s_i)\rangle \rightarrow \Psi\langle child(s_i)\rangle ee\langle s_i\rangle\Phi\$\Psi'$
$R^{[4]} : \langle t, s_i\rangle\Phi\$\Psi'\Psi\langle child(s_i)\rangle \rightarrow \Phi\$\Psi'\Psi\langle child(s_i)\rangle\langle tt, child(s_i)\rangle$

*The tree of the traces over* **G** *looks as follows:*

$$\begin{tikzcd}
\langle tpqe, s_0\rangle\$\Lambda \arrow[d] \\
\langle pqe, s_0\rangle\$\langle tt, child(s_0)\rangle \arrow[d] \\
\langle qe, s_0\rangle\$\langle ttpp, child(s_0)\rangle \arrow[d] \\
\langle e, s_0\rangle\$\langle ttppqq, child(s_0)\rangle \arrow[d] \\
\langle ttppqq, child(s_0)\rangle\langle ee, s_0\rangle\$\Lambda \arrow[d] \\
\langle tppqq, child(s_0)\rangle\langle ee, s_0\rangle\$\langle tt, child(child(s_0))\rangle \arrow[d] \\
\ldots \arrow[d] \\
\langle ee, s_0\rangle\$\langle ttttppppqqqq, child(child(s_0))\rangle \arrow[d] \\
\langle ttttppppqqqq, child(child(s_0))\rangle\langle eee, s_0\rangle\$\Lambda \arrow[d] \\
\ldots
\end{tikzcd}$$

*The only visible layers that satisfy the condition* $|\Gamma_i| \geq |\Gamma_{i-1}|$ *are* $\langle t^{2^k} p^{2^k} q^{2^k}, s_i\rangle\langle e^k, s_0\rangle$. *For all* $k > 0$, *no two distinct words with such visible layers form a Turchin pair.*

This example shows another subtle feature of the Turchin relation over multi-layer computations. For the traces generated by the plain prefix grammars the first pair of homeomorphically embedded words also form a Turchin pair. So in some sense these two relations have the same effect when they are used as termination criteria. But as we can see in Example 5, this proposition fails for the traces generated by the multi-layer grammars.

## 5.2 Bounds on the bad sequences length

We recall that $\Psi\langle\hat{s_j}\rangle$ is the maximal subsequence of $\Psi$ not containing letters labelled by $s_j$.

**Definition 9.** *Let a multi-layer grammar* **G** *contain the rule*

$$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Psi\langle s_j\rangle\Phi\langle s_i\rangle\Theta\$F^{s_i}(\Psi\langle\hat{s_j}\rangle).$$

An imbedded prefix $\Delta$ *is a plain word corresponding to* **G**-*word* $\Phi\langle s_i\rangle$ *or any derivative of* $\Xi$ *generated by* $F^{s_i}(\Psi\langle\hat{s_j}\rangle)$.

*If* $\{\Gamma_i\$\Delta_i\}_{i=0}^{k}$ *is a trace segment generated by* **G**, *we call* a free imbedded prefix $\Theta$ *such an imbedded prefix that* $\Theta\Psi\Gamma_k$ *forms no Turchin pairs with* $\Gamma_i$ *(*$i \leq k$*).*

**Example 6.** *Let* $\mathbf{G}_{bin}$ *have the initial word* $\langle s, s_0\rangle\$\langle\text{'}E\text{'}, s_1\rangle$ *and the rewriting rules*

$R^{[1]} : \langle m,x\rangle\Phi\$\Psi \to \langle pmp,x\rangle\Phi\$\Psi$
$R^{[2]} : \langle m,x\rangle\Phi\$\Psi \to \Psi\langle y\rangle\langle pmp,x\rangle\Phi\$\Psi\langle\hat{y}\rangle$, $y = child(x)$
$R^{[3]} : \langle p,x\rangle\Phi\$\Psi \to \Phi\$\Psi$
$R^{[4]} : \langle p,x\rangle\Phi\$\Psi \to \Phi\$\Psi\langle p, child(x)\rangle$

*Imbedded prefixes in $\mathbf{G}_{bin}$ are pmp (generated by $R^{[1]}$ and $R^{[2]}$) and p (generated by $R^{[4]}$).*

**Lemma 1.** *Let us consider the Ackermann function defined as follows: $B_K(M,0) = 1$, $B_K(0,N) = N+1$, $B_K(M,N) = B_K(M-1, B_K(M,N-1)*K)$. An alphabetic multi-layer grammar $\mathbf{G}$ can generate bad sequences w.r.t. the Turchin relation not longer than $B_K(M,N)$, where $K$ is a maximal length of an imbedded prefix in $\mathbf{G}$ (the maximal rewrite depth of $\mathbf{G}$), $M$ is the number of the imbedded prefixes in the set of rewriting rules of $\mathbf{G}$ and $N$ is the total length of the initial word in the grammar.*

*Proof.* Let the initial word of $\mathbf{G}$ contain only unique letters (that do not occur in imbedded prefixes of the rewriting rules of $\mathbf{G}$)[7]. Then for every two words $\Gamma_i\$\Delta_i$, $\Gamma_j\$\Delta_j$ in a trace, if $\Gamma_i$ and $\Gamma_j$ contain different letters belonging to the visible layer of the initial word, they cannot form a Turchin pair. More formally, if the visible layer of the initial word is $\langle\Phi_0[1],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle$, then a word having suffix $\langle\Phi_0[i],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle$ cannot form a Turchin pair with a word having suffix $\langle\Phi_0[i+k],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle$. We assume that a trace containing a bad sequence contains all of the words

$$
\begin{array}{l}
\langle\Phi_0[1],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle\$\Delta_0\\
\langle\Phi_0[2],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle\$\Delta_{i_2}\\
\ldots\\
\langle\Phi_0[N-1],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle\$\Delta_{i_{N-1}}\\
\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}
\end{array}
$$

If some word $\langle\Phi_0[j],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle\$\Delta_{i_j}$ does not occur in the trace, then $\langle\Phi_0[j],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle$ is a permanently stable suffix with respect to the whole trace, so $\langle\Phi_0[j],s_0\rangle\ldots\langle\Phi_0[N],s_0\rangle$ can be deleted from all the words in the trace with no impact on the bad sequence length.

If there are no imbedded prefixes (or all of them are never free), then we can only erase letters and the longest bad sequence consists of $N+1$ words starting from the initial word and ending with $\Lambda$.

If the initial word has the length 0, then it is $\Lambda\$\Lambda$ and no rules can be applied to it. The longest bad sequence has the length 1 and contains only the initial word.

Now we make the induction step. Let the length of the initial word be $N$, the number of imbedded prefixes be $M$. For $M_1 < M$ and arbitrary $N_2$, let the length of the longest bad sequence built on the initial word of the length $N_2$ with $M_1$ free imbedded prefixes be not more than $B_K(M_1,N_2)$. For $N_1 < N$, let the length of the longest bad sequence built on the word of the length $N_1$ with $M$ free imbedded prefixes be not more than $B_K(M,N_1)$.

First, we assume that $\Delta_0 = \Lambda$.

A trace which is a candidate for a bad sequence consists of the two following segments. The first segment — say, segment $\sigma_1$ — starts with the initial word and ends with $\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}$. The second segment, $\sigma_2$, starts from the word $\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}$ and ends with a word having a Turchin pair. Whatever the rule applications on the segment $\sigma_1$ are, no word from $\sigma_1$ can form a Turchin pair with a word from $\sigma_2$ because of the properties of the initial word. So we can generate a longest possible bad sequence $\sigma_1$ and then combine it with a longest possible bad sequence $\sigma_2$ to receive a longest possible bad sequence on the whole trace.

The word $\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}$ can be transformed either by an application of a pop rule or by replacing $\langle\Phi_0[N],s_0\rangle$ by some imbedded prefix $\Xi$. Since $\Xi$ replaces the last letter of the initial word, it becomes not free for all the segment $\sigma_2$. If $\Xi$ is a prefix of a word in $\Delta_{i_N}$ which is placed in the visible layer, it becomes not free until it is erased. Therefore, a bad sequence with an application of a pop rule to $\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}$

[7]Otherwise, some accidental Turchin pairs may appear in a trace.

is not shorter than a bad sequence with an application of a non-pop rule, if $\Delta_{i_N}$ contains words not shorter than $K$.

The longest possible bad sequence $\sigma_1$ is not longer than $B_K(M,N-1)$, where $M$ is the number of imbedded prefixes in the rewriting rules of $\mathbf{G}$. $\Delta_{i_N}$ can contain words $\Psi\langle s_i\rangle$ of the length not more than $(B_K(M,N-1)-1)*K$ (is is possible to append not more than $k$ letters to the invisible layer during a rule application, and there are $B_K(M,N-1)-1$ such applications on $\sigma_1$). So the longest possible visible layer after an application of a pop rule to $\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}$ contains not more than $B_K(M,N-1)*K$ letters ($K$ more letters can be added on the visible layer by the rule). Let us denote the plain word corresponding to this visible layer as $\Phi_0'$.

After $\Phi_0'$ is placed in the visible layer, all the other invisible words in $\Delta_{i_N}$ cannot appear in the visible layer, which is shown in Proposition 1. So we can consider the word which follows $\langle\Phi_0[N],s_0\rangle\$\Delta_{i_N}$ as an equivalent of the word $\Phi_0'\langle t_w\rangle\$\Lambda$.

For every word $\Phi\langle\Phi_0'[j],t_w\rangle\ldots\langle\Phi_0'[w],t_w\rangle\$\Delta_i$ ($|\Phi|$¿0) in $\sigma_2$, at least one imbedded prefix is not free. Namely, the imbedded prefix which contains $\Phi_0'[j]$ and which application generated $\Phi_0'[j]$ in the word $\Phi_0'$, is not free. This prefix corresponds to the plain word $\Psi_1\Phi_0'[j]\Psi_2$ where $\Psi_1$ and $\Psi_2$ are some words, and $\Psi_2$ coincides with some prefix of the word $\Phi_0'[j+1]\ldots\Phi_0'[w]$. Somewhere in $\sigma_2$ before the word $\Phi\langle\Phi_0'[j],t_j\rangle\ldots\langle\Phi_0'[w],t_w\rangle\$\Delta_i$, is the word $\Theta_j=\langle\Psi_1\Phi_0'[j],t_j\rangle\ldots\langle\Phi_0'[w],t_w\rangle\$\Delta'$. After changing $\Phi\langle\Phi_0'[j],t_j\rangle$ to some word having the prefix $\Psi_1\Phi_0'[j]\Psi_2$, we get a Turchin pair with $\Theta_j$.

Hence, the longest bad sequence that can be constructed by the grammar $\mathbf{G}$ on the word $\Phi_0'\langle t_w\rangle\$\Lambda$ has the length not more than $B_K(M-1,|\Phi_0'|)$. So the bad sequence constructed on the initial word $\Phi_0\langle s_0\rangle\$\Lambda$ is not longer than $B_K(M-1,B_K(M,N-1)*K)$.

If $\Delta_0$ is not $\Lambda$, then for every $\Delta_0\langle s_i\rangle$, we can build some bad sequence on it as a prefix of a visible layer. The longest bad sequence on the prefix $\Delta_0\langle s_i\rangle$ can be build if we have all the $M$ imbedded prefixes as free. So we consider only bad sequences where $\Delta_0\langle s_i\rangle$ can be placed on the visible layer as early as possible — they are the longest possible bad sequences. And the longest of them are not longer than the longest possible bad sequence constructed from the initial word $\Delta_0\Phi_0\$\Lambda$ (because of the presence of the letter $\Phi_0[1]$ in the word).

To show that such bad sequences exist, we construct a grammar $\mathbf{G}=\langle\Upsilon,\mathbf{S},\mathbf{R},\mathfrak{F}^x_{N,K},\Gamma_0\$\Delta_0\rangle$ such that all rules in $\mathbf{R}$ are either

$$R^{[l]}:\langle a_l,s_i\rangle\Theta\$\Psi\to\Phi_l\langle s_i\rangle\Theta\$F^{s_i}(\Psi)$$

or

$$R^{[l]}:\langle a_l,s_i\rangle\Theta\$\Psi\to\Psi\langle s_j\rangle\Phi_l\langle s_i\rangle\Theta\$F^{s_i}(\Psi\langle\hat{s_j}\rangle)$$

and $|\Phi_l|$ is either a number greater than 1 (say, $|\Phi_l|=K$) or 0. If $\Phi_l=\Lambda$, then $F^{s_i}$ is always a composition of $N$ appending operations $\mathrm{App}^{s_k}[\Xi]$, where $s_k$ are incomparable children of $s_i$ ($s_k$ is a fresh label from $\mathbf{S}$ if $\Psi_1$ contains no children of $s_i$), and $|\Xi|=|\Phi_j|$ for some imbedded prefix $\Phi_j$ in $\mathbf{R}$.

In order to construct an Ackermanian-long bad sequence containing no Turchin pairs, we use the following strategy.

1. Apply all the rules $R^{[j]}$ with the different imbedded prefixes in the right-hand sides of the visible layers to the initial word.

2. (a) If applications of all imbedded prefixes $\Phi_j$ generate Turchin pairs (no free imbedded prefixes are available on the trace), apply a rule with $\Phi_j=\Lambda$.

(b) If there is an imbedded prefix $\Phi_j$ that can be prepended to the visible layer of the word without generating a Turchin pair, then use it and pop $(\Phi_j)^k$ from the invisible part[8].

Proceed with the step (1) until the empty word or a Turchin pair is generated.

This strategy constructs a trace which ends with a segment starting with the longest word in the trace $(\Phi_j)^k\$\Psi'$, the letters of the visible layer of which are erased until the word $\Lambda\$\Psi$ is reached.

Let us illustrate the construction above with a grammar of the described form where $K = 2$, and the length of the initial word is also 2.

The initial word is $\langle a^n, s_0\rangle\$\Lambda$. The rewriting rules are

$\langle a,s\rangle\Theta\$\Psi \to \langle bb,s\rangle\Theta\$\Psi$

$\langle a,s\rangle\Theta\$\Psi'\Psi\langle child_1(s)\rangle \to \Psi\langle child_1(s)\rangle\langle bb,s\rangle\Theta\$\Psi\,\mathrm{App}^{child_1(s)}[bb]\,\mathrm{App}^{child_2(s)}[cc]$

$\langle b,s\rangle\Theta\$\Psi \to \langle cc,s\rangle\Theta\$\Psi$

$\langle b,s\rangle\Theta\$\Psi'\Psi\langle child_2(s)\rangle \to \Psi\langle child_2(s)\rangle\langle cc,s\rangle\Theta\$\Psi\,\mathrm{App}^{child_1(s)}[bb]\,\mathrm{App}^{child_2(s)}[cc]$

$\langle c,s\rangle\Theta\$\Psi \to \Theta\$\Psi\,\mathrm{App}^{child_1(s)}[bb]\,\mathrm{App}^{child_2(s)}[cc]$

A trace of the grammar built by the described strategy is presented below. We assume that $s_\Gamma \triangleleft s_\Delta$ iff the word $\Gamma$ is a prefix of the word $\Delta$. Otherwise, $s_\Gamma$ and $s_\Delta$ are incomparable.

[8]We cannot prescribe a rule to pop $\Phi_j^k$ from the invisible part, but we can assign the label $s_{\Phi_j}$ to all $\Phi_j$ placed to the invisible layer, and then we can pop $\Psi\langle s_{\Phi_j}\rangle$ from it.

$$\begin{tikzcd}
\langle aa, s_0\rangle \$\Lambda \arrow[d] \\
\langle bba, s_0\rangle \$\Lambda \arrow[d] \\
\langle ccba, s_0\rangle \$\Lambda \arrow[d] \\
\langle cba, s_0\rangle \$\langle b^2, s_{01}\rangle\langle c^2, s_{02}\rangle \arrow[d] \\
\langle ba, s_0\rangle \$\langle b^4, s_{01}\rangle\langle c^4, s_{02}\rangle \arrow[d] \\
\langle c^4, s_{02}\rangle\langle c^2a, s_0\rangle \$\Lambda \arrow[d] \\
\langle c^3, s_{02}\rangle\langle c^2a, s_0\rangle \$\langle b^2, s_{021}\rangle\langle c^2, s_{022}\rangle \arrow[d] \\
\langle c^2, s_{02}\rangle\langle c^2a, s_0\rangle \$\langle b^4, s_{021}\rangle\langle c^4, s_{022}\rangle \arrow[d] \\
\langle c, s_{02}\rangle\langle c^2a, s_0\rangle \$\langle b^6, s_{021}\rangle\langle c^6, s_{022}\rangle \arrow[d] \\
\langle c^2a, s_0\rangle \$\langle b^8, s_{021}\rangle\langle c^8, s_{022}\rangle \arrow[d] \\
\langle ca, s_0\rangle \$\langle b^{10}, s_{021}\rangle\langle c^{10}, s_{022}\rangle \arrow[d] \\
\langle a, s_0\rangle \$\langle b^{12}, s_{021}\rangle\langle c^{12}, s_{022}\rangle \arrow[d] \\
{}
\end{tikzcd}$$

$$\begin{tikzcd}
\langle b^{12}, s_{021}\rangle\langle b^2, s_0\rangle \$\Lambda \arrow[d] \\
\langle c^2b^{11}, s_{021}\rangle\langle b^2, s_0\rangle \$\Lambda \arrow[d] \\
\langle cb^{11}, s_{021}\rangle\langle b^2, s_0\rangle \$\langle b^2, s_{0211}\rangle\langle c^2, s_{0212}\rangle \arrow[d] \\
\langle b^{11}, s_{021}\rangle\langle b^2, s_0\rangle \$\langle b^4, s_{0211}\rangle\langle c^4, s_{0212}\rangle \arrow[d] \\
\langle c^6, s_{0212}\rangle\langle b^{10}, s_{021}\rangle\langle b^2, s_0\rangle \$\Lambda \arrow[d] \\
\dots \arrow[d] \\
\langle b^{10}, s_{021}\rangle\langle b^2, s_0\rangle \$\langle b^{12}, s_{02121}\rangle\langle c^{12}, s_{02122}\rangle \arrow[d] \\
\langle c^{14}, s_{02122}\rangle\langle b^9, s_{021}\rangle\langle b^2, s_0\rangle \$\Lambda \arrow[d] \\
\dots \arrow[d] \\
\langle c^{2^i-2}, s_{02122\dots2}\rangle\langle b^{13-i}, s_{021}\rangle\langle b^2, s_0\rangle \$\Lambda \arrow[d] \\
\dots \arrow[d] \\
\langle c^{2^{15}-2}, s_{02122222\dots2}\rangle \$\Lambda
\end{tikzcd}$$

Until the last word is completely erased, no words in the trace form Turchin pairs.

The set of the rules above allows us to generate bad sequences with the length order $O(B(n,2))$ ($n$ is the length of the initial word). If we add the rules

$\langle c, s\rangle\Theta\$\Psi \to \langle dd, s\rangle\Theta\$\Psi$

$\langle c, s\rangle\Theta\$\Psi'\Psi\langle child_3(s)\rangle \to \Psi\langle child_3(s)\rangle\langle dd, s\rangle\Theta\$\Psi\,\mathrm{App}^{child_1(s)}[bb]\,\mathrm{App}^{child_2(s)}[cc]\,\mathrm{App}^{child_3(s)}[dd]$

$\langle d, s\rangle\Theta\$\Psi \to \Theta\$\Psi App^{child_1(s)}[bb]\,\mathrm{App}^{child_2(s)}[cc]\,\mathrm{App}^{child_3(s)}[dd]$

to the grammar (so the imbedded prefix $dd$ is added to the set of imbedded prefixes $\{cc, bb\}$), the strategy will generate words of the length $B(n,3)$, and so on.

□

We can also generate Ackermanian languages by the grammars of the described sort.

**Example 7.** *Let* $\mathbf{G_{exptower}}$ *be the following grammar. Its initial word is*

$$\langle A, s_0\rangle\langle E, s_0\rangle\$\langle Bb, s_{01}\rangle\langle, s_{02}\rangle\langle\text{'Nil'}, s_{03}\rangle$$

*and* $s_{03}$ *is a child of* $s_0$ *which is incomparable with all other labels in* $\mathbf{S}$*. The set* $\mathbf{R}$ *of rewriting rules is:*

$$\begin{array}{lll}
R^{[0]} : \langle A, s_i\rangle\Theta\$\Psi & \rightarrow & \langle Aa, s_i\rangle\Theta\$\Psi \\
R^{[1]} : \langle A, s_i\rangle\Theta\$\Psi & \rightarrow & \langle Bb, s_i\rangle\Theta\$\Psi \\
R^{[2]} : \langle a, s_i\rangle\Theta\$\Psi\langle s_{i1}\rangle\Psi\langle s_{i2}\rangle\Psi' & \rightarrow & \Psi\langle s_{i1}\rangle\Theta\$\Psi' \\
R^{[3]} : \langle B, s_i\rangle\Theta\$\Psi & \rightarrow & \langle Cc, s_i\rangle\Theta\$\Psi \\
R^{[4]} : \langle b, s_i\rangle\Theta\$\Psi\langle s_{i1}\rangle\Psi\langle s_{i2}\rangle\Psi' & \rightarrow & \Psi\langle s_{i2}\rangle\Theta\$\Psi' \\
R^{[5]} : \langle , s_i\rangle\Theta\$\Psi & \rightarrow & \Theta\$\Psi\langle Bb, s_{i1}\rangle\langle Cc, s_{i2}\rangle \\
R^{[6]} : \langle c, s_i\rangle\Theta\$\Psi & \rightarrow & \Theta\$\Psi\langle Bb, s_{i1}\rangle\langle Cc, s_{i2}\rangle \\
R^{[7]} : \langle E, s_0\rangle\$\Psi\langle s_{i1}\rangle\Psi\langle s_{i2}\rangle\langle\text{'}Nil\text{'}, s_3\rangle & \rightarrow & \Psi\langle s_{i2}\rangle\$\Lambda
\end{array}$$

*After the rule $R^{[7]}$ is applied, the second descendant of $s_0$ is placed into the visible part and the invisible part of the word becomes $\Lambda$. Each application of the rule $R^{[0]}$ adds a level to the tower of exponentials which is the length of the visible part of this word, i.e. the length of words in the generated language is $2^{2^{\cdots^{2}}}\}^{N}$.*

**Example 8.** *A program that generates traces similar to traces generated by $\mathbf{G_{exptower}}$ (the grammar from Example 7) can be as follows. The input point of the program is* `A(<N,b(B(<1,0>))>)` *(where* `N` *is an arbitrary fixed natural number).*

```
A(<x1+1,x2>)=a(A(<x1,x2>));
A(<0,x2>)=<x2+1,0>;
a(<x1+1,x2>)=x1;
B(<x1+1,x2>)=c(c(<x1+1,x2>));
b(<x1+1,x2>)=x2;
c(<x1+1,x2>)=<B(b(<x1,x2>))+1, c(c(<x1,x2>))>;
c(<0,x2>)=<B(b(<1,0>))+1, c(c(<0,0>))>;
```

*The program never stops and its call stack configurations form bad sequences of the exponential tower length (in $N$) with respect to the Turchin relation. However, with respect the homeomorphic embedding over the entire terms, a computation of this program for every $N > 0$ is terminated on the $5+N$-th step.*

## 6 Conclusion

The Turchin relation for call-by-name computations is a strong and consistent branch termination criterion, which finds pairs of embedded terms on the trace of every infinite computation. It allows a program transformation tool to construct very long configuration sequences (i.e., traces) with no Turchin pairs in them, and neither the homeomorphic embedding can replace the Turchin relation nor the Turchin relation can be considered as a simplification of the homeomorphic embedding in the case of the normal-order reduction. The Turchin relation can be used together with the homeomorphic embedding without the loss of well-binariness.

Alphabetic multi-layer grammars, which describe function call stack behaviour, are able to generate languages with very long words, but it seems they are not able to generate languages with words having a complex structure. It would be interesting to find some practical problems, which can be solved with the power of Turchin's relation (or homeomorphic embedding) on the call stack configurations for call-by-name computations.

## Acknowledgements

I would like to thank A. P. Nemytykh for many fruitful advices and help in improving the paper, and the anonymous referees for the useful feedback.

## References

[1] A. Ahmed, A. Lisitsa & A. Nemytykh (2013): *Cryptographic Protocol Verification via Supercompilation (A Case Study)*. In Alexei Lisitsa & Andrei Nemytykh, editors: *VPT 2013*, *EPiC Series* 16, EasyChair, pp. 16–29.

[2] D. Caucal (1992): *On the regular structure of prefix rewriting*. *Theoretical Computer Science* 106, pp. 61–86.

[3] D. Dolev & A.C. Yao (1983): *On the security of public key protocols*. *Transactions on Information Theory* 29, pp. 198–208.

[4] S. Greibach (1965): *A New Normal-Form Theorem for Context-Free Phrase Structure Grammars*. *Journal of the ACM* 12(1).

[5] G. W. Hamilton & N. D. Jones (2012): *Distillation with labelled transition systems*, pp. 15–24. IEEE Computer Society Press.

[6] J. E. Hopcroft & J. D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

[7] A.K. Joshi, K.V. Shanker & D. Weir (1990): *The Convergence of Mildly Context-Sensitive Grammar Formalisms*. Technical Report.

[8] I. Klyuchnikov (2014): *Nullness Analysis of Java Bytecode via Supercompilation over Abstract Values*. In: *Fourth International Valentin Turchin Workshop on Metacomputation*, pp. 161–176.

[9] A. Lisitsa & A. Nemytykh (2014): *A Note on Program Specialization. What Syntactical Properties of Residual Programs Can Reveal?*, pp. 52–65. 28, EPiC Series, EasyChair.

[10] A. Lisitsa & A. P. Nemytykh (2008): *Reachability Analysis in Verification via Supercompilation*. *International Journal of Foundations of Computer Science* 19(4), pp. 953–970.

[11] A. P. Nemytykh (2007): *The Supercompiler Scp4: General Structure*. URSS, Moscow.

[12] A. Nepeivoda (2013): *Ping-Pong protocols as prefix grammars and Turchin's relation*. In: *VPT 2013. First International Workshop on Verification and Program Transformation*, 16, EPiC Series, EasyChair, pp. 74–87.

[13] A. Nepeivoda (2014): *Turchin's Relation and Subsequence Relation in Loop Approximation*. In: *PSI 2014. Ershov Informatics Conference. Poster Session*, 23, EPiC Series, EasyChair, pp. 30–42.

[14] Y. Saouter (1995): *Halting Problem for One-State Turing Machines*. Research Report.

[15] M.H. Sørensen (1994): *Turchin's Supercompiler Revisited*. Ms.Thesis.

[16] V. F. Turchin (1986): *The Concept of a Supercompiler*. *ACM Transactions on Programming Languages and Systems* 8(3), pp. 292–325.

[17] V. F. Turchin (1989): *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts. Electronic version:`http://www.botik.ru/pub/local/scp/refal5/`.

[18] V.F. Turchin (1988): *The algorithm of generalization in the supercompiler*. *Partial Evaluation and Mixed Computation*, pp. 341–353.