Complexity of Turchin's Relation for Call-by-Name Computations

(Extended Abstract)

Antonina Nepeivoda
a_nevod@mail.ru

Program Systems Institute of Russian Academy of Sciences*
Pereslavl-Zalessky, Russia

Abstract. Supercompilation is a program transformation technique first described by V.F. Turchin in the 1970s. In supercompilation, Turchin's relation on call-stack configurations is used both for call-by-value and call-by-name semantics as a whistle. We give a formal grammar model of call-by-name stack behaviour and find the worst-case number of driving steps before the whistle using Turchin's relation is blown.

1 Introduction

Supercompilation is a program transformation method based on fold/unfold operations [2, 12, 14]. Given a program and its parametrized input configuration, a supercompiler partially unfolds the process tree of the program on the input configuration. Then it tries to fold the tree back into a graph, which presents the residual program. In general case, the process tree may be infinite. Thus, the following question appears: when is it reasonable to stop the unfolding in order to avoid going into an infinite loop?

One of the ways to solve this problem is based on "configuration similarity" relations. If a path in the tree contains two configurations, the latter of which resembles the former, that may be a sign that the path represents an unfolded loop. Thus, when a supercompiler finds two such configurations, it terminates unfolding of the path where they appear.

We recall an important relation property used for termination [5].

Definition 1 *Given a set T of terms and a set S of sequences of the terms from T, relation $R \subset T \times T$ is called* a well binary relation with respect to set S, *if every sequence $\{\Phi_n\} \in S$ such that $\forall i, j (i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is finite.*

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.*

* The reported study was partially supported by RFBR, research project No. 14-07-00133, and Russian Academy of Sciences, research project No. AAAA-A16-116021760039-0.

So, a well binary relation is "a well quasi-order without the order" (i. e., it is not necessarily transitive).

Any relation guaranteeing termination of the unfolding of a process tree must be a well binary relation with respect to the set of the traces generated in the tree. The relation most widely used for this aim, the homeomorphic embedding [1, 5, 10], is well binary with respect to arbitrary term sequences [4]. Some other relations used for termination in program transformations[1] are not well binary with respect to arbitrary term sequences. However, they are well binary with respect to the term sequences that can be generated on any computation path. In order to study termination properties of such relations, one must consider them together with their domain. Hence, the usual way of reasoning about the well-binariness of a relation as in [6][2] meets a couple of problems.

For Turchin's relation, well-binariness of which also can be proved only with respect to computation paths that appear during unfolding. That relation on call-stack configurations was the first well binary relation used for trace termination [14] (1986). Although Turchin's relation is a useful tool that helps to solve both termination and generalization problems [16], the proof of its well-binariness given by V. Turchin in [16] was presented in a semi-formal way. This work presents a formal approach to the theorem for computations in the call-by-name semantics. We introduce a notion of a multi-layer prefix grammar[3]. Elements of traces generated by such a grammar are models of call-stack configurations on computation paths in the call-by-name semantics. Based on the formalization, we could give a constructible proof of Turchin's theorem for the grammars being introduced. The constructible proof allowed us to find the upper bound on the bad sequence length with respect to Turchin's relation. The upper bound is Ackermanian.

In Section 2, we give an example showing how Turchin's relation can be efficiently used as a whistle. In Section 3 we give the formal definition for a class of grammars that model call stack behaviour for call-by-name computations. In Section 4 we very briefly show how such grammars can be used for modelling the call stack behaviour of programs in a simple functional language. Finally, in Section 5 we refine the definition of Turchin's relation for the new class of the grammars and state the result on the upper bound.

2 Turchin's Relation: an Example

The following program computes the sum of the squares starting from n down to 1 in a straightforward way.

[1] Among them is the relation used in supercompiler SCP4 [7] and the relation used in higher-order supercompiler HOSC [3].

[2] Including the "minimal bad sequence" method or methods using infinite Ramsey theorem.

[3] A precise description of the multi-layer grammars and their connection to call-stack configurations will be published in [9].

Example 1

A program computing $\sum_{k=1}^{n} k^2$
Start: $s(x)$; $s(0) = 0$; $s(x+1) = a(m(x+1, x+1), s(x))$; $a(0, y) = y$; $a(x+1, y) = a(x, y) + 1$; $m(0, y) = 0$; $m(x+1, y) = a(y, m(x, y))$;

When using a "core" homeomorphic embedding whistle, a supercompiler generates the residual program which repeats the initial program modulo renaming.

The program generated by a supercompiler which uses the composition of Turchin's relation and homeomorphic embedding as a whistle, is below.

Example 2

Residual program computing $\sum_{k=1}^{n} k^2$ *when Turchin's relation is used*
Start: $f(x, x, x)$; $f(0, 0, 0) = 0$; $f(0, 0, x+1) = f(x, x, x) + 1$; $f(0, x+1, y) = f(y, x, y) + 1$; $f(x+1, y, z) = f(x, y, z) + 1$;

The program given in Example 2 is more efficient than the initial one given in Example 1: on every step except the very last it adds 1 to the result, thus avoiding "zero" steps such as $m(0, y) = 0$; or $a(0, y) = y$;. What properties of the refined whistle made this result possible?

The part of the process tree of the program in Example 1 is shown in Figure 1. When we use the homeomorphic embedding relation as a whistle when driving the initial configuration $s(x)$ to $s(x_1 + 1)$, the whistle is blown immediately after the substitution of the narrowing $x \to x + 1$ to the right-hand side of the definition $s(x+1) = a(m(x+1, x+1), s(x))$ because the subterm $s(x_1)$ repeats the initial configuration modulo the variables' names.

After the generalization to **let** $z = s(x_1)$ **in** $a(m(x_1 + 1, x_1 + 1), z)$, the process tree of the program is unfolded until the narrowing $x_1 \to x_2 + 1$ is done. Then the term $a(a(x_2, m(x_2 + 1, x_2 + 2)) + 1, z) + 1$, which is the result of the substitution of the narrowing in the term $a(a(x_1, m(x_1, x_1 + 1)), z) + 1$, embeds the parent term or its ancestor $a(m(x_1+1, x_1+1), z)$ (depending on the strategy of a supercompiler). The msg for both the pairs is $a(u, z)$. That is the cause why the residual program coincides with the one given in Example 1.

As opposed to the homeomorphic embedding relation, Turchin's relation considers only flat call-stack structure, ignoring all the passive data. In Figure 1,

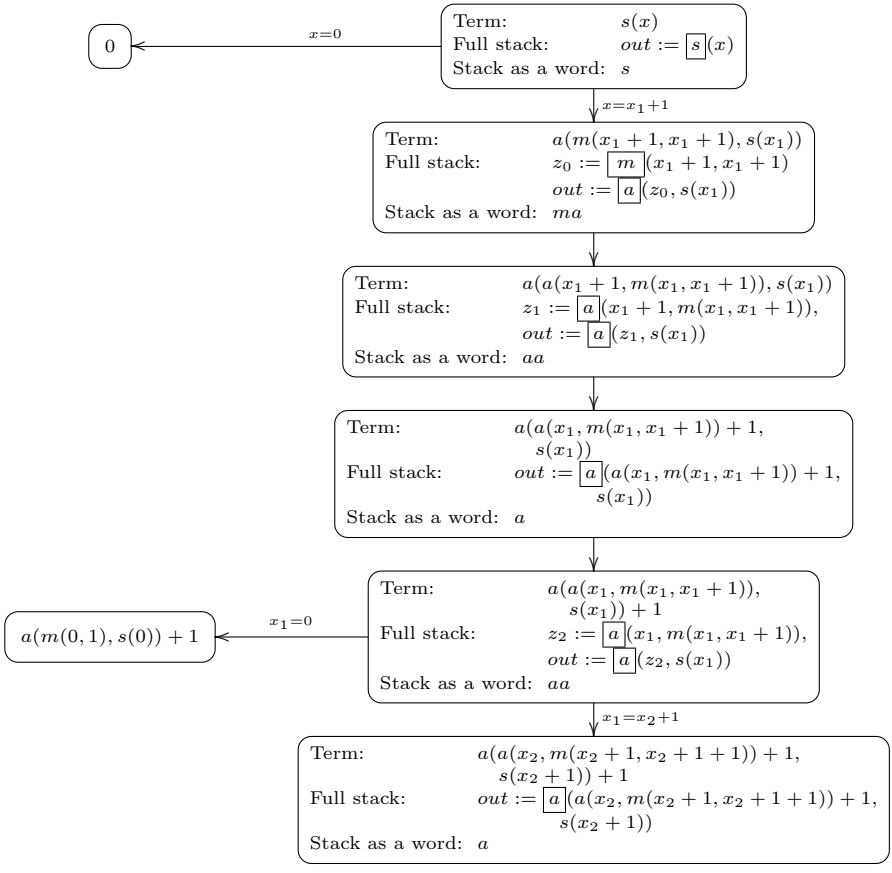

Fig. 1. A fragment of the process tree for the program of Example 1

the call-stack configurations are presented in the two forms: the full form that is constructed while interpreting the configuration, and the "word" form which contains *only* the names of the functions in the call-stack. Turchin's relation operates with the "word" forms. Namely, it checks whether the two "word forms" of the call stacks Δ_1 and Δ_2 on the path can be split into parts $[Top]$, $[Middle]$, and $[Context]$ such that $\Delta_1 = [Top][Context]$, $\Delta_2 = [Top][Middle][Context]$ and the part $[Context]$ is never changed on the path segment starting at Δ_1 and ending at Δ_2 (as it is shown in Figure 2).

Looking back to Figure 1, we can see that the first two configurations satisfying Turchin's relation are $a(a(x_1{+}1, m(x_1, x_1{+}1)), s(x_1))$ and $a(a(x_1, m(x_1, x_1{+}1)), s(x_1)){+}1$ (whose call-stacks in the "word form" look as aa). They are generalized by the most-specific generalization to $a(a(z, m(x_1, x_1 + 1)), s(x_1))$, which is much more specific than $a(u, z)$. We can notice that these two configurations

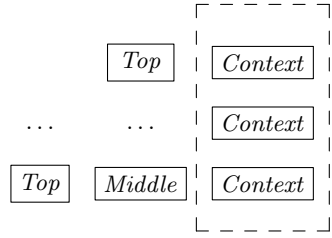

Fig. 2. Turchin's relation for call-stack configurations

do not satisfy the homeomorphic embedding relation, thus, when the composition of the relations is used as a whistle, another generalization is done. The first terms satisfying the both relations are $a(a(x_1, m(x_1, x_1 + 1)) + 1, s(x_1))$ and $a(a(x_2, m(x_2 + 1, x_2 + 1 + 1)) + 1, s(x_2 + 1)) + 1$. They are generalized to $a(a(u, m(w, w + 1)) + 1, s(w))$, which is also a good generalization with the substitutions containing no function calls. Further generalizations, which we do not discuss there, also allows a supercompiler to do driving on the whole term synchronously.

The examples above show that the composition of the homeomorphic embedding and Turchin's relation may be a better whistle than the homeomorphic embedding alone. On the one hand, they consider different properties of the configurations, which allows a supercompiler to build more specific generalizations. On the other hand, they have much in common, hence it is not likely that their composition will generate much longer bad sequences than the homeomorphic embedding alone. And even if they can sometimes generate very long bad sequences, one can have a desire that these situations will not appear too often.

3 Multi-layer Prefix Grammars

Based on the observations given in Section 2, we use the following assumptions to construct the grammar models for programs based on the call-by-name semantics.

1. A configuration can be considered as a tree of calls, and the active call stack — as a path in the tree. We use a set of labels $\mathbf{S}$ with partial order $\lhd$ for denoting the positions of the function calls in the tree.
2. Every call in the stack is modelled by a pair $<$ NAME, LABEL $>$, where LABEL $\in \mathbf{S}$.
3. Every configuration is represented as a word $\Gamma\$\Delta$ consisting of the two parts separated by the symbol \$. The structure of the active stack is placed in Γ and is linearly ordered w.r.t. labels, the function calls in the passive part of the configuration are placed in Δ.

Let Υ be a finite alphabet. Let $\mathbf{S}$ be *a label set* and $\lhd$ be a strict (non-reflexive) partial order relation over $\mathbf{S}$. We denote the 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 $\mathbf{S}' \subseteq \mathbf{S}$ and a label $t \in \mathbf{S} \setminus \mathbf{S}'$, we call t *a fresh label* w.r.t. $\mathbf{S}'$ if $\mathbf{S}'$ contains neither descendants nor ancestors of label t[4].

Henceforth, the set of finite sequences of pairs $\{\langle a, s_i\rangle | a \in \Upsilon \ \& \ s_i \in \mathbf{S}\}^*$ is denoted by $\mathsf{LW}(\Upsilon, \mathbf{S})$. Elements of $\mathsf{LW}(\Upsilon, \mathbf{S})$ are called *layered words*, and are denoted by Greek capitals Γ, Δ, Φ, Ψ, Ξ, Θ. 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 $a_1 \dots a_n$.

If Φ is a layered word, $|\Phi|$ stands for the number of the pairs in Φ 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 $\langle a, s_0\rangle$).

Expression $\Phi\langle s_0\rangle$ denotes the maximal subsequence of Φ containing only the pairs labelled with s_0. The set of all labels in Φ is denoted by $\mathbf{S}_\Phi$.

Given a label s_i and natural numbers K_1 and K_2, we define a set of *layer functions* w.r.t. label s_i, $\mathfrak{F}^{s_i}_{K_1,K_2} : \mathsf{LW}(\Upsilon, \mathbf{S}) \to \mathsf{LW}(\Upsilon, \mathbf{S})$, as a minimal set of functions containing all compositions of K_1 elementary functions, which are:

1. Append $\text{App}^{s_j}[\Psi]$ (where $s_j \in \mathbf{S}$, $\Psi \in \Upsilon^*$): given a layered word Φ, the word $\text{App}^{s_j}[\Psi](\Phi)$ is the word $\Phi\Psi\langle s_j\rangle$ such that s_j is a child of s_i w.r.t. $\mathbf{S}_\Phi \cup \{s_j\}$, s_j is fresh w.r.t. $\mathbf{S}_\Phi \setminus \{s_i\}$, and $|\Psi| \leq K_2$.
   For example, if $\text{App}^{s_1}[g] \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_1$, then
   $$\text{App}^{s_1}[g](\langle f, s_0\rangle\langle g, s_1\rangle) = \langle f, s_0\rangle\langle g, s_1\rangle\langle g, s_1\rangle$$
2. Insert $\text{Ins}^{s_j}[\Psi\langle s_k\rangle]$(where $s_j, s_k \in \mathbf{S}$, $\Psi \in \Upsilon^*$): given Φ with a non-empty $\Phi\langle s_j\rangle$, where s_j is a child of s_i w.r.t. $\mathbf{S}_\Phi$, $\text{Ins}^{s_j}[\Psi\langle s_k\rangle](\Phi)$ is the word $\Phi\Psi\langle s_k\rangle$ 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 fresh 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 $\text{Ins}^{s_1}[gf\langle s_2\rangle] \in \mathfrak{F}^{s_0}_{1,1}$[5] and $s_0 \lhd s_1$, then
   $$\text{Ins}^{s_1}[\langle gf, s_2\rangle](\langle f, s_0\rangle\langle g, s_1\rangle) = \langle f, s_0\rangle\langle g, s_1\rangle\langle gf, s_2\rangle$$
   The insert operation differs from the append operation only by introduction of an unused child label s_k, which marks the newly appended word Ψ.
3. Deleting Del^{s_j} (where $s_j \in \mathbf{S}$): given Φ with a non-empty $\Phi\langle s_j\rangle$, $s_j = \text{child}(s_i)$ w.r.t. $\mathbf{S}_\Phi$, Del^{s_j} erases $\Phi\langle s_j\rangle$ from Φ together with all $\Phi\langle t\rangle$ for which $s_j \lhd t$.
   For example, if $\text{Del}^{s_{01}} \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_{01}$, s_{02} is incomparable with s_{01}, then
   $$\text{Del}^{s_{01}}(\langle d, s_{01}\rangle\langle d, s_{02}\rangle) = \langle d, s_{02}\rangle$$
4. Copying Copy^{s_j} (where $s_j \in \mathbf{S}$): given Φ with a non-empty $\Phi\langle s_j\rangle$, $s_j = \text{child}(s_i)$ w.r.t. $\mathbf{S}_\Phi$, Copy^{s_j} appends $\Phi\langle s_k\rangle$ to Φ, where s_k is a child of s_i w.r.t.

[4] 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 $\text{child}(s_0)[\mathbf{S}']$.

[5] This condition implies that $s_2 \lhd s_1$ and $s_0 \lhd s_2$.

$\mathbf{S}_\Phi \cup \{s_j\}$, s_j is fresh w.r.t. $\mathbf{S}_\Phi \setminus \{s_i\}$, and then it appends all subsequences $\Phi\langle s_l\rangle$ labelled by the children of s_j and labels them by fresh children of s_l and so on until all the sequences $\Phi\langle t\rangle$, where $s_j \triangleleft t$, are copied exactly once. For example, if $\mathrm{Copy}^{s_{01}} \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \triangleleft s_{01}$, then

$$\mathrm{Copy}^{s_{01}}(\langle d, s_{01}\rangle) = \langle b, s_0\rangle\langle d, s_{01}\rangle\langle d, s_{02}\rangle,$$

where s_{02} is incomparable with s_{01}.

Definition 2 *Let us consider a tuple* $\mathbf{G} = \langle \Upsilon, \mathbf{S}, \mathbf{R}, \mathfrak{F}^v_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *where* Γ_0 *and* Δ_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* $j > i$ *then* $s_j \triangleleft s_i$ *or* $s_j = s_i$*,* \$ *is a special symbol,* $\$ \notin \Upsilon$*, and* $\mathfrak{F}^v_{K_1,K_2}$ *is a finite set of layer function forms where* v *runs over the label set* $\mathbf{S}$*. For every* $\mathbf{G}$*-word* $\Gamma\$\Delta$*, where* Γ *and* Δ *are words in* $\mathsf{LW}(\Upsilon, \mathbf{S})$*, we call* Γ the visible layer*, and we call* Δ the invisible layer *of* $\Gamma\$\Delta$*.*

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

- *Simple rule:*
$$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Phi\Theta\$F^{s_i}(\Psi),$$
where all the letters of Φ *are labelled either by* s_i *or by fresh descendants of* s_i*,* $F^{s_i} \in \mathfrak{F}^{s_i}$*.*
- *Pop rule: for* $\Psi\langle s_j\rangle$ *— the maximal subsequence of* Ψ *marked by some* $s_j = child(s_i) \in \mathbf{S}$*,*
$$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Psi\langle s_j\rangle\Phi\Theta\$F^{s_i}(\Psi),$$
where all the letters of Φ *are labelled either by* s_i *or by fresh descendants of* s_i*,* $F^{s_i} \in \mathfrak{F}^{s_i}$*. In a pop rule, 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. A sequence of* $\mathbf{G}$*-words starting at* $\Gamma_0\$\Delta_0$ *that are transformed by the rules from* $\mathbf{R}$ *is called a trace of* $\mathbf{G}$*.*

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

Words on the traces generated by the alphabetic multi-layer grammars are models of the call-stack configurations that appear on the path of the process tree during the call-by-name computations. Some pointers to a way for constructing these models explicitly are given in the next section.

4 Modelling Call Stack Behaviour by Multi-layer Grammars

We borrow the notions of f-function and g-function from [11] and use them in the following sense. An f-function is a function whose definition consists of a one rule with the trivial patterns (e. g., if h_1 is defined as $h_1(x_1, x_2) = x_2 + h_2(x_1 + 1)$

then h_1 is an f-function). A g-function is a function with non-trivial patterns in the definition (e. g., $h_2(x+1) = h_2(x) + h_2(x)$ is a definition of the g-function).

In order to get a grammar from a program, we treat every configuration generated by the unfolding as a tree, whose nodes are named by function or constructor names and leaves contain no function calls. First, we mark every function name in the tree by a superscript depending on the state of the function call. If the call is ready to be unfolded without unfolding of other calls, the function name is marked as "ready" (by + in the superscript). Otherwise, the function name is marked as "unready". Hence, the call names of all f-functions are always marked as "ready", while the call names of g-functions are marked as "ready" if their pattern can be matched without evaluating another call[6].

Then we delete all the nodes containing static data[7]. The remaining nodes are given the layer labels. If some node T is a descendant of a node W, the label of T is greater than the label of W. Otherwise the labels are incomparable.

Finally, we find all the nodes containing the unready call names with a single child. The child of such a node is given the label of the node. And then, all the nodes with the same labels are merged: data from the ancestor nodes are placed in the merged node after the data from their descendants.

The resulting tree is a tree form of the corresponding layered word.

Example 3 *Given the term $a(m(x_1+1, x_1+1), s(x_1))$, we transform it to a layered word. All the steps of the transformation are given in Figure 3.*

First, we mark the calls as "ready"(with + in the superscript) and "unready"(with − in the superscript), and delete the nodes with the static data. The only function call in the configuration which is ready to be unfolded without unfolding is the outermost call of m. The call $s(x_1)$ requires unfolding (which generates the narrowing on x_1), but does not require unfolding of other calls, so it is also marked as ready. The remaining call a is marked as unready.

Then we assign the layer labels in the resulting tree of the marked call names. The tree below shows that $s_0 \lhd s_{01}$, $s_0 \lhd s_{02}$.

Finally, the nodes containing the names of the calls in the active stack are extracted. These names together with the node labels take a place in the visible part of the layered word; data from the remaining node in the tree take a place in the invisible part.

---

[6] There we always can determine all the calls that are ready to be unfolded due to simplicity of the patterns. In languages with complex pattern matching (e. g., Refal [15]), that can be done only if one knows the strategy of the pattern matching applied in the interpreter.

[7] In some cases, this action can transform the tree into a forest. For example, that can happen if the configuration is $cons(h_1(x), cons(h_2(x), Nil))$. 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 \$.

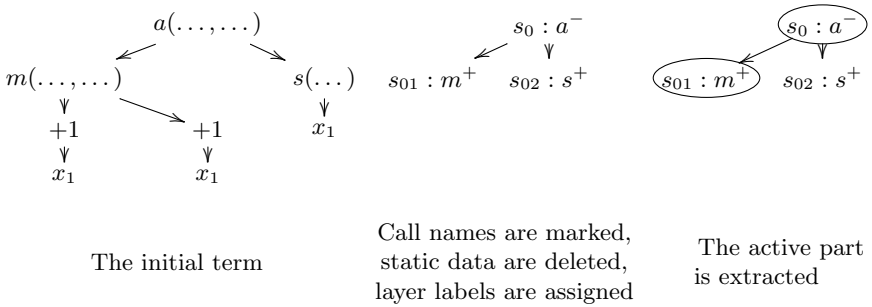

Fig. 3. Steps transforming $a(m(x_1 + 1, x_1 + 1), s(x_1))$ from a tree to layered word $\langle m, s_{01}\rangle\langle a, s_0\rangle\$\langle s, s_{02}\rangle$

5 Turchin's Relation and Multi-Layer Grammars

Definition 3 *Let* $\mathbf{G}$ *be a multi-layer prefix grammar with the set of rules that rewrite* N *letters in the visible layer. Given a trace* $\{\Gamma_k\$\Delta_k\}$ *and its segment* $[i, j]$*, suffix* Θ *of* Φ_i *is called* a permanently stable *suffix w.r.t. the segment* $[i, j]$ *if all the words* $\Gamma_k\$\Delta_k$*,* $i \le k < j$*, are of the form* $\Phi_k\Theta\$\Delta_k$ *where* Φ_k *is a layered word with the length not less than* N*, and* Γ_j *is of the form* $\Phi_j\Theta$*, where* Φ_j *may be* Λ [8]*. If* j *is not bounded,* Θ *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 at 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.

Definition 4 *Let* $\mathbf{G} = \langle \Upsilon, \mathbf{S}, R, \mathfrak{F}^v_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *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* $\Gamma_i = \Phi\Theta_0$*,* $\Gamma_j = \Phi'\Psi\Theta_0$*,* Φ *is equal to* Φ' *as a plain word (up to the layer labels) and the suffix* Θ_0 *is permanently stable w.r.t. segment* $[i, j]$*.*

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. Turchin's relation not longer than* $B_K(M, N)$*, where* K *is the maximal rewrite depth of* $\mathbf{G}$*,* M *is the number of rules in the set of rewriting rules of* $\mathbf{G}$ *and* N *is the total length of the initial word in the grammar.*

[8] In the case of alphabetic prefix grammars, when $N = 1$, the first condition implies the second.

Example 4 *A program that generates traces which are Ackermanian bad sequences w.r.t. Turchin's relation 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).*

$$
\begin{aligned}
&A(< x_1 + 1, x_2 >) = a(A(< x_1, x_2 >));\\
&A(< 0, x_2 >) =< x_2 + 1, 0 >;\\
&a(< x_1 + 1, x_2 >) = x_1;\\
&B(< x_1 + 1, x_2 >) = c(c(< x_1 + 1, x_2 >));\\
&b(< x_1 + 1, x_2 >) = x_2;\\
&c(< x_1 + 1, x_2 >) =< B(b(< x_1, x_2 >)) + 1, c(c(< x_1, x_2 >)) >;\\
&c(< 0, x_2 >) =< B(b(< 1, 0 >)) + 1, c(c(< 0, 0 >)) >;
\end{aligned}
$$

The program never stops and its call-stack configurations form bad sequences of the exponential tower length (in N*) with respect to Turchin's relation (that is, of the length* $O(2^{\left\}{2^{\cdots^2}}\right\}N})$*). However, with respect to the homeomorphic embedding relation over the entire terms, a computation of this program for every* $N > 0$ *is terminated on the* $5 + N$*-th step.*

6 Conclusion

Turchin's 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 although such sequences appear in real program runs almost never, the computational power of Turchin's relation shows that the relation can be used to solve some complex problems. In fact, the Ackermanian upper bound on the bad sequence length is rather a "good" property indicating that the relation is non-trivial, than a "bad" one indicating that the whistle using the relation will be blown too late. Example 4 shows a program that generates very long bad sequences. But the program contains an implicit definition of the Ackermanian function. From the practical point of view, such programs are very rare. If a program does not generate such complex structures, its call-stack configuration structures considered by Turchin's relation can be modelled by a grammar belonging to a smaller class than the whole class of the multi-layer prefix grammars. E.g., for the call-by-value semantics, this class coincides with the regular grammars [8]. Primitively recursive functions also cannot generate too complex stack structures: they are even incapable to compute such fast-growing functions as the Ackermanian function. Moreover, the homeomorphic embedding can produce even longer bad sequences than Turchin's relation [13, 17]. Thus, the property being described in this paper is not a thing Turchin's relation must be blamed of.

References

1. Albert, E., Gallagher, J., Gomes-Zamalla, M., Puebla, G.: Type-based homeomorphic embedding for online termination. Journal of Information Processing Letters 109(15), 879–886 (2009)
2. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems, pp. 15–24. IEEE Computer Society Press (2012)
3. Klyuchnikov, I.: Inferring and proving properties of functional programs by means of supercompilation. Ph. D. Thesis (in Russian) (2010)
4. Kruskal, J.: Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. Transactions of the American Mathematical Society 95, 210–225
5. Leuschel, M.: Homeomorphic Embedding for Online Termination of Symbolic Methods, Lecture Notes in Computer Science, vol. 2566, pp. 379–403. IEEE Computer Society Press (2002)
6. Nash-Williams, C.S.J.A.: On well-quasi-ordering infinite trees. Proceedings of Cambridge Philosophical Society 61, 697–720 (1965)
7. Nemytykh, A.P.: The Supercompiler Scp4: General Structure. URSS, Moscow (2007), (In Russian)
8. Nepeivoda, A.: Turchin's relation and subsequence relation in loop approximation. In: PSI 2014. Ershov Informatics Conference. Poster Session. vol. 23, pp. 30–42. EPiC Series, EasyChair (2014)
9. Nepeivoda, A.: Turchin's relation for call-by-name computations: a formal approach. (to appear) (2016)
10. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Proceedings of ILPS'95, the International Logic Programming Symposium. pp. 465–479. MIT Press (1995)
11. Sørensen, M.: Turchin's supercompiler revisited. Ms.Thesis (1994)
12. Sørensen, M., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming 6, 811–838 (1996)
13. Touzet, H.: A characterisation of multiply recursive functions with higman's lemma. Information and Computation 178, 534–544 (2002)
14. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (1986)
15. Turchin, V.F.: Refal-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, Massachusetts (1989), electronic version:`http://www.botik.ru/pub/local/scp/refal5/`
16. Turchin, V.: The algorithm of generalization in the supercompiler. Partial Evaluation and Mixed Computation pp. 341–353 (1988)
17. Weiermann, A.: Phase transition thresholds for some friedman-style independence results. Mathematical Logic Quarterly 53, 4–18 (2007)