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

Antonina N. Nepeivoda[1]

[1]Program Systems Institute
Russian Academy of Sciences

Fifth International Valentin Turchin Workshop on Metacomputation
Pereslavl–Zalessky, 2016

“Similarity” conditions for termination

N. Dershowitz on term-rewriting systems (approx. 1982):

wqos (and the homeomorphic embedding $\trianglelefteq$) for ensuring termination.

Wqos are considered as termination criteria for term rewriting systems:

- work with a uniform alphabet $\Rightarrow$ do not distinguish between constructor and function names.
- are required to work with every rewriting order $\Rightarrow$ are not specified to any computation order;

Seminal Work of V. Turchin

(published in 1988):

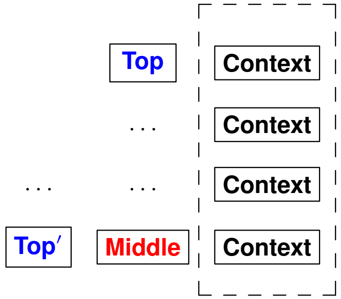
The first algorithm of generalization strategy and whistle combined in single relation $\preceq$.

The relation specialized for comparing call-stack structures:

- considers only function calls' names;
- takes the semantics into account.

Relation for Stacks

A call stack ⇒ a linear structure ⇒ can be considered as a word.



Features:

- Considers not only configuration data but also the history of the changes in the data (as in e.g. [Gallagher et al, 1996]).
- Helps to generalize w.r.t. the computation order.

Example

Computation Path	Stack Configuration		
(0) `f(x+1+1)`			$f^0(arg_1)$
(1) `f(g(x+1+1))+1`		$arg_1 = g^1(arg_2)$;	$f^1(arg_1)$
(2) `f(h(x+1))+1`		$arg_1 = h^2(arg_2)$;	$f^1(arg_1)$
(3) `f(g(x)+1)+1`			$f^1(arg_1)$
(4) `f(g(g(x)+1))+1+1`		$arg_1 = g^4(arg_2)$;	$f^4(arg_1)$
(5) `f(h(g(x)))+1+1`	$arg_2 = g^5(arg_3)$;	$arg_1 = h^5(arg_2)$;	$f^4(arg_1)$

The pairs of the similarly colored configurations are in Turchin's relation.

(1) and (5) are not in Turchin's relation (the time index of the call `f` is changed).

$f(x)$ **computes** $[\log_2(x)] + 1$**:**

```
f(0)=0;
f(x+1)=f(g(x+1))+1;

g(0)=0;
g(x+1)=h(x);

h(0)=0;
h(x+1)=g(x)+1;
```

How it works

$f(x)$ **computes** $[\log_2(x)] + 1$**:**

Initial Program

```
f(0)=0;
f(x+1)=f(g(x+1))+1;

g(0)=0;
g(x+1)=h(x);

h(0)=0;
h(x+1)=g(x)+1;
```

Residual Program

```
f(0)=0;
f(1)=1;
f(2)=1;
f(x+1+1+1)=f(g(x)+1);

g(0)=0;
g(1)=0;
g(x+1+1)=g(x)+1;
```

The residual program above is a result of supercompilation using the composition of $\trianglelefteq$ and $\preceq$. Every relation used alone gives a less efficient result.

How it works-2

$s(x)$ computes $\sum_{k=1}^{x} k^2$:

Initial Program

```
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));
```

Residual Program

```
s(x)=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 residual program above is a result of supercompilation using the composition of $\trianglelefteq$ and $\preceq$. Again the both of them, used alone, are less efficient.

Why does semantics matter?

[L. Puel, 1985] The problem with the homeomorphic embedding: definitions $LHS = RHS$ where $LHS \trianglelefteq RHS$.

- May be a rule that always leads to an infinite loop, e.g.:

```
f(x+1)=f(g(x)+1);
```

- May terminate, e.g.:

```
f(x+1)=f(g(x+1));
g(x+1)=0;
```

- May result in an infinite loop or terminate depending on the semantics, e.g.:

```
f(x+1)=h(f(g(x)+1)+1);
h(x+1)=0;
```

No relations given in [Leuschel, 2002] or [Mogensen, 2013] can distinguish between these cases.

How does Turchin’s relation work?

Advantages:

- Depends on the computation order; thus, takes the semantics into account (STACK).
- Depends on the global properties of the process tree, not only properties of the term considered (HISTORY).

Problems:

- STACK — badly formalized behaviour.
- HISTORY — space-consuming.

Call-by-Value Semantics

In call-by-value semantics, every call stack configuration can be considered as a linear structure with a finite prefix rewriting.

Program	Prefix Rewriting Rule
`f(0)=0;`	$f \to \Lambda$
`f(x+1)=f(g(x+1))+1;`	$f \to gf$
`g(0)=0;`	$g \to \Lambda$
`g(x+1)=h(x);`	$g \to h$
`h(0)=0;`	$h \to \Lambda$
`h(x+1)=g(x)+1;`	$h \to g$

Call stack transformations can be described by prefix grammars (equivalent to finite automata). Turchin's theorem is easily proved [AN,2014].

Call-by-Name Semantics

In call-by-name semantics, every call stack configuration is linear. But its transformations depend on the passive part of the configuration.

Computation Path	Stack Configuration
`f(g(g(x))+1+1)`	$f(arg_1)$
`f(g(g(g(x))+1+1))+1`	$arg_1 = g(arg_2)$; $f(arg_1)$
`f(h(g(g(x))+1))+1`	$arg_1 = h(arg_2)$; $f(arg_1)$
`f(g(g(g(x)))+1)+1`	$f(arg_1)$
`f(g(g(g(g(x))+1))+1+1`	$arg_1 = g(arg_2)$; $f(arg_1)$
`f(h(g(g(g(x))))))+1+1`	$arg_4 = g(arg_5)$; $arg_3 = g(arg_4)$; $arg_2 = g(arg_3)$; $arg_1 = h(arg_2)$; $f(arg_1)$

The red part of the last call stack is popped from the passive configuration. Observing only the active stack, we cannot predict how it is changed by an execution of the rule `g(x+1) = h(x);`

Structure of the Passive Part of Configuration

Given the configuration `b(d(x)+1,b(d(x),d(y))+1)`, we apply `b(x+1,y+1) = b(d(b(x,y+1)),y)` to it.

The initial configuration has the following structure

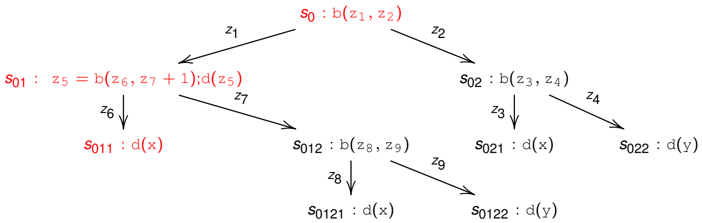
$$\begin{tikzcd}
 & s_0 : \mathtt{b}(\mathtt{z}_1 + \mathtt{1}, \mathtt{z}_2 + \mathtt{1}) \arrow[dl, "z_1"'] \arrow[dr, "z_2"] & \\
s_{01} : \mathtt{d}(\mathtt{x}) & & s_{02} : \mathtt{b}(\mathtt{z}_3, \mathtt{z}_4) \arrow[dl, "z_3"'] \arrow[d, "z_4"] \\
 & s_{021} : \mathtt{d}(\mathtt{x}) & s_{022} : \mathtt{d}(\mathtt{y})
\end{tikzcd}$$

The colored call is placed to the current call stack.

Structure of the Passive Part of Configuration

Given the configuration `b(d(x)+1,b(d(x),d(y))+1)`, we apply `b(x+1,y+1) = b(d(b(x,y+1)),y)` to it.

After the application, the function call tree becomes as follows



$$\begin{tikzcd}
 & s_0 : \mathrm{b}(z_1, z_2) \arrow[dl, "z_1"'] \arrow[drr, "z_2"] & & & \\
s_{01} : z_5 = \mathrm{b}(z_6, z_7 + 1);\mathrm{d}(z_5) \arrow[d, "z_6"'] \arrow[dr, "z_7"] & & & s_{02} : \mathrm{b}(z_3, z_4) \arrow[d, "z_3"'] \arrow[dr, "z_4"] & \\
s_{011} : \mathrm{d}(\mathrm{x}) & s_{012} : \mathrm{b}(z_8, z_9) \arrow[d, "z_8"'] \arrow[dr, "z_9"] & & s_{021} : \mathrm{d}(\mathrm{x}) & s_{022} : \mathrm{d}(\mathrm{y}) \\
 & s_{0121} : \mathrm{d}(\mathrm{x}) & s_{0122} : \mathrm{d}(\mathrm{y}) & &
\end{tikzcd}$$

The colored calls are placed to the current call stack.

A Problem of Formalization

The main interest is not *what data* can be computed by a given program but *what stack configurations* can appear along a computation path of the given program.

Turchin's relation considers every stack as a word and ignores static data.

The Problem

Given a program, what class of grammars can generate words that correspond to call stack configurations generated by the program?

Constructing a Model of the Call Stack Configuration

- The function call configuration forms a tree of calls, and the active call stack is a path in the tree of calls.
- Every call in the stack is modelled by the pair `<NAME, LABEL>`. The set of all labels **S** has partial order $\triangleleft$. The set of all names is Υ.
- Every configuration is represented as a layered word $\Gamma\$\Delta$. The structure of the active stack Γ is linearly ordered w.r.t. labels, and the invisible part Δ contains data about the passive part of the tree of the function calls.

Definition

For every layered word $\Gamma\$\Delta$, where Γ and Δ are words over $\Upsilon \times \mathbf{S}$, we call Γ *the visible layer*, and we call Δ *the invisible layer* of $\Gamma\$\Delta$.

Constructing a Model of the Call Stack Configuration

Example

Given configuration `f(g(h(g(g(x)))+1))+1`, the layered word modelling its call stack can be

$$\langle g, s_1\rangle\langle f, s_0\rangle\$\langle ggh, s_2\rangle$$

$s_0 \lhd s_1 \lhd s_2$.

The three calls in the passive part are given the same label s_2, because they must be popped from the passive part only together.

Constructing a Model of the Call Stack Configuration

(Visible) transformations of the active part of the call stack are:

- Erasure of the call on the stack top (when the call is computed and is deleted from the stack).
- Pushing a bounded number of calls to the stack (as in the call-by-value semantics).
- Popping a new stack top from the passive part of the configuration.

These transformations change the visible part Γ of the layered word $\Gamma\$\Delta$. The invisible part Δ is also changed by combinations of some basic operators on the invisible layer.

Basic Layer Operator: Append

Given a fixed label s_i and its child s_j,

$$\mathrm{App}^{s_j}[\Psi](\Phi) = \Phi\langle \Psi, s_j \rangle$$

On tree representations, the appending operator appends some new letters to an existing child of s_i.

$$\begin{tikzcd}
& \dots \arrow[d] & & & & \dots \arrow[d] & \\
& s_i : \Theta_i \arrow[dl] \arrow[dr] & & \Longrightarrow & & s_i : \Theta_i \arrow[dl] \arrow[dr] & \\
s_l : \Theta_l & & s_j : \Theta_j & & s_l : \Theta_l & & s_j : \Theta_j \Psi
\end{tikzcd}$$

Basic Layer Operator: Append

Program Data

```
f(h(g(x)+1))
     ↓
f(g(g(x))+1)
```

Layered Word

$$\langle h, s_0\rangle\langle f, s_0\rangle\$\langle g, s_1\rangle$$

$$\downarrow$$

$$\langle f, s_0\rangle\$\langle g, s_1\rangle\langle g, s_1\rangle = \langle f, s_0\rangle\$\,\mathrm{App}^{s_1}[g](\langle g, s_1\rangle)$$

Program

```
f(0)=0;
f(x+1)=
  f(g(x+1))+1;

g(0)=0;
g(x+1)=h(x);

h(0)=0;
h(x+1)=g(x)+1;
```

Basic Layer Operator: Insert

Given a fixed label s_i and its child s_j,

$\mathrm{Ins}^{s_j}[\Psi\langle s_k\rangle](\Phi) = \Phi\langle\Psi, s_k\rangle$, where s_k is a new label that is a child of s_i and the parent of s_j.

Differs from App^{s_j} only by introduction of an unused child label s_k, which marks Ψ.

On tree representations, the insert operator inserts a new node between the nodes labelled by s_i and s_j.

$$\begin{tikzcd}
& \dots \arrow[d] & & & & \dots \arrow[d] & \\
& s_i : \Theta_i \arrow[dl] \arrow[dr] & & \Longrightarrow & & s_i : \Theta_i \arrow[dl] \arrow[dr] & \\
s_l : \Theta_l & & s_j : \Theta_j & & s_l : \Theta_l & & \color{red}{s_k : \Psi} \arrow[d] \\
& & & & & & s_j : \Theta_j
\end{tikzcd}$$

Basic Layer Operator: Insert

Program Data

```
f(f(g(x)+1))
↓
f(f(g(g(x)+1))+1)
```

Layered Word

$$\langle f, s_0\rangle\langle f, s_0\rangle\$\langle g, s_1\rangle$$

$$\downarrow$$

$$\langle f, s_0\rangle\$\langle g, s_1\rangle\langle gf, s_2\rangle = \langle f, s_0\rangle\$\,\mathrm{Ins}^{s_1}[gf\langle s_2\rangle](\langle g, s_1\rangle)$$

Program

```
f(0)=0;
f(x+1)=
 f(g(x+1))+1;


g(0)=0;
g(x+1)=h(x);

h(0)=0;
h(x+1)=g(x)+1;
```

Basic Layer Operator: Delete

Given a fixed label s_i and its child s_j,

$$\mathrm{Del}^{s_j}(\Phi) = \Phi', \text{ where } \Phi' \text{ is the subsequence of } \Phi \text{ not containing letters labelled by } s_j = \mathrm{child}(s_i) \text{ or by descendants of } s_j.$$

On tree representations, Del^{s_j} deletes the subtree whose uppermost node is labelled by s_j.

$$\begin{tikzcd}
& \cdots \arrow[d] & & & & \cdots \arrow[d] \\
& s_i : \Theta_i \arrow[dl] \arrow[dr] & & \Longrightarrow & & s_i : \Theta_i \arrow[d] \\
s_l : \Theta_l & & {\color{red} s_j : \Theta_j} \arrow[d] & & & s_l : \Theta_l \\
& & {\color{red} s_{j+1} : \Theta_{j+1}} & & &
\end{tikzcd}$$

Basic Layer Operator: Delete

Program Data

```
b(b(0, d(x)+1),d(x))
          ↓
     b(1,d(x))
```

Layered Word

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

Program

```
b(0,y)=1;
b(x,0)=x;
b(x+1,y+1)=
 b(d(b(x,y+1)),
   y);

d(0)=0;
d(x+1)=
 d(x)+1+1;
```

Basic Layer Operator: Copy

Given a fixed label s_i and its child s_j,

$\mathrm{Copy}^{s_j}(\Phi) = \Phi\Phi'$, where Φ' repeats the subsequence of Φ labelled by s_j and its descendants, with the fresh labels.

On tree representations, Copy^{s_j} makes a copy of the subtree whose uppermost node is labelled by s_j.

$$\begin{tikzcd}
 & \dots \arrow[d] & & & & \dots \arrow[d] & \\
 & s_i : \Theta_i \arrow[dl] \arrow[dr] & & \Longrightarrow & & s_i : \Theta_i \arrow[dl] \arrow[d] \arrow[dr] & \\
s_l : \Theta_l & & s_j : \Theta_j \arrow[d] & & s_l : \Theta_l & s_j : \Theta_j \arrow[d] & {\color{red} s_k : \Theta_j} \arrow[d] \\
 & & s_{j+1} : \Theta_{j+1} & & & s_{j+1} : \Theta_{j+1} & {\color{red} s_{k+1} : \Theta_{j+1}}
\end{tikzcd}$$

Basic Layer Operator: Copy

Program Data

```
b(x+1,d(y)+1)
      ↓
b(d(b(x,d(y)+1)),d(y))
```

Layered Word

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

Program

```
b(0,y)=1;
b(x,0)=x;
b(x+1,y+1)=
 b(d(b(x,y+1)),
   y);

d(0)=0;
d(x+1)=
 d(x)+1+1;
```

Layer Functions: Summary

- are allowed to modify only *the children* of the node s_i.
  The following is forbidden!

$$\begin{tikzcd}
 & s_i : \Theta_i \arrow[dl] \arrow[dr] & & & & s_i : \Theta_i \arrow[dl] \arrow[dr] & \\
s_l : \Theta_l & & \cdots \arrow[d] & \Longrightarrow & s_l : \Theta_l & & \cdots \arrow[d] \\
 & & s_{j+k} : \Theta_{j+k} & & & & s_{j+k} : \Theta_{j+k}\Psi
\end{tikzcd}$$

- are allowed to increase a longest word fragment starting from s_i *at most by a constant*.
  The following is also forbidden!

$$\begin{tikzcd}
 & s_i : \Theta_i \arrow[dl] \arrow[dr] & & & s_i : \Theta_i \arrow[d] \\
s_l : \Theta_l & & s_j : \Theta_j & \Longrightarrow & s_j : \Theta_j \arrow[d] \\
 & & & & s_l : \Theta_l
\end{tikzcd}$$

Formal Definition of Multi-Layer Prefix Grammars

A multi-layer prefix grammar — a tuple $\mathbf{G} = \langle \Upsilon, \mathbf{S}, \mathbf{R}, \mathfrak{F}^x, \Gamma_0\$\Delta_0 \rangle$, where:

- Υ is an alphabet of names, **S** is a set of labels;
- $\Gamma_0\$\Delta_0$ is the initial word, Γ_0 is linearly ordered w.r.t. labels;
- $\mathfrak{F}^x$ is a finite set of basic-layer-operator compositions (x runs over **S**);
- **R** is a finite set of rewriting rules, which are either:
  - Simple rules:
    $$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Phi\Theta\$F^{s_i}(\Psi),$$
    $|\Xi| \leq L$, $|\Phi| \leq K$, all the letters of Φ have label s_i, $F^{s_i} \in \mathfrak{F}^{s_i}$.
  - Pop rules: for Ψ' — a maximal subsequence of Ψ marked by some $s_j = \mathrm{child}(s_i) \in \mathbf{S}$,
    $$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Psi'\Phi\Theta\$F^{s_i}(\Psi),$$
    $|\Xi| \leq L$, $|\Phi| \leq K$, all the letters of Φ have label s_i, $F^{s_i} \in \mathfrak{F}^{s_i}$.

Alphabetic multi-layer prefix grammars: $\Xi = \Lambda$ ($L = 0$).

Turchin's Relation for Multi-Layer Prefix Grammars

Definition

Given an alphabetic multi-layer grammar **G**, *a common context* for the two words $\Gamma_1\$\Delta_1$ and $\Gamma_2\$\Delta_2$ in a trace generated by **G** is a maximal common suffix Θ of Γ_1 and Γ_2 such that $\Theta[1]$ was preceded at least by a one letter on the whole trace segment starting with $\Gamma_1\$\Delta_1$ and ending by $\Gamma_2\$\Delta_2$.

This means that the common context was not changed!

Turchin's Relation for Multi-Layer Prefix Grammars

Definition

Let **G** be a multi-layer prefix grammar. Given two layered words $\Xi_i = \Gamma_i\$\Delta_i$, $\Xi_j = \Gamma_j\$\Delta_j$ in a trace generated by **G**, we say that the words form *a Turchin pair* (thus, $\Xi_i \preceq \Xi_j$) 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 the common context of Ξ_i and Ξ_j.

Theorem (Strong Turchin's Theorem)
Every infinite trace generated by a multi-layer prefix grammar **G** contains an infinite subsequence which is linearly ordered w.r.t. Turchin's relation.

Hence, every computation path modelled as such a trace contains an infinite chain of call stack configurations w.r.t. $\preceq$. That allows us to combine $\preceq$ with usually used wqos.

Complexity of Turchin's relation

Given a multi-layer grammar **G**:

- let rules of **G** increase a word along a longest path in the corresponding labelled tree at most by K letters;
- let the number of the different prefixes involved in the right-hand sides of the rules be M;
- let the total length of the initial word be N.

We define the following Ackermann function:
$B_K(M, 0) = 1$
$B_K(0, N) = N + 1$
$B_K(M, N) = B_K(M - 1, B_K(M, N - 1) * K)$

Theorem
An alphabetic multi-layer grammar **G** can generate bad sequences w.r.t. Turchin's relation not longer than $B_K(M, N)$.

Proof Idea

Given the initial word Γ_0, build the maximal bad sequence using $M - 1$ prefixes and all the letters of Γ_0 without the last letter (segment σ_1). Store the maximal longest path Δ_{k_1} consisting of a remaining prefix to the invisible layer.

Pop this path instead of $\Gamma_0[\mathit{last}]$ and proceed constructing the bad sequence using $M - 2$ prefixes and Δ_{k_1} as the initial word.

Repeat the construction until all the prefixes are used in Δ_{k_i}.

$$\left.\begin{array}{r}\Gamma_0\$\Lambda\\ \dots\\ \Gamma_0[\mathit{last}]\$\Delta_{k_1}\end{array}\right\}\sigma_1$$

$$\left.\begin{array}{r}\Delta_{k_1}\$\Lambda\\ \dots\\ \Delta_{k_1}[\mathit{last}]\$\Delta_{k_2}\end{array}\right\}\sigma_2$$

$$\dots$$

$$\left.\begin{array}{r}\Delta_{k_n}\$\Lambda\\ \dots\\ \Lambda\$\Delta_{k_{n+1}}\end{array}\right\}\sigma_n$$

Advantage or Disadvantage?

- Turchin's relation possibly produces very long and not useful unfolding, which consumes time and can imply unreadable residual programs.
- Long bad sequences are practically rare. Usually — more useful unfolding. For primitively recursive functions — no problem at all, since they cannot express Ackermann functions.

The homeomorphic embedding can also admit bad sequences with a hyper-ackermanian length even in the case all the terms are unary! [Touzet, 2002].

Usual Example: Accurate Termination

Configuration $\mathtt{f}(\mathtt{g}(\mathtt{x}_1)+1)$ is embedded in $\mathtt{f}(\mathtt{g}(\mathtt{g}(\mathtt{x}_1)+1))$. But the call stacks of the configurations correspond to layered words $\langle f, s_0\rangle\$\langle g, s_1\rangle$ and $\langle gf, s_0\rangle\$\langle g, s_1\rangle$ and do not satisfy Turchin's relation. It is satisfied on the framed configurations.

Process Tree for `f(h(x))`

$$\begin{tikzcd}
 & \boxed{f(h(x))} \arrow[dl, "x=0"'] \arrow[dr, "x=x_1+1"] & \\
f(0) \arrow[d] & & f(g(x_1)+1) \arrow[d] \\
0 & & f(g(g(x_1)+1))+1 \arrow[d] \\
 & & f(h(g(x_1)))+1 \arrow[dll, "x_1=0"'] \arrow[d, "x_1=x_2+1"] \\
f(h(0))+1 \arrow[d] & & f(h(h(x_2)))+1 \arrow[dl, "x_2=0"'] \arrow[d, "x_2=x_3+1"] \\
f(0)+1 \arrow[d] & f(h(0))+1 \arrow[d] & \boxed{f(h(g(x_3)+1))+1} \arrow[d] \\
1 & f(0)+1 & \dots
\end{tikzcd}$$

Rare Example: Ackermanian Bad Sequence

$$\begin{array}{lcl}
\mathtt{A}(<\mathtt{x}+1,\mathtt{y}>) & = & \mathtt{a}(\mathtt{A}(<\mathtt{x},\mathtt{y}>)); \\
\mathtt{A}(<0,\mathtt{y}>) & = & <\mathtt{y}+1,0>; \\
\mathtt{a}(<\mathtt{x}+1,\mathtt{y}>) & = & \mathtt{x}; \\
\mathtt{B}(<\mathtt{x}+1,\mathtt{y}>) & = & \mathtt{c}(\mathtt{c}(<\mathtt{x}+1,\mathtt{y}>)); \\
\mathtt{b}(<\mathtt{x}+1,\mathtt{y}>) & = & \mathtt{y}; \\
\mathtt{c}(<\mathtt{x}+1,\mathtt{y}>) & = & <\mathtt{B}(\mathtt{b}(<\mathtt{x},\mathtt{y}>))+1,\mathtt{c}(\mathtt{c}(<\mathtt{x},\mathtt{y}>))>; \\
\mathtt{c}(<0,\mathtt{y}>) & = & <\mathtt{B}(\mathtt{b}(<1,0>))+1,\mathtt{c}(\mathtt{c}(<0,0>))>;
\end{array}$$

The input point is $\mathtt{A}(<\mathtt{N},\mathtt{b}(\mathtt{B}(<1,0>))>)$ (where $\mathtt{N}$ is an arbitrary fixed natural number)

- The program never stops.
- The length of the computation path until a Turchin pair on its call stack configurations appears is $O(2^{\left.2^{\cdots^{2}}\right\}N})$.
- The length of the computation path until a pair of homeomorphically embedded configurations appears is $5+N$.

Conclusion

Turchin's relation $\preceq$:

- a natural way to discover loops on computation paths with accordance to the semantics;
- a rather strong whistle that can deal with complex computations;
- works with flat structures, hence cheap as a whistle (and we even can omit time indices, it will still help a lot!).

Thus, $\preceq$ is more perspective as a whistle than e.g. the "Eulerian" whistle considering term trees as words [Mogensen, 2013] because $\preceq$ considers *natural* flat structure of the call-stack, not artificial one.

Bibliography

- J. Gallagher, L. Lavafe: *Regular approximation of computation paths in logic and functional languages*, 1996.
- M. Leuschel: *Homeomorphic Embedding for Online Termination of Symbolic Methods*, 2002.
- T. Mogensen: *A Comparison of Well-Quasi Orders on Trees*, 2013.
- A. P. Nemytykh: *The Supercompiler Scp4: General Structure*, 2007.
- A. Nepeivoda: *Turchin's Relation and Subsequence Relation in Loop Approximation*, 2014.
- L. Puel: *Using Unavoidable Set of Trees to Generalize Kruskal's Theorem*, 1985.
- H. Touzet: *A Characterisation of Multiply Recursive Functions with Higman's Lemma*, 2002.
- V. F. Turchin: *The algorithm of generalization in the supercompiler*, 1988.

Thank You