Turchin's Relation for Call-by-Name Computations
A Formal Approach

Antonina N. Nepeivoda[1]

[1]Program Systems Institute
Russian Academy of Sciences

Workshop on Verification and Program Transformation
Eindhoven, 2016

Simple Programming Language

Features:

- Data types: natural numbers and lists. Constructors: `cons(...)`, a tuple constructor `<...>`, increment (`+1`).
- No nested functions, no multiple variables in left-hand sides.
- The patterns should be matched from top to bottom.
- Semantics: call-by-name.

Example (binary logarithm+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;
```

Supercompilation: Introduction

Supercompilation is a program transformation method based on fold/unfold technique, that was proposed by V. Turchin in the 1970s.

Observes the behaviour of a functional program running on partially defined input point. Tries to fold its computation tree into a graph which produces the residual program with better properties.

Two general questions upon the supercompilation:

- When it is better to unfold the computation path, and when — to terminate unfolding?
- How it is better to fold the computation tree into a graph?

Termination of Unfolding Procedure

Definition

A relation R is *well binary* on the set of sequences **S**, if every infinite sequence $\{\Phi_n\} \in$ **S** contains Φ_i, Φ_j such that $i < j$ and $(\Phi_i, \Phi_j) \in R$.

It is sufficient for R to be well-binary for traces that can be observed in computation trees.

Examples:

- The Turchin relation for call stacks in SCP4 [A. Nemytykh, 2000–2007].
- The De-Brujin-index-based relation in HOSC [I. Klyuchnikov, 2010];

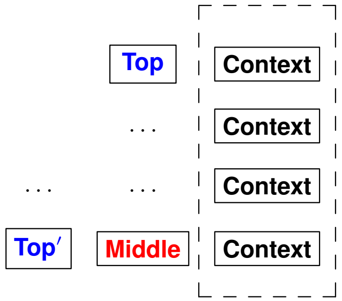
Definition

Sequence $\{\Phi_n\}$ such that $\forall i, j (i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is *a bad sequence with respect to R*.

The longer bad sequences are, the slower the program transformation tool works, and the more finite computation paths are discovered by its work.

Stack Similarity Relation as a Branch Termination Condition

A stack configuration is a linear structure. Two stack configurations on a computation path can be tested for similarity as follows.



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. global properties of the path.

Example

Computation Path:	Stack Configuration:		
(0) `f(x+1+1)`			$f^0(arg1)$
(1) `f(g(x+1+1))+1`		$arg1 = g^1(arg2);$	$f^1(arg1)$
(2) `f(h(x+1))+1`		$arg1 = h^2(arg2);$	$f^1(arg1)$
(3) `f(g(x)+1)+1`			$f^1(arg1)$
(4) `f(g(g(x)+1))+1+1`		$arg1 = g^4(arg2);$	$f^4(arg1)$
(5) `f(h(g(x)))+1+1`	$arg2 = g^5(arg3);$	$arg1 = h^5(arg2);$	$f^4(arg1)$

The upper indices represent time: all of them correspond to the number of a configuration in the path in where the function call first appeared in the stack.

The pairs of the similarly colored configurations are in the Turchin relation with the empty context. The configurations (2) and (5) are not in the Turchin relation (the time index of the call `f` is changed).

Challenges

- A formal proof of consistency of the Turchin relation as a termination criterion for call-by-name computations.
- Whether the Turchin relation can be used in composition with the homeomorphic embedding relation?
- What is the worst-case complexity of the Turchin relation?

Challenges

Theorem (Strengthened Turchin's Theorem)

Every computation path contains an infinite chain of call stack configurations w.r.t. the Turchin relation.

- For the call-by-value semantics, the formalization of the Turchin relation pointed to the way to verify cryptographical protocols by the means of the refined relation due to the strengthened Turchin's theorem.
- We prove the analogous strengthened version of Turchin's theorem for a formalization of the call-by-name semantics.

A Problem of Formalization

The main interest is not what data can be computed by a given program but what stack configurations can appear on a computation path of the given program on a parameterized input point.

The Turchin 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?

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 rewriting grammars (prefix grammars), which are equivalent to finite automata.

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)`	$f(arg1)$
`f(g(g(g(x))+1+1))+1`	$arg1 = g(arg2);\quad f(arg1)$
`f(h(g(g(x))+1))+1`	$arg1 = h(arg2);\quad f(arg1)$
`f(g(g(g(x)))+1)+1`	$f(arg1)$
`f(g(g(g(g(x))+1))+1+1`	$arg1 = g(arg2);\quad f(arg1)$
`f(h(g(g(g(x)))))+1+1`	$arg4 = g(arg5);\quad arg3 = g(arg4);$ $arg2 = g(arg3);\quad arg1 = h(arg2);\quad f(arg1)$

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 program sentence `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(x1+1,x2+1) = b(d(b(x1,x2+1)),x2)` to it.

The initial configuration has the following structure

$$\begin{tikzcd}
 & & s_0 : \mathtt{b}(\mathtt{z1}+\mathtt{1}, \mathtt{z2}+\mathtt{1})) \arrow[dll, "z1"'] \arrow[drr, "z2"] & & \\
s_{01} : \mathtt{d}(\mathtt{x}) & & & & s_{02} : \mathtt{b}(\mathtt{z3}, \mathtt{z4}) \arrow[dll, "z3"'] \arrow[d, "z4"] \\
 & & s_{021} : \mathtt{d}(\mathtt{x}) & & s_{022} : \mathtt{d}(\mathtt{y})
\end{tikzcd}$$

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(x1+1,x2+1) = b(d(b(x1,x2+1)),x2)` to it.

After the application, the function call tree becomes as follows

$$\begin{tikzcd}
 & & s_0 : \mathtt{b}(\mathtt{z1}, \mathtt{z2})) \arrow[dll, "z1"'] \arrow[drr, "z2"] & & & \\
s_{01} : \ \mathtt{z5} = \mathtt{b}(\mathtt{z6}, \mathtt{z7} + 1);\mathtt{d}(\mathtt{z5}) \arrow[d, "z6"'] \arrow[drr, "z7"] & & & & s_{02} : \mathtt{b}(\mathtt{z3}, \mathtt{z4}) \arrow[d, "z3"'] \arrow[dr, "z4"] & \\
s_{011} : \mathtt{d}(\mathtt{x}) & & s_{012} : \mathtt{b}(\mathtt{z8}, \mathtt{z9}) \arrow[d, "z8"'] \arrow[dr, "z9"] & & s_{021} : \mathtt{d}(\mathtt{x}) & s_{022} : \mathtt{d}(\mathtt{y}) \\
 & & s_{0121} : \mathtt{d}(\mathtt{x}) & s_{0122} : \mathtt{d}(\mathtt{y}) & &
\end{tikzcd}$$

The colored calls are placed to the current call stack.

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 $\lhd$. 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+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 model 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,

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

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

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

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

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,

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

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

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,

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

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

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

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,x2)=1;
b(x1,0)=x1;
b(x1+1,x2+1)=
 b(d(b(x1,x2+1)),
   x2);

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

Basic Layer Operator: Copy

Given a fixed label s_i,

$\mathrm{Copy}^{s_j}(\Phi) = \Phi\Phi'$, where Φ' repeats the subsequence of Φ labelled by $s_j = \mathrm{child}(s_i)$ 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.

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

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

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,x2)=1;
b(x1,0)=x1;
b(x1+1,x2+1)=
 b(d(b(x1,x2+1)),
   x2);

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

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 form compositions (x runs over **S**);
- **R** is a finite set of rewriting rules, which are either:
  - Simple rules:
    $$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Phi\Theta\$F^{s_i}(\Psi),$$
    where 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 $s_j = \mathrm{child}(s_i) \in \mathbf{S}$,
    $$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Psi\Phi\Theta\$F^{s_i}(\Psi),$$
    where all the letters of Φ have label s_i, $F^{s_i} \in \mathfrak{F}^{s_i}$.

Alphabetic multi-layer prefix grammars: $\Xi = \Lambda$.

Languages Generated by Multi-Layer Prefix Grammars

Multi-layer prefix grammars model call stack configurations $\Rightarrow$ only words on the visible layer are of interest.

We add an endmark symbol STOP to alphabet Υ. Let the initial word of multi-layer prefix grammar **G** contain STOP in the invisible part.

Definition

A language generated by **G** is the set of all the words $A \in \Upsilon^*$ s.t. A is a plain word corresponding to the visible layer Γ of some layered word $\Gamma\$\Delta$, s.t.:

- $\Gamma\$\Delta$ is produced by a finite trace of multi-layer prefix grammar **G**;
- Δ does not contain STOP.

Languages Generated by Multi-Layer Prefix Grammars

STOP can be considered as the name of a call which we want to trace. When STOP disappears from the invisible layer, we fix the stack configuration where it happened.

Some Facts

- Non-alphabetic multi-layer prefix grammars can generate every recursively enumerable language.
- Alphabetic multi-layer grammars can generate every context-free language and some languages that are not context-free (e. g., $\{a^{2^n} | n \in \mathbb{N}\}$).

Turchin's Theorem for Multi-Layer Prefix Grammars

Definition

A common context for the two words $\Gamma_1\$\Delta_1$ and $\Gamma_2\$\Delta_2$ in a trace 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 with $\Gamma_2\$\Delta_2$.

This means that the common context was not changed!

Turchin's Theorem 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* (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 the common context of Ξ_i and Ξ_j.

Theorem (Strengthened 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. the Turchin relation.

Hence, every computation path modelled as such a trace contains an infinite chain of call stack configurations w.r.t. the Turchin relation.

More Features of the Turchin relation

Application

- The Turchin relation can be used in composition with the homeomorphic embedding relation. The unfolding will remain finite.

Unfolding Time

For the call-by-name semantics, the maximal bad sequence length w.r.t. $\preceq$ is Ackermanian in the initial program size.

- 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.

Turchin Relation for Termination: Example

(L. Puel, 1985) The problem with the homeomorphic embedding: definitions as

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

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

Configuration `f(g(x1)+1)` is embedded in `f(g(g(x1)+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 the Turchin relation. It is satisfied on the framed configurations.

When generalized, the framed pair produces more efficient computation than the red pair.

$$\begin{tikzcd}
 & \boxed{f(h(x))} \arrow[dl, "x=0"'] \arrow[dr, "x=x1+1"] & \\
f(0) \arrow[d] & & f(g(x1)+1) \arrow[d] \\
0 & & f(g(g(x1)+1))+1 \arrow[d] \\
 & & f(h(g(x1)))+1 \arrow[dll, "x1=0"'] \arrow[d, "x1=x2+1"] \\
f(h(0))+1 \arrow[d] & & f(h(h(x2)))+1 \arrow[dl, "x2=0"'] \arrow[d, "x2=x3+1"] \\
f(0)+1 \arrow[d] & f(h(0))+1 \arrow[d] & \boxed{f(h(g(x3)+1))+1} \arrow[d] \\
1 & f(0)+1 & \dots
\end{tikzcd}$$

An Example of Ackermanian Bad Sequence

```
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 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 $O(2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\}^{N})$.
- The length of the computation path until a pair of homeomorphically embedded configurations appears is $5 + N$.

Conclusion

Call stack transformations for call-by-name computations:

- have a rather complex structure and can produce rapid growth of the call stack length between configurations with branching;
- still not as powerful as Turing machines (the upper bound of the bad sequence length w.r.t. the homeomorphic embedding is Ackermanian, not hyper-Ackermanian).

The Turchin relation:

- a natural way to discover loops on computation paths using the history of the call stack that can be used as a termination criterion for both call-by-name and call-by-value semantics;
- can be used together with the homeomorphic embedding.

Bibliography

- J. Gallagher, L. Lavafe: *Regular approximation of computation paths in logic and functional languages*, 1996.
- I. Klyuchnikov: *Inferring and Proving Properties of Functional Programs by Means of Supercompilation*, 2010.
- 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