Termination analysis with recursive calling graphs

Teng Long,*, Wenhui Zhang

School of Information Engineering, China University of Geosciences (Beijing), 29#, Xueyuan Road, Haidian District, Beijing, PR China

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, 4th, South Fourth Street of Zhongguancun, Haidian District, Beijing, PR China

Available online 16 July 2015

Keywords:
Termination analysis
Green software programs
Size-change termination principle

ABSTRACT

As one of the significant aspects for green software systems, termination analysis is related to the optimization of the resource utilization. The approach for size-change termination principle was first proposed by Lee, Jones and Ben-Amram in 2001, which is an effective method for automatic termination analysis. According to its abstracted constructs (size-change graphs), the principle ignores the condition and return values for function call. In this paper, we devise a new construct including the ignoring features to extend the set of programs that are size-change terminating in real life. The main contribution of our paper is twofold: firstly, it supports the analysis of functions in which the returned values are relevant to termination. Secondly, it gains more accuracy for oscillating value change in termination analysis.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

With the development of information and communication technologies, it gives rise to the increasing popularity of electronic devices, raising serious environment concerns for the society. Trying to address these issues, there is a critical need to consider greenness in the whole life cycle of software systems.

A guarantee of right behaviors of software systems is significant, which can lighten greatly network loads and avoid wasting resources. Termination property is one of the properties that describe the right behaviors of the software system. Moreover, the termination analysis (Kuwahara et al., 2014; Heizmann et al., 2010; Codish et al., 2010, 2012; Farzan et al., 2015) is a much-studied topic. In real-life software systems, methods to derive termination properties from recursive and mutually recursive functions are useful in program analysis.

The size-change termination (SCT) principle which was presented in Lee et al. (2001) is simple but surprisingly rich enough to capture the progress of many real-life programs. Size-change graphs (SCG) are essential auxiliary constructs to support SCT. More importantly, the relations between input and output variables of a function call are restricted to the forms as $x > y$ or $x \geq y$. It operates over the variables whose “size” is well-founded based on the function call’s structure in the program. Once a combination of the relations has been found, termination follows if one variable at least is guaranteed to decrease. The termination analysis is unrelated to the order of the variables which is decided in SCT. Therefore, it is one of the approaches which are successfully applied in a large class of programs for termination analysis. There are some attempts (Ben-Amram and Genaim, 2014, 2013; Ben-Amram, 2009; Cook et al., 2013) to relax the well-founded restriction. However, the downside is that the SCG can only express the transitions involving decreases in well-founded partial orders. In fact, there are still many situations with oscillating value change, in which the traditional SCT approach is not applicable.

We present a technique for deriving program termination properties from size-change information, by constructing a recursive callings graph (RCG) and a set of extended size-change graphs (ESCGs) for the program. The former one describes the relation to function and function return based on locations in the program. The latter ones approximate the oscillating size change at each location in the program. The strong connected components in the RCG are an approximation of all sequences of function calls that could be idempotent.2 Our paper presents an important technique which could help streamline the application of the green software programs. The main contribution of our paper with recursive calling graphs is twofold: firstly, it supports the analysis that can handle recursive and mutually recursive functions in which return values are relevant to termination. Secondly, the approach extends the set of programs that are size-change terminating.

The paper is organized as follows: in Section 2, we introduce the syntax of the language used in this paper and the definitions that are related to size-change termination principle. We propose the definitions
of auxiliary constructs and do termination analysis in Section 3. Section 3 outlines how to record the returned values, and how to do a more precise analysis. An algorithm is proposed in Section 4. Section 5 illustrates the application of the approach on an example with self-recursive function callings. Related work is discussed in Section 6. We end with concluding remarks in Section 7.

2. Preliminaries

In this section, we will list the syntax of the language used in this paper and the definitions in size-change termination principle in Lee et al. (2001).

2.1. Language

The language used in this paper is a simple first-order call-by-value functional language, defined in Table 1. A variable can represent any expression. Similarly, a constant can represent any expression. An expression can be a conditional choice based on the equational theory (if..then..else). Functions are used to represent definitions about these variables. So does the expressions.

2.2. Size-change termination principle

Definition 1 (Size-change graph (SCG)). Suppose functions \( f \) and \( g \) are defined in program \( P \). A size-change graph \( G : f \rightarrow g \) for \( P \) is a set of labeled arcs \( x \xrightarrow{e} y \) or \( x \xrightarrow{\delta} y \) where \( x \in \text{Variables}(f), y \in \text{Variables}(g) \).

Functions \( f \) and \( g \) (the caller and callee) are respectively called the source and the target of \( G \). \( G \) is an abstraction of a call transition – from \( f \) to \( g \). The arcs in the graph are as abstract transitions.

Definition 2 (Closure). The closure of a set of size-change graphs is the smallest set \( cl(g) \) such that

- \( g \subseteq cl(g) \)
- If \( G_1 : f \rightarrow g \) and \( G_2 : f' \rightarrow g' \) are in \( cl(g) \), then \( G_1 ; G_2 \in cl(g) \), \( G_1 ; G_2 : f \rightarrow g' \) with arc set \( E \) defined below: \( E = \{ x \xrightarrow{\delta} y \mid w . x \xrightarrow{\epsilon} y \} \) or \( x \xrightarrow{\delta} y \), \( w \in \{ >, \geq, \leq \} \) and \( x \xrightarrow{\delta} y \) are respectively arcs of \( G_1 \) and \( G_2 \).

Size-change graph \( G \) is idempotent if \( G ; G = G \). A self-recursive function call can be expressed by an idempotent size-change graph.

Theorem 1 (Size-change termination principle). Program \( P \) is SCT terminating iff every idempotent \( G \) in \( cl(g) \) has an arc \( z \xrightarrow{\delta} z \).

The SCT is described in Lee et al. (2001) as follows: A program terminates on all inputs if every infinite call sequence (following program control flow) would cause an infinite descent in some data values.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( e )</th>
<th>( op )</th>
<th>( c )</th>
<th>( e )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
</tr>
<tr>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
</tr>
<tr>
<td>( \text{Var} )</td>
<td>( \text{Const} )</td>
<td>( \text{Exp} )</td>
<td>( \text{Exp} )</td>
<td>( \text{Exp} )</td>
<td>( \text{Def} )</td>
</tr>
<tr>
<td>( \text{Prim} )</td>
<td>( \text{Const} )</td>
<td>( \text{Exp} )</td>
<td>( \text{Exp} )</td>
<td>( \text{Exp} )</td>
<td>( \text{Exp} )</td>
</tr>
<tr>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
<td>( \in )</td>
</tr>
<tr>
<td>( \text{Variables} )</td>
<td>( \text{Constant} )</td>
<td>( \text{Expression} )</td>
<td>( \text{Expression} )</td>
<td>( \text{Expression} )</td>
<td>( \text{Expression} )</td>
</tr>
</tbody>
</table>

A closure set of size-change graphs (sets of abstract transitions) is used to express the set of all the possible call sequences. All of the idempotent ones in closure set stand for the infinite call sequences. If a descent arc “\( \xrightarrow{\delta} \)” can be found, it satisfies the “infinite descent” in the principle.

3. Termination analysis

The termination analysis for programs with nested self-recursive functions over integer domain is complex, because of the complicated “size change” situation. The possible cases are as follows:

1. The value of variables in the program declines all the time.
2. The value of variables in the program increases in each function call.
3. The value of variables in the program is oscillating, such as in the McCarthy’s 91 function.

There are only two arc forms in size-change graphs, such as \( x \xrightarrow{\delta} y \) or \( x \xrightarrow{\delta} y \), therefore, it can deal with the first case. In our work, additional variables and a new auxiliary construct are used to address the latter two cases.

3.1. Additional variables

In this work, additional variables consist of counter variables and returned variables. The former one is used to record the number of times for some function call/return, while the latter one is used to record the returned value.

The returned values are always ignored in available approaches, so functions in which the returned values are relevant to termination cannot be analyzed correctly. Adding extra variables in the program is the first step to extend the range of programs for termination analysis.

A set of additional variables \( V_{\text{add}} \) consists of a set of returned variables and a set of counter variables. After augmenting the program with additional variables, the set of variables \( V \) in the program will augmented as \( V \cup V_{\text{add}} \).

3.2. Definitions of auxiliary constructs

We construct the recursive calling graph (RCG) to describe the control flow for the augmented program which can express not only the function calls but the function returns. The transitions in a RCG are described as ESCGs which are extended by transforming the arc types from \( x \xrightarrow{\delta} y \) to \( x \xrightarrow{\delta} y \) where \( \delta \in Z \) is the changed value in the transition.

The changes of values in ESCGs have to be satisfied in size reducing. In other words, the corresponding code cannot be revisited, and the recursive calling graph cannot result in any infinite cycle.

3.2.1. Extended size-change graph

Definition 3 (Extended size-change graph (ESCG)). Suppose functions \( f \) and \( g \) are defined in program \( P \). Location \( a \) is the place where the value of variables function \( g \) changed in program \( P \). An extended size-change graph \( B_a : f \rightarrow g \) for \( P \) is a set of labeled arcs \( x \xrightarrow{\delta} y \) where \( x \in \text{Variables}(f), y \in \text{Variables}(g), \delta \in Z \) is the changed value for function calls.

Suppose functions \( f, g \) are defined in the program. An ESCG \( B_a \in B \), \( B_a : f \rightarrow g \) for the program is a set of labeled arcs \( x \xrightarrow{\delta} y \) where \( \delta \in Z, x \in \text{Variables}(f), y \in \text{Variables}(g) \). Functions \( f \) and \( g \) are respectively called the source and the target of \( B(\text{source}(B) = f, \text{target}(B) = g) \).
Definition 4 (E-Multipath). An e-multipath $\mathcal{M}$ is a transition sequence $B_1, B_2, B_3, \ldots$ such that $\text{target}(B_i) = \text{source}(B_{i+1})$ for $i = 1, 2, \ldots$. An e-thread is a connected path of arcs in $\mathcal{M}$ that starts from some $B_0$, $t \geq 1$, $\theta = z_{i-1}^{\delta_1} z_{i+1}^{\delta_2} \ldots$ with each $\delta_i \in \mathbb{Z}, z_i \in V, j \in \mathbb{N}$.

Definition 5 (Composition). The composition of two transitions $B : f \rightarrow g$ and $B : g \rightarrow h$ is $B : f \rightarrow h$ with arc set $E$ defined below. Notation: write $x \xrightarrow{\delta_1} y \xrightarrow{\delta_2} z$ if $\delta_1, \delta_2$ are the changed values of variables in each function, $x \rightarrow y$ and $y \rightarrow z$ are respectively arcs of $B$ and $B'$.

Arc is a set of arcs

$Arc = \{x \xrightarrow{\delta_1} y \xrightarrow{\delta_2} z \in B, y \xrightarrow{\delta_3} z \in B'\}$

For nested functions, if there are inner function $f_{out}$ and outer function $f_{out}$, we can use composition of ESCGs to express the function call in nested functions. There are ESCGs $B_1 : f_{out} \rightarrow f$ and $B_2 : f \rightarrow f_{out}$, and variables $r \in V_{add}$, $x \in \text{Variables}(f_{in})$ and $y \in \text{Variables}(f_{out})$. There are arc $x \xrightarrow{\delta_1} y$ in $B_1$ and arc $r \xrightarrow{\delta_2} y$ in $B_2$, where $\delta_1 \in \mathbb{Z}, \delta_2 \in \mathbb{Z}$. We use $B = B_1 \cdot B_2$ to show the connection between the returned values and the termination of programs.

Definition 6 (M-Transition). An m-transition is a transition as $B : f \rightarrow g$ which is the composition for all the transitions in a multipath $\mathcal{M} : B_1, B_2, \ldots, B_n$, where $\text{source}(B_1) = f, \text{target}(B_n) = g$.

Assume an arc $x \xrightarrow{\delta_1} y$ in m-transition $B$ comes from $x \xrightarrow{\delta_1} z_1 \xrightarrow{\delta_2} z_2 \xrightarrow{\delta_3} y$. We should consider the changed value in arcs $z_1 \xrightarrow{\delta_1} z_2$ and $z_2 \xrightarrow{\delta_2} z_3$ which express the self-recursive function calls. All the possible call multipaths in the multipath should be considered by multiplying the visiting number of times at the locations in the program.

Definition 7 (Tendency transition). Let $B : f \rightarrow g$ be an m-transition for a multipath $\mathcal{M} : B_1, B_2, \ldots, B_n$, where $\text{source}(B_1) = f, \text{target}(B_n) = g$. We define $B$ with integer labeled arcs as $x \xrightarrow{\delta} y, \delta \in \mathbb{Z}$, and the tendency transition $B_T$ of $B$ with symbol labeled arcs as $x \xrightarrow{\delta} y, \delta \in \mathbb{Z}$.

Function $\text{Tend}$ is defined as $B \rightarrow B_T$

$\text{Tend}(B) = \text{Tend}(f \rightarrow g) = (x \xrightarrow{\delta} y | x \xrightarrow{\delta} y, E > 0, x \in \text{Variable}(f), y \in \text{Variable}(g))$

$\cup (x \xrightarrow{\delta} y | x \xrightarrow{\delta} y, E < 0, x \in \text{Variable}(f), y \in \text{Variable}(g))$

$\cup (x \xrightarrow{\delta} y | x \xrightarrow{\delta} y, E = 0, x \in \text{Variable}(f), y \in \text{Variable}(g))$

Function $\text{Tend}$ transform transition $B$ with integer labeled arcs to the tendency transition $B_T$ with symbol labeled arcs that express the ascent or descent in some data value.

We compose many transitions in an e-multipath. The result for composition seems like a new transition which describes the whole changed value after all the function calls.

E-Idempotent $B$ is that $\text{Tend}(B) = \text{Tend}(B; B)$.

Definition 8 (Deterministic). Given an m-transition $B$, if there is no arc in the tendency transition $B_T$ of $B$, transition $B$ is non-deterministic. Otherwise, it is deterministic.

3.2.2 Recursive calling graph

Definition 9 (Recursive calling graph). The recursive calling graph (RCG) for program is $\mathcal{G} = (\mathcal{P}, B, \mathcal{L})$:

- $\mathcal{P} = \{ f | \text{function name} \}$: A set of points in RCG consists of the function names in the program. We define a function $\text{Variables}(\varphi)$ as program points $\varphi$.
- $B = \{ B_i \}$ is one of the locations in the program, $B_i$ is an ESCG: A set of ESCGs; to express the transitions from one point to another point in the RCG.
- $\mathcal{L} = \{ \mathcal{L}(v) | v \in \text{Variables}(p), p \in \mathcal{P} \}$: A set of labels (expressions $\mathcal{L}(v)$ is related to some variable $v$): to express the conditions of function call.

Suppose positions $p_1, p_2 \in \mathcal{P}$ are defined in the program. There are transitions $B_0, B_1, B_2 \in \mathcal{B}$, ESCGs $B_0 : f \rightarrow g$ with labeled arcs $x \xrightarrow{\delta} y$ and $v \xrightarrow{\delta} u$, $B_0 : g \rightarrow f$ with labeled arcs $y \xrightarrow{\delta} x'$ and $u \xrightarrow{\delta} v'$ where $\delta_1, \delta_2, \delta_3, \delta_4 \in \mathbb{Z}, x, v \in \text{Variables}(f), y, u \in \text{Variables}(g)$.

An example of RCG is in Fig. 1. Nodes in the RCG are the program points. The program point 1 describes function $f$ with calling condition $L_1$. By contrast, the program point 2 describes function $g$ with calling condition $L_2$. Blocks in the RCG are transitions in the program. There are transitions between the points with conditions $L_1$ and $L_2$ respectively as follows:

- $1 \rightarrow 2: f(x, v) \rightarrow g(y, u), L_1$
- $2 \rightarrow 1: g(y, u) \rightarrow f(x, v), L_2$

The blocks $f(x, v) \rightarrow g(y, u)$ at location $a$ and $g(y, u) \rightarrow f(x, v)$ at location $b$ are described as ESCGs.

All the calling condition can be expressed in the form as $lg \geq rg$ or $lr > rg$, where $lg, rg \in \mathbb{V} \cup \text{const}$ (const is a set of constant). There is an equation size$(lg, rg) = \text{abs}(lg - rg)$, where $\text{abs}(\mathcal{E})$ is the absolute value of expression $\mathcal{E}$.

For example, under condition $n > 0$, there is $\text{size}(n, 0) = \text{abs}(n)$, while under condition $100 \geq n$, there is $\text{size}(100, n) = \text{abs}(100 - n)$.

3.3. Termination analysis with RCG

Lemma 1. Given a RCG $\mathcal{G} = (\mathcal{P}, B, \mathcal{L})$ for a program, every strong connected component in $\mathcal{G}$ has an e-idempotent m-transition.

Proof. Every strong connected component $C$ in $\mathcal{G}$ can be expressed as a sequence of nodes: $P_0, P_1, \ldots, P_n, P_0$, where each $P \in \mathcal{P}$. We can get an e-multipath $M$ from $C$: $f \rightarrow g_1 \rightarrow g_2 \rightarrow \cdots \rightarrow f$ where $f \in P, g_j \in P_{j+1}, j \in \{0 \ldots N\}$.

The m-transition of $M$ is $B : f \rightarrow f$. An arc as $x \xrightarrow{\delta} x'$, where $x, x' \in \text{Variables}(f)$ can be constructed in $B$. 

![Fig. 1. The recursive calling graph.](image-url)
There are different situations in SCC as follows:

- There is an arc \( x \rightarrow y \) for a path an arc as \( f \). Following Lemma 1, there is an e-idempotent transition \( \delta \). Therefore, each SCC in RCG is related to an e-idempotent transition.

There is an e-thread as \( x \rightarrow y \rightarrow \ldots \rightarrow x ', x, x ' \in \text{Variables}(f), y_j \in \text{Variables}(g), j \in [0..n] \). We have

\[
E = \sum_{j=0}^{N} (\delta_j)
\]

In this case, there are no other loops in C. The whole changed value \( E \) for a path an arc as \( x \rightarrow x ' \) is just the sum of each changed value during the loop.

For deciding whether a transition is e-idempotent or not, we should consider the situations mentioned above:

- \( E < 0 \): There is size(con,x) \( \rightarrow \) size(con',x) in \( \text{Tend}(B) \), where \( con \in \text{const} \) and \( x \in V \), where \( \text{const} \) is a set of constant.
- \( E > 0 \): There is size(x,con) \( \rightarrow \) size(x,con') in \( \text{Tend}(B) \), where \( \text{con} \in \text{const} \) and \( x \in V \), where \( \text{const} \) is a set of constant.
- \( E = 0 \): There is size(g,r,g') \( \rightarrow \) size(g,r,g') in \( \text{Tend}(B) \), where \( \text{lg} \in V \) and \( \text{rg} \in \text{const} \), or \( \text{rg} \in V \) and \( \text{lg} \in \text{const} \).

Thus, each SCC in RCG is related to an e-idempotent transition.

Lemma 2. Given a RCG \( G \) for an augmented program, any maximal strong connected component of \( G \) satisfies a decreasing ordering on a size function for some condition L.

Proof. Every maximal strong connected component C in \( G \) can be expressed a sequence of nodes as: \( P_1, P_{i+1}, \ldots, P_n, P_0 \), where each \( P_i \in \mathcal{P} \). By Lemma 1, we can construct an e-idempotent transition \( B : f \rightarrow f' \) in which there exists an arc \( x \rightarrow x ' \) where \( x, x ' \in \text{Variables}(f) \). There is an arc \( x \rightarrow x ' \) in \( \text{Tend}(B) \) respectively.

Following Lemma 1, there is \( \text{Tend}(B) \) with some changed tendency \( R \) related to \( C \). There is size(v,con), where \( v, v \in V \). According to the changed tendency " - " for \( v \) (the descent in value of "v"), it violates the condition \( L : v > \text{con} \). Or there is size(con,v), where \( v, v \in V \). According to the changed tendency " + " for \( v \) (the ascent in value of "v"), it violates the condition \( L : \text{con} > v \).

Theorem 2 (Extended size-change termination principle). Let \( G = (\mathcal{P}, B, L) \) be a RCG for an augmented program. If every MSCC of \( G \) is well-founded for some size function, then all of the functions terminate on all inputs.

Proof. Each variable in \( V \) is relate to some \( P \in \mathcal{P} \). Each changed tendency \( R \) is related to the transition \( B \in B \). Each size function \( \text{size}(g,rg) \) is relate to some transition condition \( L \in L \). For each loop in \( G \), there exists a size function which is well-founded ordering by Lemma 2. We can get that there is no infinite loops in the program. The program will terminate for all the inputs.

4. Algorithm: R_Termination

For some functions whose returned value may be related to the termination of the program, some additional constructs may be used. We add the returned variable \( r \) and the counter variables \( \text{co} \) in the program.

4.1. Preprocessing

4.1.1. Capturing returned values in the program

Since the language used here is a functional language, it seems appropriate to capture the returned values of functions. Sometimes these returned values are required for termination analysis (as in the McCarthy's 91 function). The first extension proposed here is to add in a global returned value for function calls in the program. For McCarthy's 91 function, the process responding to the returned value should be expressed as a composition between two extended size-change graphs \( G_a \) and \( G_b \) at location \( a \) and \( b \) in the program (3).

4.1.2. Adding counter variables in extend program

It is possible to composite the transitions in the RCG, and analyze the termination of the program for McCarthy's 91 function directly. However, it is impossible to detect the termination of the McCarthy's 91 function. The analysis is therefore extended to take advantage of other available information.

Sometimes, the value of some variable decreases in several steps but increases in some step, therefore, we cannot get the final decision that the program terminate or not. But in the real world, it will finally terminate. The non-deterministic is caused by missing the relation between the nested functions. The expressions \( \text{co} = \text{co} + 1 \) at location \( c \), \( \text{co} = \text{co} - 1 \) at location \( b \) and the condition \( \text{co} > 0 \) in Dec. 3 can describe the relations between function calls and function returns.

4.1.3. Constructing size functions

We will deal with each calling conditions by constructing a size function defined on a well-founded domain. Such function is related to some variable in a calling or return condition. The condition of the transition will be unsatisified according to the oscillating value. If we cannot construct such size function for some MSCC, it means that there are still some loops in the execution of the program. Otherwise, the program will terminate.

4.2. Algorithm: R_Termination

We propose an algorithm R_Termination to describe the analyzing process after augmenting and transforming.

After augmenting, there is a set of variables that consists of the original variables and the additional ones \( V_{\text{add}} = \{r \cup \text{co}\} \), where including returned value \( r \) and counter value \( \text{co} \). After transforming the augmented program to a RCG \( G' \), we take \( G = (\mathcal{P}, B, L \cup L_{\text{co}}) \) as the input for algorithm R_Termination. We take \( \mathcal{P} \) as a set of points in the graph, \( B \) as a set of blocks (ESCGs), \( L \) as a set of transition conditions from some node to another one, \( L_{\text{co}} \) as the condition with counter variable \( \text{co} \) to express the constraint that there is no more number of times for function return than for function call:

1. We analyze the termination of the augmented program by dealing with the RCG \( G' \) and the set of locations \( A \). Firstly, we construct a set of ESCGs for each location in the program.
2. Then we deal with the loops in \( G' \) by function Loop(\( G' \)), if there is no loop in the RCG, such that all functions terminate on all inputs according to Theorem 1.
3. Otherwise, we decompose the RCG \( G' \) into many MSCCs \( \{C_0, \ldots, C_k\} \). Each MSCC \( C_i \) consists of points, ESCGs and conditions \( C_i \in (\{P_j, \ldots, P_{j+k}\}, (B_j, \ldots, B_{j+k}), (L_j, \ldots, L_{j+k})) \).
4. For each $C_{i}$, there exists an idempotent transition $B_{i}$ according to Lemma 1. We composite ESCGs in $C_{i}$, get a set of compositions $S$. For all techniques which is applied in McCarthy’s 91 function (Manna and van den Dries, 2003). If we cannot construct some size function $\text{size}(v) > \text{size}(v')$, we relate to some $L$, then we should remove the condition respectively. The condition $L$ will finally be unsatisfied according to the tendency $R$ for some variable $v$ in the SCC.

5. We deal with the sub-RCG recursively. If we cannot find such size function, we have no proof of termination for the program on some inputs.

7. If we can construct size function for MSCC in each iteration, the program terminates for all the inputs.

Algorithm 1. R-Termination ($G$).

1: $\text{SCG}(G, A)$ into a set of ESCGs $B = \{B_{n}, B_{n}, \ldots \}$, $a, b, \ldots \in A$
2: $\text{Loop}(G)$
3: if there are loops then
4: $\text{Decompose}(G)$ into a set of MSCCs $C_{0}, \ldots, C_{k}$
5: for $i = 0; i \leq k; i + +$ do
6: $\text{Composition}(C_{i}, B)$ into a set of composition of ESCGs in $S = \{B_{m}, B_{n}, \ldots \}$, $a, b, \ldots \in B$
7: $R = \text{Tend}(S)$
8: if $\text{size}(v) > \text{size} (v')$ then
9: $G' = \text{Remove}(G, L, B)$
10: $\text{Loop}(G')$
11: else
12: halt and print “there is no proof of termination for some inputs”
13: end if
14: end for
15: else
16: there are no loops
17: print “the program terminates for all inputs”
18: end if

We declare the functions in Algorithm 1 as follows:

- $\text{SCG}(G, A)$: We construct ESCGs for each location in the program.
- $\text{Loop}(G)$: We analyze that whether there are loops in $G$.
- $\text{Decompose}(G)$: We decompose the graph as a set of loops which contain nodes, transitions and transition conditions.
- $\text{Composition}(C_{i}, B)$: We composite ESCGs in the MSCC $C_{i}$. The idempotent one is related to a loop.
- $\text{Tend}(S)$: We get tendency $R$ by doing composition with each transition $B_{i}$ between each pair of nodes $P_{i}$ and $P_{j+1}$ in ESCC $C$.
- $\text{Remove}(G, L, B)$: We remove the condition $L \in L$ relate to some ESCG in $C$ from $G$, get $G'$.

We extend the program by adding many constructs. And then we take the RCG $\mathcal{G}'$ of such program as the input for Alg. R-Termination. Each SCC in $\mathcal{G}'$ is the possible loop we should cut off. If all the possible loops in the program can be cut off, the program will terminate. Otherwise, there must exist some loops, which is the reason for the non-termination of the program.

5. Example

At this stage we introduce an example to illustrate the analysis techniques which is applied in McCarthy’s 91 function (Manna and McCarthy, 1970). For all $n \leq 101$, the returned value of the function $f(n)$ is

91. Neither SCT nor affine-SCT style (Anderson and Khoo, 2003) analysis work for this function, because the returned value of the function and the relative numbers of calls are relevant to its termination.

$$f(n) = \begin{cases} a & \text{if } n > 100 \\ b : f(c : f(n + 1)) & \text{otherwise} \end{cases}$$

(3)

Considering the program of McCarthy’s 91 function (3), there are a nested loop with two designated locations (the location $c$ for the inner function and the location $b$ for the outer function) in a self-recursive function. Despite its simplicity, there is no such sequence of size-change graphs that would make this program size-change terminate. There are only relations $>$ and $\geq$ in the approach of size-change principle. The auxiliary constructs for the approach are size-change graphs which include transitions for non-decreasing relations in function callings. Applying size-change principle in the example, we can get the size-change graph in Fig. 2. Because of the action $n + 11$, there are no transitions in the size-change graph $G_{f, \mathcal{G}}$. The traditional size-change principle cannot deal with the program with recursive function callings.

![Fig. 2. The SCG of The McCarthy's 91 Function in SCT.](image)

But we can get that the inner loop will terminate because when the value of variable $n$ is greater than 100, the execution of the program will move to location $a$ and then the returned value $n - 10$ will be taken as an input value to the outer function (at location $b$). Because it is impossible that the number of times for function return is more than the number of times for function call, the expression $\text{size}(a, \mathcal{G})$ shows the size of some variable is oscillating but finally decreased in each iteration.

The main steps of our approach are as follows: firstly the original program is augmented by additional variables (including returned variables and counter variables), which are used to record the returned value and the number of times for function call/return, respectively. Secondly, we transform the extended program to a recursive calling graph. Then we analyze the maximal strong connected components (MSCCs) of the RCG. The program is guaranteed to terminate if all the MSCCs are well-founded. For the motivated example: the McCarthy’s 91 function, we have: Step 1: The first step is used to augment program with additional variables $r$ and $co$. We assign an initial value 0 to both counter variable $co$ and returned variable $r$. There are $co = co + 1$ at location $a$ (recording the number of times for function call), $co = co - 1$ at location $b$ (recording the number of times for function return) and $r = n - 10$ at location $a$ (recording the returned value “ $n - 10$”).

- $co = 0$
- $r = 0$
- $f(n) = \begin{cases} a & \text{if } n > 100 \\ T : n - 10 & \text{otherwise} \end{cases}$

else

$$b : f(c : f(n + 1)); co = co + 1$$

(4)
compositions are in Fig. 5. $B_2, B_3$ is the composition of $B_x$ and $B_y$, which describes the changed value for function return 2 → 1 and 2 → 2. The composition $B_x, B_2, B_3$ is one of the transitive closures of the transitions. Because there is no more visiting times for function return than function call, we can get $co > 0$ for each point in the program. According to this, the visiting numbers of times at location $b$ are no more than the numbers at location $c$.

$$
\begin{array}{ccc}
B_2 & B_3 \\
\xrightarrow{n-10} n' & n \xrightarrow{-11} n'
\end{array}
$$

There is $E = Δ_1 - Δ_2$, according to the condition $co > 0$, we can get that $(Δ_1 \text{ div } 10) > (Δ_2 \text{ div } 10)$, so that $E > 0$ is true.

We can handle the MSCCs with the combinations of ESCGs. The MSCC is well-founded if there is at least one descending sequence. After removing the descending condition from the RCG, the process for dealing with the MSCC in the remain part of RCG is repeated several times until there is no MSCC or no descending sequence. In this example, the parameters are in Table 2.

The MSCC that consists of points $P_1, P_2$ and the composition $B_x, B_2, B_3$ for location $a, b, c$, has a descending sequence $size(100, n) > size(n, 100)$. After removing condition $100 \geq n$ from RCG, the MSCC consists of points $P_2$ and composition $B_x, B_3$ for location $a, b$, with a descending sequence $size(n, 100) > size(n', 100)$. After removing condition $n > 100$, there is no cycle in the RCG. The program will terminate.

### 6. Related work

A constraint transition system (CTS) is an abstract program with a finite description. The nodes in a control flow graph are finite, so does a set of variables associated with each nodes. Monotonicity constraints were first introduced to termination analysis in 1991 (Sagiv, 1991). It considers relations $> \geq$, which means that the proof of termination bases on a non-increasing sequence. And there are a series of termination analysis approaches applying in both functional programs and imperative programs (Ben-Amram and Codish, 2008; Spoto et al., 2010; Alias et al., 2010). Codish (2005) extended the SCT to monotonicity constraints and the integers in theory. They argued that in imperative programs, in which the constraints are not of the SCG type, but terminate over the integers. Their approach (Codish et al., 2012) for checking termination by SAT solving. But the termination analysis by abstraction functions is according to their syntactic structure. We generate auxiliary constructs by using relations instead of functions. The paper (Ben-Amram, 2011) was motivated from the discussion to present an algorithm for constructing global ranking functions for terminating the CTS in integer domain. Zuleger et al. (Zuleger et al., 2011; Colcombet et al., 2014) applied CTS over integer domain in the context of cost analysis. But the control flow graph in CTS cannot express the self-recursive function call such as $f(f(n+11))$. And the relations with label $>$ and $\geq$ may be too coarse to lead to more precise analysis.

In papers (Avery, 2006; Schmitz, 2014) with bound analysis, another method of SCT was introduced. Some approach deals with imperative languages such as C programs without function calls by bound analysis. In the method, the graphs are standard SCT graphs, although including as well as the $x < y'$ relation. Termination is guaranteed in the same way as for SCT. But it cannot deal with the programs with function calls.

The approach in paper (Ben-Amram and Genaim, 2014) can deal with ackman function but not McCarthy's 91 function. This
kind of function is different from the functions that they can deal with. Primarily, the returned value of the function and the relative numbers of calls are relevant to its termination. Meanwhile, in the McCarthy’s 91 function, the value of some variable decreases in many steps but increases in some steps.

For dealing with the functional programs, paper (Manolios and Vroon, 2010) proposed a method with bounded domain. This more powerful termination principle may create the control flow graph in different ways by choosing call sites to be flow-points rather than function names. But it still cannot express the transition for returning values. The calling context graph (CCG) related to the McCarthy’s 91 function is in Fig. 6.

Fig. 6. The CCG of the McCarthy’s 91 function.

The nodes 1 and 2 are described as follows:

1. \( f, \lfloor n \leq 100 \rfloor, f(n+11) \)
2. \( f, \lfloor n > 100 \rfloor, f(f(n+11)) \)

And the size-change graphs are as Fig. 7.

\[
\begin{align*}
\phi_1 & : 1 \rightarrow 1 \\
\phi_2 & : 1 \rightarrow 2 \\
\phi_3 & : 2 \rightarrow 1 \\
size(n) & \not\geq size(n')
\end{align*}
\]

Fig. 7. The SCGs of The McCarthy’s 91 Function in CCG.

The CCG approach cannot solve the problem because of losing the relation (arcs in \( \phi_2, \phi_3 \) in Fig. 7) between the returned value of inner function and the input value of the outer function in nested recursion functions.

7. Conclusions

The first main contribution of this paper is the design of an auxiliary construct named recursive calling graph to express the nested functions, which, in contrast to size-change graphs, is more generally applicable (with respect to the relation between the number of calling times for the inner function and the outer function found and returned value considered). And we provide an algorithm that can handle programs with oscillating value change, with proper preprocessing (additional variables).

More generally, our work establishes a termination analysis over integers, which can handle nested functions in which returned values are relevant to termination. And it can finally detect function call sequences that repeatedly oscillate a value that eventually violates some calling conditions. The approach could help optimizing the allocation of resources by extending the set of programs that are size-change terminating.

There is nevertheless room for many improvements. We will consider the situation when termination depends on a fairness schedule in concurrent programs.

Acknowledgments

This work is supported by the Fundamental Research Funds for the Central Universities under Grant no. (2-9-2014-011) (CUGB), the National Program on Key Basic Research Project of China (973 Program) under Grant no. (2014CB340701) and the National Natural Science Foundation of China under Grant no. (61272135).

References


Cook B, See A, Zuleger F. Ramsey vs. lexicographic termination proving. In: Tools and algorithms for the construction and analysis of systems—19th international symposium, TACAS 2013, held as part of the European joint conferences on theory and practice of software, ETAPS 2013, Zürich, Switzerland; April 7–12, 2013. p. 25–46. doi:10.1007/978-3-642-36742-7_2.

Cook T, See A, Zuleger F. Ramsey vs. lexicographic termination proving. In: Tools and algorithms for the construction and analysis of systems—19th international symposium, TACAS 2013, held as part of the European joint conferences on theory and practice of software, ETAPS 2013, Zürich, Switzerland; April 7–12, 2013. p. 25–46. doi:10.1007/978-3-642-36742-7_2.


