THE NATIONAL UNIVERSITY
of SINGAPORE

School *of* Computing
Lower Kent Ridge Road, Singapore 119260

TRA9/03

Affine-Based Size-Change Termination

Anderson HUGH and Siau Cheng KHOO

September 2003

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Affine-based Size-change Termination

Hugh Anderson and Siau-Cheng Khoo

Department of Computer Science
School of Computing
National University of Singapore
hugh,khoosc@comp.nus.edu.sg

Abstract. The size-change principle devised by Lee, Jones and Ben-Amram, provides an effective method of determining program termination for recursive functions over well-founded types. Termination analysis using the principle involves the classification of functions either into size-change terminating ones, or ones which are not size-change terminating. Size-change graphs are constructed to represent the functions, and decreasing parameter sizes in those graphs that are idempotent are identified. In this paper, we propose a translation of the size-change graphs to affine-based graphs, in which affine relations among parameters are expressed by Presburger formulæ. We show the correctness of our translation by defining the size-change graph composition in terms of affine relation manipulation, and identifying the idempotent size-change graphs with transitive closures in affine relations. We then propose an affine-based termination analysis, in which more refined termination size-change information is admissible by affine relations. Specifically, our affine-related analysis improves the effectiveness of the termination analysis by capturing constant changes in parameter sizes, affine relationships of the sizes of the source parameters, and contextual information pertaining to function calls. We state and reason about the corresponding soundness and termination of this affine-related analysis. Our approach widens the set of size-change terminating functions.

1 Introduction

There are many approaches to termination analysis. For example, Colón and Sipma [3] use linear ranking functions to prove termination of program loops. Another approach could be to limit the *language* to ensure termination: if any function defined over an inductive type is restricted in the form of the definition to one using the elimination rule of the type, then the function is known to terminate [5]. An alternative method derives from the observation that, in the case of a program with well-founded data,

> *"a program terminates on all inputs if every infinite call sequence would cause an infinite descent in some program values"* [11]

In this framework, we have a finite number of functions, with the underlying data types of at least some of the parameters expected to be well-founded. In addition, the only technique for repetition is recursion. Given this framework, we can give the intuition behind the size-change termination method. We begin by considering the *size* of the function parameters. If the type of the parameter was a natural number, then the size of this parameter could be its value, and we cannot reduce the size of this natural number indefinitely, as eventually it will reach zero and may no longer be reduced. If the type of the parameter was an inductively defined list of items, then the size of this parameter could be its length, and again we cannot reduce the size of this list indefinitely, as eventually it will reach the empty list and (again) may no longer be reduced. Both of these parameter types are well-founded, and it is on the use of these data types that the termination method relies.

Consider a non-terminating program. This program can only be non-terminating through an infinite recursion through at least one function entry point, since the number of different functions is finite. If we consider the chain of possible function calls within each one of those functions, and determine the size-change of each of its parameters between successive calls, then if any one of those parameters reduces on each call, we have a conflict. In particular, if it reduces infinitely often, then the data is not well-founded. As a result of this, we can assert the contrapositive, an observation that may be made over a program with well-founded data types which precludes it from being non-terminating; *"if every infinite call sequence in a program would cause an infinite descent in some program values then the program terminates on all inputs"*. This gives the intuition behind the size-change termination method. Note that size-change termination is not a general termination method, but it is still useful.

In [11], Lee, Jones, and Ben-Amram present a practical technique for deriving program termination properties using the size-change termination method (hereafter called LJB-analysis), by constructing a set of size-change graphs for the program. These size-change graphs approximate, to a set of four symbolic values, the relation between the sizes of source parameters and destination arguments for each call to a function. We believe a more refined representation of size-change relation can widen the set of size-change terminating functions. In particular, Presburger formulæ, or affine relations in particular, may be a good candidate for encoding size-change graphs, for the following three reasons:

1. They allow the capturing of constant changes in parameter size. This allows the effect of constant increment and decrement to be cancelled out during the analysis.
2. They can express size change of a destination argument by a linear combination of source parameters. This enables more accurate representation of size change.
3. They can constrain the size change information with information about call context, thus naturally extending the analysis to be context-sensitive. The LJB-analysis method ignores test conditions, but these can be expressed naturally using Presburger constraints.

To illustrate that our termination analysis is strictly more powerful than the LJB-analysis, we list below three example programs in a simple first-order function definition language which can be successfully analyzed for termination by our analysis, but which

are outside the scope of LJB-analysis. The first example alternately increases and reduces a parameter on successive function calls corresponding to the first reason above.

```
f(m) = if m ≤ 0 then 1 else g(m + 1);
g(n) = if n ≤ 0 then 1 else f(n − 2);
```

An example of the second reason given is the following function in which the LJB-analysis is unable to establish that the first argument must decrease:

```
k(m, n) = if m ≤ 0 then 1 else k(m − n, n + 1);
```

An example of the third reason given is the following function in which the variables are all natural numbers, and only one of the two calls can be performed, constrained by the condition $m < n$. The LJB-analysis fails to establish termination:

```
j(m, n) = if m + n = 0 then 0
          else if m < n then j(m − 1, n + 1)
               else j(m + 1, n − 1);
```

In this paper, we elaborate on the use of affine relations to capture size-change information, and show the soundness and termination of our analysis.

In Sections 2 and 3, preliminary definitions and notation used in the paper are outlined along with a brief outline of the LJB size-change termination method. In Section 4 we introduce the concept of affine-based size-change graphs, and provide a translation of LJB's size-change graphs to affine-based size-change graphs. In Section 5 we give an algorithm for building the closure of affine size-change graphs, and explore various properties of the algorithm. In Section 6 we show how this new process can establish termination properties for a wider range of functions than the original method, using a simple example. In Section 7 we outline the relation of this work to others, and conclude with some observations about the direction of our research in this area.

2 Preliminaries

x	$\in$ **Var**	⟨ Variables ⟩
op	$\in$ **Prim**	⟨ Primitive operators ⟩
f, g, h	$\in$ **FName**	⟨ Function names ⟩
c	$\in$ **Const**	⟨ Constants ⟩
e	$\in$ **Exp**	⟨ Expressions ⟩
	$e ::=$ **if** e_0 **then** e_1 **else** e_2	
	$\mid x \mid c \mid e_1\ op\ e_2 \mid f(e_1, \ldots, e_n)$	
d	$\in$ **Decl**	⟨ Definitions ⟩
	$d ::= f\ x_1 \ldots x_n = e$	

Table 1. The language syntax

In order to concentrate on the mechanism behind our termination analysis, we choose to work on a simple first-order functional language, defined in Table 1. Additional lan-

guage features can be included in the subject language. As these will only complicate the generation of the initial set of size-change graphs, but not the analysis, we prefer not to include them in our discussion.

Formulæ:	$\phi \in \mathtt{F}$	⟨ Formulæ ⟩
	$\phi ::= \psi \mid \{[v_1, \ldots, v_m] \to [w_1, \ldots, w_n] : \psi\}$	
	$\psi ::= \delta \mid \neg\psi \mid \exists v.\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$	
Size Formulæ:	$\delta \in \mathtt{Fb}$	⟨ Boolean expressions ⟩
	$\delta ::=$ True $\mid$ False $\mid a_1 = a_2 \mid a_1 \neq a_2$	
	$\mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2$	
	$a \in \mathtt{Aexp}$	⟨ Arithmetic expressions ⟩
	$a ::= n \mid v \mid n \star a \mid a_1 + a_2 \mid -a$	
	$n \in \mathcal{Z}$	⟨ Integer constants ⟩

Table 2. Syntax of Presburger formulæ

Affine relations are captured using Presburger formulæ, and each such relation has explicitly identified source and destination parameters. The syntax is defined in Table 2.

Throughout the paper, we assume the function parameters to take values of the *naturals* type $\mathbb{N}$, which is well-founded. Correspondingly, our affine relations are defined over naturals. We interpret an affine relation as a set of pairs of numbers satisfying the affine relation. For example, the following affine relation $\phi = \{[m, n] \to [p] : p = m+n+1\}$ can be interpreted as a subset of $\mathbb{N}^2 \times \mathbb{N}$. Some of the pairs belonging to this set are: $([1, 0], [2])$, $([3, 4], [8])$ and so on.

This interpretation enables us to talk about subset inclusion between set solutions of affine relations. It induces a partial ordering relationship among the affine relations, and corresponds nicely to the implication relation between two affine relations, when viewed as Presburger formulæ. As a result, ϕ implies the relation

$$\phi' = \{[m, n] \to [p] : p > m + n\}$$

because the set generated by ϕ is a subset of that generated by ϕ', denoted by $\phi \subseteq \phi'$. Throughout the paper, we adopt the subset notation as relation implication.

Operations over affine relations: The first operation of interest is the composition operation for affine relations. This operation is meaningful only when we interpret an affine relation as a binary relation over two sets of parameters. The idea of composing two relations, as in $\phi_1 \circ \phi_2$, is to identify the second parameter set of ϕ_1 with the first parameter set of ϕ_2. Formally, composition is defined as follows:

$$\phi_1 \circ \phi_2 \stackrel{def}{=} \{(x, z) \mid \exists y : (x, y) \in \phi_1 \wedge (y, z) \in \phi_2\}$$

Thus, for composition over two affine relations $\phi_1 \circ \phi_2$ to be definable, we must have the number of parameters in the first set of ϕ_2 to be the same as the number of parameters in the second set of ϕ_1.

Fact 1. The composition operator over affine relations is monotone.

The second operation of interest is the union of affine relations. This is definable when all the affine relations have the same set of parameters, modulo variable renaming. Formally, union is defined as follows:

$$\phi_1 \cup \phi_2 \stackrel{def}{=} \{(x, y) \mid (x, y) \in \phi_1 \vee (x, y) \in \phi_2\}$$

Fact 2. The union operation over a set of affine relations is monotone. In fact, the union operation computes the least upper bound of the set of affine relations, if we consider all affine relations with the same set of parameters as a lattice partially ordered by set inclusion.

The third operation of interest is the transitive closure operation over an affine relation. It is defined for an affine relation ϕ as follows:

$$\phi^+ \stackrel{def}{=} \bigcup_{i \geq 0} \phi^i$$

where ϕ^i means composing ϕ with itself i times. Note that for the closure operation to work properly, ϕ must be represented as a relation over two sets of parameters, with both sets of equal size. From facts 1 and 2, we deduce that the transitive closure operation is monotone.

Fact3. The transitive closure operation is idempotent. That is, $\phi^+ \circ \phi^+ = \phi^+$.

3 LJB size-change termination

In LJB-analysis, the size-change graphs approximate the relation between the sizes of source parameters and destination arguments for each call to a function. In the size-change graph, we record only the following information about each destination argument:

- it is the same size ($=$) as some source parameter;
- it is smaller ($\downarrow$) than some source parameter;
- it is either the same size or smaller ($\overline{\downarrow}$) than some source parameter;
- it is larger than some source parameter, or it has no clearly defined relation to the source parameters (**`unknown`**).

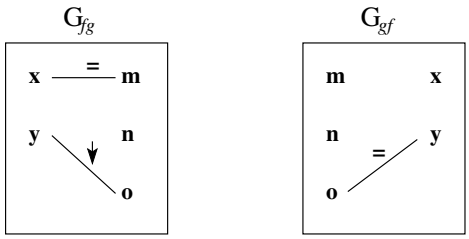
These simple relations are the only ones used to form the size-change graphs in [11,10], and we will refer to this style of size-change graph as an LJB *size-change graph*.

Definition 1. *An LJB size-change graph is a size-change graph, with each destination argument having a simple relation ($=$, $\downarrow$, $\overline{\downarrow}$ or **`unknown`**) to each source parameter.*

We encode an LJB size-change graph by specifying a tuple relation between the source parameters and destination arguments, although the existing literature normally uses a graphical representation. Consider the following functions:

```
f(x, y) = if x ≥ 0 then y else g(x, 0, y − 1);
g(m, n, o) = if o = 0 then m + n + 1 else f(m + 1, o);
```

In Figure 1(a), we are specifying that in function $f(x, y)$, calling function $g(m, n, o)$, the first argument is the same *size* as x ($=$), the second has no relation to the parameters of f (**unknown**), and the third is always smaller than y ($\downarrow$). Similarly for the other function in Figure 1(b). We do not draw the **unknown** relation on the diagram.



(a) f calls g (b) g calls f

Fig. 1. LJB size-change graphs

We now use the two graphs to analyze the mutually recursive functions for termination. A non-terminating sequence beginning with function f would involve an infinite succession of calls to (alternately) the functions f and g. New graphs are constructed, representing the composition of the two original graphs as shown in Figure 2.

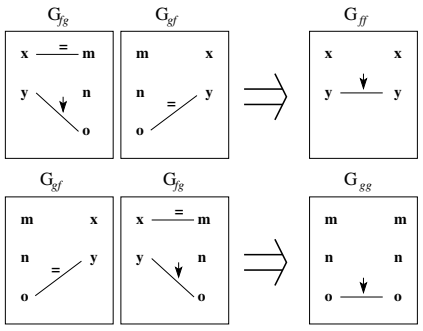


Fig. 2. LJB size-change graphs for functions $f \circ g$ and $g \circ f$

These graphs represent the two successive calls, and demonstrate that the y parameter must reduce. We may express the infinite call sequence as the regular expression $(fg)^n$ or $f(gf)^n$. We know that the sequences $f \circ g$ and $g \circ f$ must occur infinitely often, and are the only infinite call sequences for the program. As a result, since every infinite call sequence in a program would cause an infinite descent in a program value, then the program terminates on all inputs.

The central theorem of the LJB size-change graph construction algorithm, explained and proved in [11], is:

Theorem 1. *Program P is not size-change terminating iff $\mathcal{G}$ contains $g : f \to f$ such that $g = g \circ_G g$ and g has no arc of the form $x \overset{\downarrow}{\to} x$, where $\mathcal{G}$ is the set of possible size-change graphs, and $\circ_G$ represents the composition of two size-change graphs.*

This theorem gives rise to a relatively efficient technique for deriving termination properties from a closure computation over size-change graphs, briefly described in Appendix B.

4 Affine-based size-change graphs

Within the general framework of LJB size-change termination analysis, we now develop concepts for the new affine-based size-change graphs.

We elect to represent size-change information using the notation and syntax of the Omega calculator and library [8], which can manipulate Presburger formulæ. We term the graphs thus represented *affine size-change graphs*.

Definition 2 (Affine graph). *An affine size-change graph is a size-change graph such that each destination argument is constrained by the source parameters arranged in an affine relation.*

Such a graph is sometimes termed an affine tuple relation, mapping n-tuples to m-tuples constrained by an affine formula. In the LJB size-change graph, the possible relations between a source parameter m and a destination argument m' are just simple ones such as $=, \downarrow, \overline{=}\downarrow$ or **`unknown`**. By contrast, in an affine size-change graph, the possible relations between a source parameter m and a destination argument m' can be any expression represented by a Presburger formula. For example, $m' \leq 2m - 4$ is a valid relation in an affine size-change graph.

4.1 Translation of size-change graphs

In this section, we translate the LJB size-change graphs as described in [11] to our affine size-change graph form. We provide a definition of the LJB-graph composition

in terms of operations on affine relations. Notwithstanding the fact that the correctness of the translation is important to the correctness of our latter development, the process of translation also sheds light on the correctness and termination of affine-based termination analysis.

Given two lists of source parameters and destination arguments, we consider a set $\mathcal{G}$ and a set $\mathcal{F}$ of (respectively) all possible LJB-graphs and affine graphs generated from these lists. We define a translation that maps LJB-graphs to affine graphs, as follows:

Definition 3 ($g2a$ **Translation).** *For each edge e_i in $g \in \mathcal{G}$, translation $g2a :: \mathcal{G} \to \mathcal{F}$ produces an affine relation r_i according to the following translation:*

$$\begin{array}{ccc} e_i & & r_i \\ \\ m \xrightarrow{=} n & \mapsto & n = m \\ m \xrightarrow{\downarrow} n & \mapsto & n < m \\ m \xrightarrow{=\!\downarrow} n & \mapsto & n \leq m \end{array}$$

In addition, for each source parameter x_j and destination argument y_k, we associate the constraint $x_j \geq 0$ and $y_k \geq 0$, respectively. Finally, we have

$$g2a(g) = \{[x_1, \ldots, x_m] \to [y_1, \ldots, y_n] : (\bigwedge_i r_i) \wedge \mathcal{D}\}$$

$$\mathcal{D} = (\bigwedge_j x_j \geq 0) \wedge (\bigwedge_k y_k \geq 0)$$

Note that we do not map edges with the **`unknown`** symbol to any constraint, since the symbol implies no knowledge about how the parameter size is changed. We note that $g2a$ is an injection. The relation $\mathcal{D}$ specifies the *boundary constraints*, and asserts that all parameters take non-negative values.

In order to show the correctness of our translation, we define an *abstraction function* from affine graphs to LJB-graphs.

Definition 4 **(Abstraction** $a2g$**).** *The abstraction $a2g : \mathcal{F} \to \mathcal{G}$ is defined as follows:*

$$a2g(a) = g2a^{-1}(\pi(a))$$

where π is defined as follows:

$$\begin{array}{rcl} \pi(a) & = & \{[x_1, \ldots, x_m] \to [y_1, \ldots, y_n] : r \wedge \mathcal{D}\}; \\ r & = & \{\bigwedge(y_j \ op \ x_i) \mid a \subseteq P(x_i, y_j, op),\ 1 \leq i \leq m, \\ & & \qquad 1 \leq j \leq n,\ op \in \{=, <, \leq\}\}; \\ P(u, v, op) & = & \{[x_1, \ldots, x_m] \to [y_1, \ldots, y_n] : v \ op \ u \wedge \mathcal{D}\}; \end{array}$$

π is composed of projection functions that project a graph a onto a rectangular polygon enclosing a. This polygon is made up of affine relations of the following forms: $y\ op\ x$ where y is a destination argument, x is a source parameter, and op is either $=$, $<$, or $\leq$. In addition, we have that all source parameters and destination arguments are constrained to be non-negative. It is easy to show that the function π, and consequently $a2g$, are monotone.

An example of the use of the function π is

$$\pi(\{[x,y] \to [x',y'] : x' = x+2 \wedge x' = y-1 \wedge \mathcal{D}\}) = \{[x,y] \to [x',y'] : x' < y \wedge \mathcal{D}\}$$

where it is seen that the π function retains exactly the information used in LJB-analysis.

In the following, we use $\circ_F$ to represent affine-graph composition.

Property 1. Function π enjoys the following properties:

1. $id_a \subseteq \pi$.
2. It is *idempotent*: $\pi \circ \pi = \pi$.
3. For any $a_1, a_2 \in \mathcal{F}$,
$$\pi(a_1 \circ_F a_2) \subseteq \pi(a_1) \circ_F \pi(a_2).$$

We now provide a definition of LJB-graph composition in terms of affine-graph operations, and give a characterization of the idempotent LJB-graphs using affine graphs.

Lemma 1 (LJB-graph composition $\circ_G$). *For all $g_1,\ g_2 \in \mathcal{G}$,*

$$g_1 \circ_G g_2 \stackrel{def}{=} a2g(g2a(g_1) \circ_F g2a(g_2))$$

Functions $a2g$ and $g2a$ are "tightly" related, in the following sense:

Property 2. Let id_F be the identity function on $\mathcal{F}$, and id_G that of $\mathcal{G}$. The following holds:

$$g2a \circ a2g \subseteq id_F$$
$$a2g \circ g2a = id_G$$

Now, we have the following theorem relating the idempotent LJB-graph to affine graphs:

Theorem 2 (Idempotent Graphs). *Let $g \in \mathcal{G}$ and $a = g2a(g)$.*

$$g \circ_G g \ = \ g \text{ if and only if } \pi(a) = a \text{ and } a = a^+.$$

The proof of this theorem is found in Appendix A.

5 Affine-based termination analysis

The affine-based termination analysis is computed in a similar fashion to the LJB-analysis. The analysis begins with the set of affine graphs representing parameter size-change for each function call in the program. Consequently, arguments' size changes are captured during the analysis for all possible call sequences. This change is computed by composing the existing affine size-change graphs in all the legitimate combinations.

The Omega library [8] can calculate the composition of affine relations efficiently, and assist in the calculation of the closure for a set of such relations. For example, consider the following program:

```
f(m) = if m ≤ 0 then 1 else g(m + 1);
g(n) = if n ≤ 0 then 1 else f(n − 2);
```

The corresponding affine size-change graphs for the call to function g in f (let's label it f_1), and the call to function f in g (label it g_1), are encoded with two affine relations using the Omega library representation as follows:

$$f_1 = \{[m] \rightarrow [m'] : m' = m + 1 \wedge \mathcal{D}\}$$
$$g_1 = \{[n] \rightarrow [n'] : n' = n - 2 \wedge \mathcal{D}\}$$

In each of these representations, the source parameters and destination arguments have one member each, constrained by a relation expressed as a Presburger formula. The first expresses that the destination m' must be one more than the source m. The second expresses that the destination n' must be two less than the source n. From this we can see that we are not only capturing the information about size reduction, but also the size of parameter changes, and perhaps other more subtle relationships. There already exist well-documented ways for extracting affine relations from a program for other purposes. For example, [9] describes a contextual analysis to retrieve such information using sized types.

5.1 Associating affine with abstract graphs

A crucial administrative task in ensuring termination of our analysis is the association of each affine graph with an abstract graph. We could view this as a process of *classifying* our affine size-change graphs. The elements of the abstract size-change graphs provide the different classifications, and the affine graphs provide concrete instances of some of these classifications. An abstract graph is defined as follows:

Definition 5 (Abstract graph). *An affine size-change graph is called an abstract graph if all its parameters are non-negative, and each of its destination parameters y_i can only be related to the source parameters x_j in an affine relation y_i op x_j, where op $\in$ $=, <, >, \leq, \geq$. Moreover, there is no other affine relation among the parameters.*

We see that the set of affine graphs produced from LJB-graphs via the translation $g2a$ is a set of abstract graphs. To obtain a more accurate termination analysis, we need to extend this set of abstract graphs to include those of which the destination parameter can have size greater than some source parameters.

Given a program, we generate all its possible abstract graphs $\mathcal{A}$. Let $\mathcal{F}$ be the corresponding affine graphs that can possibly be created from the program. We then associate each affine graph with an abstract graph in $\mathcal{A}$, as follows:

$$\forall f \in \mathcal{F}, a \in \mathcal{A}, associate(f,a) \stackrel{def}{=} a = \bigcap_i \{r_i \mid f \subseteq r_i,\ r_i \in \mathcal{A}\}$$

where $\phi_1 \cap \phi_2 \stackrel{def}{=} \{(x,y) \mid (x,y) \in \phi_1 \wedge (x,y) \in \phi_2\}$.

We note that the associated abstract graph a thus obtained is minimum, in that any other abstract graph that "contains" the affine graph f will also contain a. Consequently, we call it the *minimum association*.

It is important to point out that, in actual implementation of the algorithm, we maintain not just the association of graphs, but also (approximated) information about call sequences leading to the creation of that particular graph. This call-sequence information is crucial to the accuracy of the termination analysis. However, as the inclusion of call-sequence manipulation will obscure the presentation of our algorithm, we ignore, without loss of generality, call-sequence information in our presentation.

Consider the example given earlier in this section. The minimal set of possible abstract graphs A for the program is $A = \{A_1, A_2, \ldots, A_{10}\}$, where,

For the label f_1:
$A_1 = \{[m] \rightarrow [m'] : m' < m\}$
$A_2 = \{[m] \rightarrow [m'] : m' \leq m\}$
$A_3 = \{[m] \rightarrow [m'] : m' = m\}$
$A_4 = \{[m] \rightarrow [m'] : m' > m\}$
$A_5 = \{[m] \rightarrow [m']\}$

For the label g_1:
$A_6 = \{[n] \rightarrow [n'] : n' < n\}$
$A_7 = \{[n] \rightarrow [n'] : n' \leq n\}$
$A_8 = \{[n] \rightarrow [n'] : n' = n\}$
$A_9 = \{[n] \rightarrow [n'] : n' > n\}$
$A_{10} = \{[n] \rightarrow [n']\}$

In the graphs above, and for the rest of this paper we omit the inclusion of the boundary constraints $\mathcal{D}$. Note that in these tables we only consider the operations $=, <, >, \leq$, excluding the $\geq$ operator. In this particular analysis, the $\geq$ operator does not add any information of interest. The initial set of affine graphs for the functions is $F = \{F_1, F_2\}$ where:

$$F_1 = \{[m] \rightarrow [m'] : m' = m + 1\}$$
$$F_2 = \{[n] \rightarrow [n'] : n' = n - 2\}$$

The elements F_1 and F_2 are concrete instances of the elements A_4 and A_6 of A, as

$$\{[m] \rightarrow [m'] : m' = m + 1\} \subseteq \{[m] \rightarrow [m'] : m' > m\}$$
$$\{[n] \rightarrow [n'] : n' = n - 2\} \subseteq \{[n] \rightarrow [n'] : n' < n\}$$

and we associate each affine graph with its classification using the set of pairs

$$\mathcal{C} = \{(F_1, A_4), (F_2, A_6)\}$$

The process of *classifying* our affine size-change graphs using the abstract size-change graphs, leads to a finite number of affine size-change graphs. We use this property in the proof of termination of the termination analysis algorithm.

5.2 Affine-based closure algorithm

The core of the termination analysis is the algorithm $\mathcal{T}$, which builds the closure of the new set of affine size-change graphs F uses a simple technique, constructing the compositions of the existing affine size-change graphs until no new affine graphs are created. The algorithm $\mathcal{T}$:

```
Classify initial affine graphs into C';
C := ∅;
while C' ≠ C do {
    C := C';
    F' := generate(C);
    foreach g ∈ F' {
        (r, A_g) := classify(g, C');
        if idempotent(g, A_g) then
            g := g+;
        g := ω(hull(r ∪ g));
        if nullgraph(r) then
            C' := C' ∪ (g, A_g)
        else
            C' := (C'\(r, A_g)) ∪ (g, A_g);
    }
}
```

The main idea of the algorithm, ignoring the termination issues, can be described using the following metaphor: recall that each affine graph can be associated with a minimum abstract graph. Imagine each abstract graph (there are a finite number of them) as a container, which will contain those affine graphs under its minimal association. The containers will have a cap labeled with the corresponding abstract graph. Thus, some containers will be considered as idempotent containers when they are labeled/capped with an idempotent abstract graph.

Initially, only the initial set of affine graphs will be kept in some of the containers. At each iteration of the algorithm, a composition operation will be performed among all legitimate pairs of affine graphs, including self-composition. The resulting set of affine graphs will again be placed in the respective containers they are (minimally) associated with. Assume that such an iteration process will eventually terminate, with no more new affine graphs created. Then we need to identify those non-empty idempotent containers, checking their cap to see if the idempotent abstract graphs labeled therein contain a decreasing edge. If all these idempotent abstract graphs have decreasing edges, we

conclude that the associated program terminates. If one of these graphs does not have a decreasing edge, we shall conclude that the associated program does not belong to the size-change terminating programs.

Suppose we associate each affine size-change graph with its classification, using the set of pairs

$$\mathcal{C} = \{(F_1, A_j), (F_2, A_k), \ldots\}$$

This set grows in size as the algorithm runs, but has a maximum size determined by the size of $\mathcal{A}$. The function **`classify`** returns a pair (r, A_k), with r a null graph if this classification has not been made before, or with the previous value for the affine size-change graph if this classification has been made before. The function **`generate`** returns a new set of affine size-change graphs constructed by composing any possible pairs of existing size-change graphs. The legitimate compositions only include those which result in a legitimate call sequence, as is the case for LJB-analysis. In the event that a graph g is idempotent, we keep the transitive closure g^+ of the graph, reflecting the idea that any idempotent graph may result in an infinite series of calls through itself. The function ω performs a widening operation to produce an affine graph that is larger than the argument graph. This is a common technique used in ensuring finite generation of abstract values [6]. The function **`idempotent`** checks if a graph g and its self composition $g \circ g$ are both minimally associated with the same abstract graph A_g.

In contrast with the metaphor described above, the algorithm maintains at most *one* affine graph in each container (classification). When a new graph is found to belong to an existing non-empty container, it is combined with the existing graph in the container, using the union, hull, and widening operations. Finally, the algorithm's main structure $\mathcal{C}$ is continually updated into $\mathcal{C}'$ until it reaches a fixed point.

A widening operator: In order to guarantee termination for our algorithm, we propose a widening operation, which retains as much information as possible from the container of affine graphs, and guarantees the stabilization of the affine graph in that container under graph composition. The idea of using a widening operator to control termination is not new, and can be found in (for example) [6,12].

To better understand how our widening function ω is defined, we first note that the affine constraints in an affine graph can be divided into three sets:

1. The boundary constraints, $\mathcal{D}$.
2. The contextual constraints, $\mathcal{K}$: for the context in which the call represented by the affine graph is called. There are no destination arguments in these constraints.
3. The output constraints $\mathcal{E}$: this constrains the destination arguments by other variables, such as source parameters.

Furthermore, an output constraint over destination argument y_j can be expressed as follows:

$$y_j \; op \; \Sigma_k a_k x_k - \Sigma_l b_l x'_l + c$$

where $op \in \{=, <, >, \leq, \geq\}$, x_k, x'_l are source parameters, $a_k, b_l \geq 0$ and c is any integer.

Definition 6. *Given that the affine relations in an affine graph a can be divided into three sets of constraints, $\mathcal{D}$, $\mathcal{K}$, and $\mathcal{E}$, as described above. The widening operator ω applying over a may now be defined as follows:*

$$\omega(a) = \{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : (\bigwedge_i E(r_i)) \wedge \mathcal{D} \wedge (\bigwedge_j K(r'_j)) | r_i \in \mathcal{E}, r'_j \in \mathcal{K}\}$$

where $E(r_i)$ is a function defined by the following table (assume $c > 0$):

Form of r_i	**Conditions**	$E(r_i)$
$y\ op\ x + c$	$op \in \{=, >, \geq\}$	$y\ op\ x + c$
$y = \Sigma_k a_k x_k + c$	$a_k > 0, k > 1$	$\bigwedge_k y \geq a_k x_k + c$
$y > \Sigma_k a_k x_k + c$	$a_k > 0, k > 1$	$\bigwedge_k y > a_k x_k + c$
$y \geq \Sigma_k a_k x_k + c$	$a_k > 0, k > 1$	$\bigwedge_k y \geq a_k x_k + c$
$y\ op\ x - c$	$op \in \{=, <, \leq\}$	$y\ op\ x - c$
$y = x - (\Sigma_l b_l x_l) - c$	$b_l > 0, l > 0$	$y \leq x - c$
$y < x - (\Sigma_l b_l x_l) - c$	$b_l > 0, l > 0$	$y < x - c$
$y \leq x - (\Sigma_l b_l x_l) - c$	$b_l > 0, l > 0$	$y \leq x - c$
all other forms		true

and $K(r'_j)$ is a function defined by the following table (assuming $c > 0$, and m and n are arbitrary variables):

Form of r'_j	**Conditions**	$K(r'_j)$
$m\ op\ n$	$op \in \{=, <, >, \leq, \geq\}$	$m\ op\ n$
$m\ op\ c$	$op \in \{=, >, \geq\}$	$m\ op\ c$
all other forms		true

Imagine our mn-tuple relations in $m + n$-space. For example if our source tuple had two parameters (x, y) and our destination arguments just one (z) then we might imagine a 3-space, with the axes x,y and z. The widening operation projects an arbitrary affine relation onto each of the two axes x and y. A relation such as $z = x + y + 4$ would be widened to

$$z \geq x + 4 \wedge z \geq y + 4$$

Property 3. The widening operator ω defined in Definition 6 is monotone and idempotent. Furthermore, for any $a \in \mathcal{F}, \omega(a) \subseteq \pi(a)$.

In addition to being monotone, ω also ensures the generation of a finite number of affine graphs which are less precise than the initial one. For instance, if an application of ω produces an affine graph having the following constraint:

$$y \geq 5x_1 + 2x_2 + 3$$

then, there are only a finite number of constraints which are of identical form

$$y \geq \Sigma_k a_k x_k + c$$

and are less precise, namely:

$$y \geq d_1 x_1 + d_2 x_2 + d_3$$

where $0 < d_1 < 5$, $0 < d_2 < 2$ and $0 < d_3 < 3$. This finiteness in the number of less precise constraints guarantees that iterative computation of affine graphs will stabilize in a finite number of steps.

Definition 7. *An affine graph of the following form is said to be in stable form:*

$$\{[x_1, \ldots, x_m] \rightarrow [y_1, \ldots, y_n] : Er \wedge D \wedge Kr\}$$

where Er and Kr are conjunctions of constraints of the forms described by the range of functions E and K respectively.

Property 4. Given an affine graph g defined in a stable form, there are a finite number of affine graphs of identical form which are less precise than g.

5.3 Properties of affine-based termination analysis

It is easy to verify that the analysis, if it terminates, computes more accurate information than the LJB-analysis. To see that, we drop all the contextual information in the initial affine graphs collected, by making all such contextual information true. We then replace the widening operator ω in our analysis by π defined in Definition 4. This turns those affine graphs, which are associated with idempotent abstract graphs, into their respective abstract graphs during the analysis. With these changes, we obtain an analysis that mimics the computation of LJB-analysis.

Termination of the algorithm: The key components of the algorithm ensure that the algorithm terminates. In the following proof, we use the finite cardinality of the size-change graphs, and the monotonic nature of the graph operations.

Theorem 3. *The algorithm $\mathcal{T}$ terminates.*

Proof. The algorithm terminates when $\mathcal{C}'$ is no longer changing. $\mathcal{C}'$ can change in only two ways, either by

1. *creating* an association for a new abstract graph A_g, or by
2. *replacing* an existing set element (r, A_g) with a new (g', A_g).

Proof of termination is done here by showing that neither of the above actions can be done an infinite number of times.

Addressing the first point, the cardinality of the set $\mathcal{C}'$ is finite, bounded by the cardinality of the set of abstract graphs $\mathcal{A}$. As a result of this, $\mathcal{C}'$ cannot continue increasing in size forever, and hence we cannot keep adding in a new association (g', A_g).

Addressing the second point, consider the replacement of an existing set element (r, A_g) with a new set element (g', A_g). From the algorithm, we know that

$$g' = \omega(\mathbf{hull}(r \cup g))$$

Thus g' is in stable form. Since any update of g' (in further iteration of the algorithm) will result in an affine graph, g'', which cannot be more precise than g' (by Property 3), and the application of ω ensures that g'' is also in stable form, by Property 4, there exists a finite number of iterations in which the update of the affine graph will stabilize. This completes the termination proof.

6 Example using affine size-change graphs

In this example we demonstrate constant change cancellation. In an earlier section, we introduced the following two affine size-change graphs:

$$F_1 = \{[m] \to [m'] : m' = m + 1\}$$
$$F_2 = \{[n] \to [n'] : n' = n - 2\}$$

Our initial value for $\mathcal{C}$ is

$$c_1 = \{(F_1, A_4), (F_2, A_6)\}$$

The first call to **`generate`** returns the following new affine size-change graphs:

$$F_3 = \{[n] \to [n'] : n' = n - 1\} \qquad \text{(from the sequence } g_1 f_1\text{)}$$
$$F_4 = \{[m] \to [m'] : m' = m - 1\} \qquad \text{(from the sequence } f_1 g_1\text{)}$$

Since these are idempotent, we calculate the closure of F_3 and F_4:

$$F_3 = \{[n] \to [n'] : n' < n\}$$
$$F_4 = \{[m] \to [m'] : m' < m\}$$

After this first iteration, $\mathcal{C}$ has the value

$$c_2 = \{(F_1, A_4), (F_2, A_6), (F_3, A_{11}), (F_4, A_{12})\}$$

where

$$A_{11} = \{[n] \to [n'] : n' < n\}$$
$$A_{12} = \{[m] \to [m'] : m' < m\}$$

After a few more iterations we get one more graph, and $\mathcal{C}$ has a stable value. The new graph is:

$$F_5 = \{[m] \to [n'] : n' \leq m\} \qquad \text{(from the sequence } f_1 g_1 f_1\text{)}$$

The idempotent functions are F_3 and F_4, which each have a reducing parameter, and so we conclude that the size-change termination property applies to this function. There is another example in Appendix C.

In general, using this technique we can capture termination for all the functions captured by the LJB technique, and also some others. In other words we can capture termination for a larger set of programs.

7 Related work and conclusion

This paper presented an approach to improve the analysis of program termination properties based on the size-change termination method. We encoded size-change graphs using Presburger formulæ representing affine relations, and explored more refined size-change graphs admissible by these affine relations. The algorithm for calculating the closure of the affine size-change graphs has been shown to terminate. Consequently, our affine-related analysis improves the effectiveness of the LJB termination analysis by capturing constant changes in parameter sizes, and affine relationships of the sizes of the source parameters. Our approach widens the set of functions that are size-change terminating.

The way in which we attempt to find closures is different from the normal approach of finding closures and fixed points. Conventionally, program analysis will attempt to find a closure for each function definition, such as those found in [2] and in the work on termination done for the Mercury system [14], and many works on program analysis, for example [7,1,6].

In our approach, we express the ultimate closure in terms of several closures called the idempotent graphs. This is similar to the idea of polyvariant program analysis [15,4], but differs in that there are also some graphs around during the analysis which cannot be made idempotent, yet are important for closure building. There are two ways to obtain polyvariant information for termination analysis: one way is to make use of the constraint enabling the call. Such constraints are commonly obtained from the conditional tests of the code leading to the call. We have provided an account of using this information in our analysis. The use of Presburger formulæ enables us to easily include such information in the analysis, resulting in a context-sensitive analysis [12,13].

Another way to capture polyvariant information is to capture the possible function call sequence, such as a function g can be called from another function f, but not from k. The LJB-analysis uses this information to achieve polyvariant analysis. While this information can be captured in our algorithm (by creating more distinct abstract graphs with call sequence information), we do not present it, due to lack of space. Nevertheless, it will be interesting to look into integrating call-sequence information into constraints, so that there is only one source for controlling polyvariance.

As it is, the termination analysis deals with a set of mutually recursive functions at a time. It would be interesting to investigate the modularity of the analysis, so that each function can be analyzed separately, and the results from various functions be composed to yield the final termination result. One such opportunity is to integrate the affine-based termination analysis with a type-based system. Currently, we have begun working towards a type-inference system which collects the relevant size-change information for performing this sort of termination analysis.

The use of constraints in expressing argument change enables us to consider termination beyond the *well-founded* method. For example, through constraints, we can express the fact that an increasing argument can be bounded above. This idea has been explored in the work on sized typing in [2]. The bounded-increment of an argument is also investigated in [3], which computes a linear ranking function for a loop-based program. In contrast, our technique deals directly with recursive function calls instead of loop-based programs. We plan to enhance the existing technique to include such techniques.

Acknowledgements.

The authors would like to thank Lindsay Groves and the anonymous referees for their many insightful and helpful comments. This work has been supported by the research grant R-252-000-138-112.

References

1. A. Aiken. Introduction to Set Constraint-Based Program Analysis. *Science of Computer Programming*, 35(1999):79–111, 1999.
2. W.N. Chin and S.C. Khoo. Calculating Sized Types. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 62–72, 2000.
3. M. Colón and H. Sipma. Practical Methods for Proving Program Termination. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 442–454. Springer, 2002.
4. C. Consel. Polyvariant Binding-Time Analysis For Applicative Languages. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 66–77, 1993.
5. T. Coquand and C. Paulin. Inductively Defined Types. In P. Martin-Lof and G. Mints, editors, *Proceedings of COLOG'88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. ACM, Springer, 1990.
6. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.
7. N. Heintze. *Set Based Program Analysis*. PhD thesis, CMU, 1992.
8. P. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Version 1.1.0 Interface Guide. Technical report, University of Maryland, College Park, November 1996.
9. S.C. Khoo and K. Shi. Output Constraint Specialization. *ACM SIGPLAN ASIA Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 106–116, September 2002.

10. C.S. Lee. Program Termination Analysis in Polynomial Time. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 218–235. ACM, Springer, October 2002.
11. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 28, pages 81–92. ACM press, January 2001.
12. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
13. O. Shivers. Control Flow Analysis in Scheme. *ACM SIGPLAN Notices*, 7(1):164–174, 1988.
14. C. Speirs, Z. Somogyi, and H. Sondergaard. Termination Analysis for Mercury. In *Static Analysis Symposium*, pages 160–171, 1997.
15. W. Vanhoof and M. Bruynooghe. Binding-time Analysis for Mercury. In *International Conference on Logic Programming*, pages 500–514, 1999.

A Proof of Theorem 2

($\Rightarrow$) Suppose $g \circ_G g = g$, we show that $\pi(a) = a$ and $a = a^+$, where $a = g2a(g)$. To show that $\pi(a) = a$, we note that a comprises constraint of the form $y_j \ op \ x_i$, which is identical to $P(x_i, y_j, op)$, the constituents of the result of applying π to any graph. Consequently, apply π on a will not modify any of a's constraints. Thus, $\pi(a) = a$.

To show that $a = a^+$, we show that $a = \bigcup_{i>0} a^i$ using subset inclusion relation. Since $\bigcup_{i>0} a^i = a \cup \bigcup_{i>0} a^{i+1}$, thus $a \subseteq \bigcup_{i>0} a^i$. To show the other way round, we make use of the assumption that $g \circ_G g = g$, this gives us $\pi(a \circ_F a) = a$.

$$
\begin{array}{l}
a2g\ (g2a(g) \circ_F g2s(g)) \ = \ g \\
\Leftrightarrow \text{[Definition of } g2a\text{]} \\
\quad g2a^{-1}(\pi(a \circ_F a)) \ = \ g2a^{-1}(\pi(a)) \\
\Leftrightarrow [\pi(a) = a] \\
\quad g2a^{-1}(\pi(a \circ_F a)) \ = \ g2a^{-1}(a) \\
\Leftrightarrow [g2a^{-1} \text{ injective over range of } \pi] \\
\quad \pi(a \circ_F a) \ = \ a
\end{array}
$$

Thus, we show by induction that $a^i \subseteq a$ for all $i > 0$.

Case $i = 1$, we have $a \subseteq a$.

Inductive case: Assume that $a^n \subseteq a$,

$$
\begin{array}{lll}
a^{n+1} & = & a^n \circ_F a \\
& \subseteq & \text{[property 1.1]} \\
& & \pi(a^n \circ_F a) \\
& \subseteq & \text{[property 1.3]} \\
& & \pi(a^n) \circ_F \pi(a) \\
& \subseteq & \text{[Induction hypothesis]} \\
& & a \circ_F \pi(a) \\
& = & a \circ_F a \\
& \subseteq & \text{[property 1.1]} \\
& & \pi(a \circ_F a) \\
& = & \text{[assumption]} \\
& & a.
\end{array}
$$

Hence a is a upper bound of ${a^i}_{i>0}$, and therefore $\bigcup_{i>0} a^i \subseteq a$.

Thus, $a = a^+$.

($\Leftarrow$) Given that $a = g2a(g)$. Assume that $\pi(a) = a$, and $a = a^+$, we want to show that $g \circ_G g = g$. Equivalently, to show that $\pi(a \circ_F a) = a$.

Since $a = a^+$, by the property of transitive closure, we have $a \circ_F a = a$. So, we need to show that $\pi(a) = a$, which is what we have assumed.

B Graph construction

There are only a finite number of different compositions of graphs. We need to calculate the closure of these size-change graphs to ensure that we do not miss analyzing the behaviour of some sequence of function calls, and this closure may be computed in a conventional fashion by composing any possible pairs of graphs, and checking if the composite graph is a new one, or just a repetition of an existing one. We keep composing pairs of graphs until no new graphs are created.

We then look at the *idempotent* graphs $G_i \in \mathcal{G}^+ : G_i = G_i \circ_G G_i$. We can view these as the graphs that represent a recursive chain of calls through a particular function entry point. For each of these idempotent graphs, we examine the size-change information. If all of them have at least one parameter that reduces, then we have our conflict, and can conclude that the program terminates on all inputs.

The technique is interesting in many ways, but not least because the termination argument may be done automatically, without appealing to higher-level reasoning techniques. For example, it is common to use a lexicographic ordering to demonstrate termination of a function, and such a function may require some inspection to determine a suitable ordering of the parameters. The following recursive countdown function c can be inspected, and termination confirmed by noting that the lexicographic ordering **h:t:o** (hundreds:tens:ones) always reduces on every call[1].

```
c(h, t, o) = if h + t + o = 0 then 0
             else if o = 0 then
                     if t = 0 then
                        c(h − 1, 9, 9)
                     else
                        c(h, t − 1, 9)
                  else
                     c(h, t, o − 1);
```

However the LJB size-change termination method is independent of this order, and is able to successfully confirm termination for this example without guidance. The closure of the size-change graphs is shown in Figure 3, and each graph is idempotent, and has a reducing parameter. As a result we conclude that the function terminates on all inputs.

[1] This function is chosen deliberately to highlight the suitable lexicographic ordering.

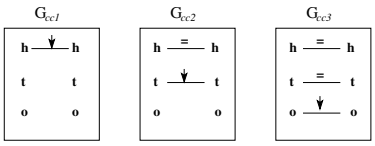


Fig. 3. LJB size-change graphs for function c

C An example with context analysis

In this paper, we focus on the use of size-change analysis to yield termination properties of a program. Chin and Khoo [2] have explored the use of the Omega calculator to infer other properties of arbitrary functions, and it seems appropriate to extend the termination analysis using this sort of inference. Consider the following code segment:

```
j(m, n) = if m = 0 ∨ n = 0 then 0
          else if m < n then
                  j(m − 1, n + 1)
               else
                  j(m + 1, n − 1);
```

It is easy to convince yourself that this function terminates, as only the first or the second recursive call to j will be made. However, the LJB size-change termination method cannot be applied, and similarly for the improvement to the LJB method just explored. The closure of the size-change graphs will be:

$$F_1 = \{[m,n] \to [m',n'] : m' = m - 1 \wedge n' = n + 1\}$$
$$F_2 = \{[m,n] \to [m',n'] : m' = m + 1 \wedge n' = n - 1\}$$
$$F_3 = \{[m,n] \to [m,n]\}$$

We cannot infer termination from this, as the composition of the two calls has no reducing parameters. However, we can include extra information that we can capture from the code segment. For example, the first call is only performed when $m < n$. The second call is only performed when $m \geq n$. By including this information in the constraining affine relation, the new calculation of the closure is as follows:

$$F_1 = \{[m,n] \to [m',n'] : m' = m - 1 \wedge n' = n + 1 \wedge m < n\}$$
$$F_2 = \{[m,n] \to [m',n'] : m' = m + 1 \wedge n' = n - 1 \wedge m \geq n\}$$
$$F_3 = \{[m,n] \to [m',n'] : \text{False}\}$$

The result of the calculation of the composition of the functions indicates that this situation cannot occur (i.e. it is not possible for the two calls to occur). As a result the closure is just F_1 and F_2, and each of these has a reducing parameter. Again, we have shown that by collecting and keeping more information, we can deduce termination for at least this function.