THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRB6/05

Bounded Size-change Termination

Siau-Cheng KHOO and Hugh ANDERSON

June 2005

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

Bounded Size-change Termination

Siau-Cheng Khoo and Hugh Anderson

Department of Computer Science
School of Computing
National University of Singapore
{khoosc,hugh}@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. In this paper, we extend size-change termination beyond the original well-founded condition to include arbitrary bounds that are expressed by linear constraints. Our bounded termination condition determines if repeated calls to a function will monotonically move the call arguments toward the boundary of the guard. We also present a technique which allows the analysis of functions in which the return values are relevant to termination. Our analysis exploits the decidability and expressive power of affine constraints. These techniques significantly extend the set of programs that are size-change terminating.

1 Introduction

There are many approaches to termination analysis. One 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"* [13]. 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, and the only technique for repetition is recursion.

We begin with an informal outline of the size-change termination method. Firstly consider 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. Secondly, we consider all possible *infinite* call sequences. If in the infinite call sequences, a parameter reduces infinitely often, then the data is not well-founded. As a result of this, by contradiction we can assert termination.

In [13], Lee, Jones, and Ben-Amram present a practical technique for deriving program termination properties from size-change information (hereafter called LJB-analysis), by constructing a set of size-change graphs (or just *graphs*) for the program. These graphs approximate the relation between the sizes of source parameters and destination arguments for each call to a function. The graphs are then composed to reveal all sequences of function calls that could be idempotent.
In [1], the authors extended LJB-analysis by a new encoding of the LJB size-change graphs. This small extension increased the set of programs that are size-change terminating by representing the graphs as a set of affine tuple relations, and by including

contextual information to constrain the tuple relations. This had two effects, the first one being that the more refined encoding allowed arithmetic calculations on the size-change information and could record the size of the change, which was useful for some types of functions. The second effect was that the contextual information could guide the composition of size-change graphs.

In this paper, the authors describe significant extensions and variations to LJB-analysis which exploit the capability of affine constraints and produce a more precise set of composite graphs. The resulting analysis can be applied to guarded functions which terminate through exceeding bounds, and in addition handle functions in which return values are relevant to termination. This is done by constructing and evaluating more refined graphs, and handling call sequences that repeatedly change a bounded value. These changes extend the affine-graph based size-change termination analysis conservatively: any program that previously could be shown to be size-change terminating can still be analysed.

As a vehicle for some aspects of the presentation we use McCarthy's 91 function [15], which has been extensively studied, with many proofs of both its behaviour and its termination properties. It is commonly used as an example of a function that does not respond easily to *automatic* proof methods because the returned value of the function is relevant to its termination. The following annotated code shows the 91 function:

$f91(n)$ **= if** $n > 100$ **then**
$n - 10_a$
else
$f91(f91(n + 11)_b)_c$;

The function has the property that for all $n \leq 101$, the returned value of the function is 91 (and hence the name). Analysis of this function using LJB-analysis is not possible, as the graph for function call c has no information relating the size of the parameter n and the call argument. The enhanced termination-detection ability is made possible by extending the affine graphs to include a result value to capture the return of function calls at specific affine-graphs, and a sufficient condition for detecting that the change of a call argument is bounded.

In Section 2 we begin with the language and notations used, and an introduction to affine-based size-change graphs and termination analysis. In Section 3 we introduce a new class of graphs, and demonstrate how a more refined sequence of allowable function calls may be constructed by guiding the graph composition process. In Section 4 we formalize an intuitive notion of size-change termination, that it should work for function sequences that monotonically change a value that is bounded. In addition, the affine graph construction and termination algorithm is described. In Section 5 we show a worked example using both the techniques. Finally, in Section 6 we outline the relation of this work to others, and conclude with some observations about the direction of our research.

2 Preliminaries

In order to concentrate on the mechanism behind our analysis, we choose to work on the simple first-order functional language defined in Appendix A. Affine relations are

captured using Presburger formulæ, with explicitly identified source and destination parameters, also defined in Appendix A.

Throughout this paper, we assume the affine relations to be defined over either integers or naturals. We interpret an affine relation as a set of pairs of numbers satisfying the relation. For example, $\phi = \{[m,n] \rightarrow [p] : p = m + n + 1\}$ can be interpreted as a set of pairs of the form $([m,n],[p])$. Some pairs in 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 $\phi' = \{[m,n] \rightarrow [p] : p > m + n\}$ because the set generated by ϕ is a subset of that generated by ϕ', denoted by $\phi \subseteq \phi'$.

2.1 LJB size-change graphs

In LJB-analysis, the graphs approximate the relation between the sizes of source parameters and destination arguments for each call to a function. In particular, we record if it is the same size ($=$), smaller ($\downarrow$), or the same or smaller ($\overline{=}\!\!\downarrow$) than some source parameter. Otherwise we record if it is unknown or not clearly defined (**unknown**). We refer to this style of size-change graph as an LJB *size-change graph*. Consider the following mutually recursive functions:

$f(x,y)$ **= if** $x \geq 0$ **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 the left-hand graph of Figure 1, 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 ($\downarrow$). We do not draw the **unknown** relation on the diagram.

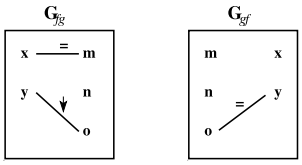

Fig. 1. LJB size-change graphs

These graphs can then be composed to construct new graphs that represent the effect on parameters of function calls.

2.2 Affine-based analysis

In [1], we encoded LJB size-change graphs by specifying a tuple relation between the source parameters and destination arguments. For example, the corresponding *affine size-change graphs* (or *affine graphs* for short) for the above calls are encoded with two affine relations using the following representation:

$$G_{fg} = \{[x, y] \rightarrow [m, n, o] : m = x \wedge o < y \wedge \mathcal{D}\}$$
$$G_{gf} = \{[m, n, o] \rightarrow [x, y] : y = o \wedge \mathcal{D}\}$$

In each of these relations, the source parameters and destination arguments are constrained by a relation expressed as a Presburger formula. The first expresses that the destination m is the same as the source x and that destination o is less than the source y. The second expresses that the destination y is the same as the source o. The relation $\mathcal{D}$ represents other contextual constraints, described in detail in the paper. By default, we omit the contextual constraints to clarify the presentation. From this we can see that we are capturing basic information about size reduction and also, since the expressions can include Presburger formulæ, 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, [12] describes a contextual analysis to retrieve such information using sized types.

Affine relations can be composed, as in $\phi_1 \circ \phi_2$, to identify, and internalize, the second parameter set of ϕ_1 with the first parameter set of ϕ_2. Formally, composition is a monotone operation defined as follows:

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

Two affine relations can be combined to form a "bigger" relation using the *union* operation. This is definable when all the affine relations have the same set of parameters, modulo variable renaming. Union is monotonic and is defined as:

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

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 (modulo renaming) as a lattice partially ordered by set inclusion.

Lastly, we can define the *transitive closure* operation over an affine relation. It is defined for an affine relation ϕ as $\phi^+ = \bigcup_{i>0} \phi^i$, 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. The transitive closure operation is monotone, but more importantly, transitive closure produces an affine relation that is *idempotent*; *ie.*, $\phi^+ \circ \phi^+ = \phi^+$.

In summary, affine tuple relations provide a more refined representation of parameter size-change relations. In this paper we switch between the graphical and affine tuple relation notations freely, whenever it is appropriate.

3 Extended size-change graphs

In order to capture returned values of function calls, we extend size-change graphs to the form shown in Figure 2. It is divided into two parts: the upper part, called the *base*,

is the original graph, while the lower part is the *extension*. Affine relations may also be specified over the extension.

3.1 Value size-change graphs

The first extension adds in a global *result* value for function calls. This is for call sequences that use a call return value. To reflect the effect of a function result value on a following call, we represent the return of function call by an affine graph, called a *value* graph. The original size-change graphs are *call* graphs.

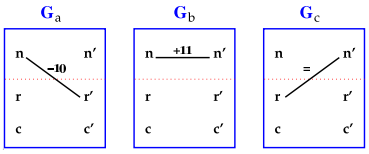

Fig. 2. Extended *value* and *call* graphs for the 91 function

As an example, the value graph, G_a, for the 91 function is shown in Figure 2, signifying the return of value $n-10$ from a call. Accompanying G_a are two *call* graphs available in the 91 function: G_b and G_c signifying two recursive calls in the function definition. Note that the graph G_c states a transfer of value from the result source (r) to the parameter destination (n). Their corresponding affine relations are expressed as follows:

$$a = \{[n, r, c] \rightarrow [n', r', c'] : r' = n - 10\}$$
$$b = \{[n, r, c] \rightarrow [n', r', c'] : n' = n + 11\}$$
$$c = \{[n, r, c] \rightarrow [n', r', c'] : n' = r\}$$

The effect of a nested function call using the return value of previous call can be described by composing a value graph with a call graph, as in the case of the 91 function call shown in Figure 3.

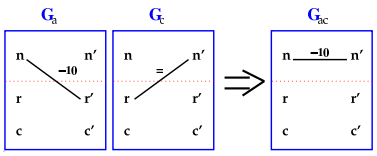

Fig. 3. Composition of nested call

Note the composed graph shows a decrease in the 91's only argument, n. This reflects the transfer of a value to a call c from some *last* recursive call. The corresponding tuple

relation is expressed as follows:

$$\{[n, r, c] \rightarrow [n', r', c'] : n' = n - 10\}$$

It is worth noticing that we only capture *some*, and not all, of the returns to functions in value graphs. Specifically, we only capture the return of the last call in a series of nested calls involving the same label. Semantically, returns from a series of nested calls transfer the result from callees back to the callers, and we elect to ignore this information in order to maintain clarity in our exposition. This results in a simpler classification of call sequences.

It is possible to simply drop a value graph into the pool of existing call graphs, and apply LJB-analysis on this graph pool to attempt to compute the termination of the 91 function, but the attempt will be unsuccessful. We therefore refine the analysis to take advantage of other information available from static analysis of the functions: call sequences and relative numbers of calls.

3.2 CFGs and Counters

We wish to limit the possible ways of composing a value graph with a call graph, so as to exclude the generation of graphs representing illegal call sequences. We provide an informal account of our approach in order to simplify our presentation.

In the 91's case, it is obvious that a function return will *not* be followed by a call at label b. Such composition should therefore be ruled out. To facilitate this extraction of legal call sequences from program text, we can begin by turning any recursive function into a *flattened* recursive procedure using global variables for the returned values for the functions. A more refined analysis result can be obtained by using more than one global variable for each possible returned value. The flattened 91 function, using a global integer **`result`** is:

```
f91(n) = if n > 100 then
           result := n − 10_a
         else {
           f91(n + 11)_b;
           f91(result)_c
         };
```

The flattened syntax reveals the CFG representing a *trace* of the sequence of labelled calls and returns to the function:

$$\mathcal{P} \rightarrow a \mid b\mathcal{P}c\mathcal{P}$$

where b and c represent the calls to the functions at location b and c, and a represents the function return. As the representation is context-free, this CFG defines a set of traces that is larger than the actual set of allowable traces, but is still fairly refined, and certainly can guide the composition of the initial size-change graphs to result in a smaller set of allowable traces. Any string recognized by the CFG forms a *complete* call trace. To be able to detect non-termination, we need to also consider *incomplete* call traces. Nevertheless, the CFG formation of call traces reveals two useful considerations of graph composition.

Fact 1. It is impossible to compose a value graph (represented by a in the CFG above) with the first call graph (represented by b) in any recursive branch of the CFG. Furthermore, every call graph excluding the first one in a recursive branch (such as c) must be preceded by a value graph.

This fact indicates that during the composition of graphs we need not consider certain composed graphs such as $G_{ab} \stackrel{def}{=} (G_a \circ_G G_b)$, avoiding many illegal graph compositions. In the 91 function, we are only interested in the composition of G_{ac} and G_b, and call these the *initial* graphs. The traces generated by this set of initial graphs through graph composition can be represented by$\mathcal{P}_2 \rightarrow b\mathcal{P}_2 \mid ac\mathcal{P}_2$. By contrast, LJB graph composition would consider the language generated by $\mathcal{P}_3 \rightarrow b\mathcal{P}_3 \mid c\mathcal{P}_3$.

It is useful to construct more accurate traces. If $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ represent progressively more abstract languages, and $\mathcal{L}_1 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_3$, then if we can prove a (termination) property for $\mathcal{L}_2$ or $\mathcal{L}_3$, then the property will hold for $\mathcal{L}_1$. Analysis of a more refined trace can be used to derive termination properties for a larger set of programs.

This new set of initial graphs can be mechanically derived from the CFG for a function by constructing a new call graph (or new call graphs) for each terminal to the right of an embedded non-terminal. For example, if we had two non-terminals Q and P in the grammar $Q = a \mid bQcP$ and $P = d \mid ePfQ$, then we would consider the call graphs generated by $Q = bQ \mid acP$ and $P = eP \mid dfQ$. These graphs encapsulate information about the returned values of calls, and also still provide a more precise set of graph compositions.

Fact 2. If a terminal x precedes another terminal y in a recursive choice branch of a CFG, then in any incomplete call trace forming the prefix of a string recognizable by the CFG, x occurs at least as many times as y in the trace.

This fact indicates that we cannot *leave* call nestings more often than we *enter*. We observe that an infinite call trace for the 91 function (represented by recursively unfolding a non-terminal in its original CFG) must have infinite occurrences of the first call. To support this observation, we add a set of new labels, called *counters* (denoted by $c_1 \ldots c_n$) in the extended part of each graph.

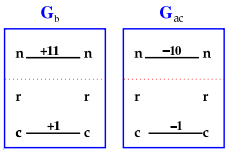

Fig. 4. Counters for Extended graphs of the 91 Function

The extended size-change graph in Figure 4 has a counter for the call b and ac. Every call to b involves an increment in nesting level. On the other hand, every call to ac involves a decrement in the nesting level. A composed graph is considered *legal* if it is composed from other legal graphs and its counter remains positive. Such legal graphs mimic a (set of) legal call trace(s).

Counters can be mechanically included in a set of the initial graphs. For each recursive choice branch in the CFG for a function, we allocate a counter for each embedded non-terminal. This counter is used to limit the relative number of calls made by the (terminal) call graphs on either side of the non-terminal. Each occurrence of left-hand call increases the counter by 1, and each occurrence of right-hand call decreases the counter by 1. The generation of any new affine graph g is limited by presetting the counter c_g value to 0 before constructing the composition, and by placing a contextual constraint $c_g > 0$ on the graph for the right-hand call. For example, if we had $aPbQcR$, we would construct two new counters c_P and c_Q. The affine call graph for graph a would increment c_P. The call graph for graph b would include $c_P > 0$ in the contextual constraint for the graph, and decrement c_P. Similarly for graphs b and c.

The machinery of counters is a little cumbersome, and perhaps not needed to deal with matching procedure calls and returns. In [17], Reps et al consider the class of IFDS (Interprocedural, finite, distributive, subset) problems, transforming these problems into a graph *reachability* problem. The analysis of the resultant graphs is restricted to those which are *realizable*, for example only considering returns that are matched by calls. In [16], this is characterized as CFL-reachability. However we were unable to see how CFL-reachability could be applied to the particular problem solved by the counters, which quantify precisely the level of nesting achieved.

In summary, we extend affine size-change graphs with return values and counters. These allow us to capture various useful behaviours of a function and function call sequence, without affecting the decidability of the size-change algorithm. Not discussed here are other additions which allow initial parameter values to be saved and returned for later use.

4 From well-founded types to boundedness

With the introduction of affine constraints, it becomes feasible to lay the termination decision upon the concept of *boundedness* rather than (strictly) inductively defined variables. In the original paper on size-change termination, the principal concern was with well-founded data. If it was possible to establish that every infinite sequence of function/procedure calls would result in an infinite descent in some program values, then termination was established.

However, this does not (for instance) handle the case where every infinite sequence of function/procedure calls would result in an infinite increase in some values, which are bounded above by some condition that inhibits the respective calls. We thus accept that either *monotonically increasing* and *bounded above* or *monotonically decreasing* and *bounded below* is sufficient to establish the termination property. This is realized by changing the presentation of size-change termination to including a guard, derived from the source program, in any idempotent function specification.

An example of this from the 91 function is in the argument which establishes that a series of calls to function b always terminates. The affine size-change graph for function b is $\{[n] \to [n'] : n' = n + 11\}$, and if we include the guard derived from the source program ($n \leq 100$), then we can imagine a graph of the form $b^\omega : n' > n$, where b^ω represents possibly infinitely many recursive calls to b. We evaluate this graph in two

steps: first that n is monotonically increasing in b, and second that $n \leq 100$ guards all calls to b. As a result of these two statements we can say that b^ω is terminating (i.e. ω is finite).

4.1 Bounded termination

We now give a formal treatment of a new property over our guarded graphs, the property of *bounded-termination*. In previous work on size-change termination, an infinite reduction of a well-founded type led to an impossibility argument for the graph; as no such infinite reduction can occur, then we can conclude that the corresponding function call cannot occur infinitely often. The test for this is referred to here as *reduction-termination*. A similar argument can be made over guarded graphs, and Theorem 1 asserts a sufficient condition to establish bounded-termination. We begin with some preliminary definitions for the domain and range restrictions associated with graphs.

Definition 1. *The domain restriction function* $\lhd\colon \mathbb{P}\, X \times (X \leftrightarrow Y) \to (X \leftrightarrow Y)$[1] *is defined by*

$$\forall D : \mathbb{P}\, X;\ G : (X \leftrightarrow Y) \bullet D \lhd G = \{(a, b) : G \mid a \in D\}$$

Definition 2. *The range restriction function* $\rhd\colon (X \leftrightarrow Y) \times \mathbb{P}\, Y \to (X \leftrightarrow Y)$ *is defined by*

$$\forall G : (X \leftrightarrow Y);\ R : \mathbb{P}\, Y \bullet G \rhd R = \{(a, b) : G \mid b \in R\}$$

A recursive function $f(x_1, \ldots, x_n)$ is monotonic if it has at least one argument x_i such that $x_i' > x_i$ or $x_i' < x_i$. In addition, we define a new set, the restricted domain of a function, which is defined over self recursive monotonic functions such as the ones we are approximating with the graphs. This allows us to define bounded termination.

Definition 3. *Given a self-recursive function f defined over a domain r, the* ***restricted domain*** $\mathrm{dom}(f^n, r)$ *of recursive function f is*

$$\{v \in \mathrm{domain}(f) \mid (\{v\} \lhd f^n \rhd r) \neq \varnothing\}$$

Definition 4. *A recursive and monotonic function f over a domain r is* ***bounded-terminating by*** *r if, for all $v \in r$*

$$\exists n > 0 : (\{v\} \lhd f^n \rhd r) = \varnothing$$

The idea here is that the restricted domain in some sense defines the allowable set of input values for the function f^n, and if this set is empty, then the function f cannot be applied to itself n times.

During the analysis, the function f may be approximated by a monotonic affine relation. Consequently, we need to restrict the domain r such that bounded-termination of an approximation of f inevitably implies boundedness of repeated calls to f during runtime. A sufficient condition that ensures this is the definition of a *half-space* domain r below:

[1] $\mathbb{P}\, X$ is the powerset of X, $X \leftrightarrow Y$ the set of all affine relations between X and Y.

Definition 5. *Given a recursive and monotonic affine relation g over a domain r, let $\bar{r}$ be the complement of r. The domain r is said to be a* ***half-space domain*** *if, for all $v \notin r$,*

$$\forall n > 0 : \text{range}(\{v\} \lhd g^n) \subseteq \bar{r}$$

We now can establish a testable condition, sufficient to assert that a function f (and its approximation g) is bounded-terminating, and closely associated with function termination in that if a function is bounded-terminating, then it must be terminating. Bounded termination subsumes the reduction-termination test used for size-change termination. Any reduction-terminating graph is bounded-terminating, but a bounded-terminating graph may not be reduction-terminating.

Theorem 1. *Given a self recursive function f which is monotonic, and over a half-space domain r, then if $r \subseteq \text{dom}(r \lhd f^+ \rhd \bar{r}, \top)$ then f is bounded-terminating by r.*

Proof. Suppose f is *not* bounded-terminating, then by the definition of bounded-termination and the continuous property of r, there exists a value $v_1 \in r$ such that $\forall n : (\{v_1\} \lhd f^n \rhd \bar{r}) = \varnothing$ (there is at least one value in r such that the range never gets outside the domain/guard), and so

$$\exists v_1 \in r : (\{v_1\} \lhd f^+ \rhd \bar{r}) = \varnothing \qquad (\text{Defn. of } \mathrm{f}^+ = \textstyle\bigcup_{\mathrm{n}>0} \mathrm{f}^{\mathrm{n}})$$

However, we are also told that

$$\begin{aligned}
& r \subseteq \text{dom}(r \lhd f^+ \rhd \bar{r}, \top) \\
\Rightarrow\ & r \subseteq \{v \in \text{domain}(f) \mid (\{v\} \lhd (\{r\} \lhd f^+ \rhd \bar{r}) \rhd \top) \neq \varnothing\} && (\text{Defn. of dom}) \\
\Rightarrow\ & r \subseteq \{v \in \text{domain}(f) \mid (\{v\} \lhd f^+ \rhd \bar{r}) \neq \varnothing\} && (\text{Properties of } \top, \lhd, \rhd) \\
\Rightarrow\ & \forall v_x \in r : (\{v_x\} \lhd f^+ \rhd \bar{r}) \neq \varnothing
\end{aligned}$$

This contradicts the first supposition, and so we conclude (by contradiction) that f is bounded-terminating by r. □

The intution here is that the term $r \lhd f^+ \rhd \bar{r}$ specifies all the graphs that must have a range outside the guard r of the function call f. If the guard is a subset of the restricted domain, then all initial values for the function must lead out of the restricted domain, leading to termination of the recursive call. This theorem provides a mechanism for testing if a final size-change graph for guarded function calls leads to a terminating program. The test is not computationally expensive, requiring only domain and range restriction and set inclusion to an existing graph in the final step of termination analysis.

4.2 Termination analysis

Termination properties of a program may be derived from a closure computation over the size-change graphs. A crucial administrative task in ensuring termination of this 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 providing the different classifications, and the affine graphs providing concrete instances of some of these classifications.

Definition 6. [1] *An affine graph is called an* ***abstract*** *graph if each of its destination parameters y_i is related to the source parameters x_j in an affine relation $y_i \,\mathrm{op}\, x_j$ where* $\mathrm{op} \in \{=, <, >, \geq, \leq\}$.

It is easy to see that the set of abstract graphs is finite, and that the set of graphs used by LJB-analysis is a subset of the abstract graphs.

Given a program, we generate a finite set of abstract graphs $\mathcal{A}$ as needed, and associate with each abstract graph A_g a guard r_g derived from an analysis of the conditional structure of the program. Let $\mathcal{F}$ be the corresponding affine graphs that can be created from the program. We then associate each affine graph with an abstract graph in $\mathcal{A}$, as follows:

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

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

The major step of the termination analysis is the algorithm $\mathcal{T}$, which builds a closure of an initial set of affine size-change graphs, constructing compositions of existing affine size-change graphs until no new affine graphs are created:

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

After the algorithm $\mathcal{T}$ terminates, then a second step is to examine the resultant set of graphs, and establish if each idempotent recursive call graph is bounded-terminating. The function **`classify`** returns a pair (x, A_k), with x 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 legitimate pairs of existing size-change graphs.

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 [9]. 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.

The main idea of the algorithm can be described using the following metaphor. Imagine each abstract graph as a container, which will contain those affine graphs under its minimal association. The containers will have a label with the corresponding abstract graph. Thus, some containers will be considered as idempotent containers when they are labeled with an idempotent abstract graph.

At first, 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.

Such an iteration process will eventually terminate, with no more new affine graphs created. The algorithm terminates when $\mathcal{C}'$ is no longer changing. $\mathcal{C}'$ can change in only two ways, either by creating an association for a new abstract graph A_g, or by replacing an existing set element (x, A_g) with a new (g, A_g) and neither of these can be done an infinite number of times. The proof of termination for the algorithm is similar to that found in the technical report [2].

The second step of the termination analysis is to identify those non-empty idempotent containers, and then check to see if the affine graphs therein contain a varying component that matches the guard r_g for that abstract graph (A_g). The test derived from Theorem 1 is computationally inexpensive, and captures many indicators of termination including ones that rely on even/odd tests, and increasing or reducing arguments towards a guard.

If all these idempotent graphs pass the test, then we conclude that the associated program terminates. If one of these graphs does not pass the test, we shall conclude that the associated program does not belong to the size-change terminating programs. The affine-based analysis described here captures more information from a program, but is still decideable.

In summary, the termination analysis is a two-step process. The first step uses algorithm $\mathcal{T}$ which performs graph composition repetitively until a fixed point is reached, at which point the algorithm halts. The second step involves testing the resultant idempotent graphs for the bounded-termination property.

4.3 Bounded termination examples

```
p(x, y) = if x < 10 then
              if y > 7 then
                  p(x − 1, y − 1)_a
              else
                  p(x + 3, y)_b;
```

From a static analysis of this function, the affine graph a_{call} is restricted in it's domain to $x < 10 \wedge y > 7$ and the affine graph b_{call} is restricted in it's domain to $x < 10 \wedge y \leq 7$. We use these restrictions to generate initial extended affine graphs which are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating a final classification of graphs:

Trace	Final affine graphs
$g_1 = a^+$	$\{[x,y] \to [x',y'] : x' + 1 \leq x \leq 9 \wedge 7 + x \leq y + x'\}$
$g_2 = b^+ + g_2 g_3$	$\{[x,y] \to [x',y'] : x + 3 \leq x' \leq 12 \wedge y \leq 7\}$
$g_3 = a^+ b^+ + g_3 g_2$	$\{[x,y] \to [x',y'] : x \leq 9 \wedge x' \leq 12 \wedge 10 + x \leq y + x' \wedge 8 \leq y\}$

The composition of any pairing of these graphs generates no new graphs, and so the algorithm $\mathcal{T}$ finishes. Step two of the analysis involves testing the idempotent affine graphs using $r_1 = \{(x,y) \mid x < 10 \wedge y > 7\}$ and $r_2 = \{(x,y) \mid x < 10 \wedge y \leq 7\}$ and confirms that *the function terminates for all traces.*

$$r_1 \subseteq \mathrm{dom}(r_1 \lhd g_1 \rhd \bar{r_1}, \top)$$
$$r_2 \subseteq \mathrm{dom}(r_2 \lhd g_2 \rhd \bar{r_2}, \top)$$
$$r_1 \subseteq \mathrm{dom}(r_1 \lhd g_3 \rhd \bar{r_1}, \top)$$

This test is surprisingly general, capturing many indicators of termination including ones that rely on even/odd tests as in:

```
f(x, y) = if even(x) then
                f(x + y, c)_a;
```

The call argument c is a constant that may be either even or odd, and we express the guard as $r_1 = \{[x,y] : (\exists\, \alpha : 2\alpha = x)\}$. The boundedness test returns

$$r_1 \subseteq \mathrm{dom}(r_1 \lhd g_{c=\mathrm{odd}} \rhd \bar{r_1}, \top) = \mathrm{true}$$
$$r_1 \subseteq \mathrm{dom}(r_1 \lhd g_{c=\mathrm{even}} \rhd \bar{r_1}, \top) = \mathrm{false}$$

In summary, we use a guard derived from a static analysis of the program in the final step of the size-change termination algorithm along with a testable sufficient condition to establish if an arbitrary self-recursive guarded call is bounded-terminating. This handles monotonically decreasing and bounded below, as well as monotonically increasing and bounded above, and combinations of these.

5 Termination for $f91$

We now give details of the steps of the analysis for the 91 function, which uses bounded-termination, CFG and counter analysis. The CFG for the 91 function is $\mathcal{P} \to a \mid b\mathcal{P}c\mathcal{P}$. We approximate the CFG by one that combines a value affine graph with the appropriate call graph c: $\mathcal{P}_2 \to b\mathcal{P}_2 \mid ac\mathcal{P}_2$. We also include a counter c to control the frequency of occurence of ac's with respect to b's. From a static analysis of the program, the affine graph a_{val} is restricted in it's domain to $n > 100$ and the affine graph b_{call} is restricted in it's domain to $n \leq 100$. We use these restrictions to generate the initial extended affine graphs:

$$\{[n,r,c] \to [n',r',c'] : n' = n - 10 \wedge c' = c - 1 \wedge n > 100 \wedge c > 0\}$$
$$\{[n,r,c] \to [n',r',c'] : n' = n + 11 \wedge c' = c + 1 \wedge n \leq 100 \wedge c \geq 0\}$$

These graphs are used as the initial graphs in the algorithm $\mathcal{T}$, which then goes about generating all the possible graphs. These lead to a final classification of graphs into containers, with constraints as follows:

Trace	Final affine graph constraints
g_1	$10c + n' = n + 10c' \wedge 0 \leq c' < c \wedge 91 + 10c \leq n + 10c'$
g_2	$11c + n' = n + 11c' \wedge 0 \leq c < c' \wedge n + 11c' \leq 111 + 11c$
g_3	$91 \leq n' \leq 101 \wedge n \leq 100 \wedge 0 \leq c' \wedge 0 \leq c \wedge n + 11c' < 11c + n$

The composition of any pairing of the graphs (g_1, g_2 and g_3) generates no new affine or abstract graphs, and so the algorithm terminates. We can move on to the second step of the analysis of the resultant containers.

First, we identify monotonicity of those idempotent graphs. This can be easily determined from their respective abstract graphs. Next, we test each one using $r_1 = \{(n, r, c) \mid n > 100 \wedge c > 0\}$ and $r_2 = \{(n, r, c) \mid n \leq 100 \wedge c \geq 0\}$:

$$r_1 \subseteq \mathrm{dom}(r_1 \lhd g_1 \rhd \bar{r_1}, \top)$$
$$r_2 \subseteq \mathrm{dom}(r_2 \lhd g_2 \rhd \bar{r_2}, \top)$$
$$r_2 \subseteq \mathrm{dom}(r_2 \lhd g_3 \rhd \bar{r_2}, \top)$$

We can now assert the final result: *the 91 function terminates for all traces.*

6 Related works and conclusion

Termination analysis of programs has a long history. One approach is by limiting the *language* for a program, where it is possible to ensure termination by construction. For example, a theory of inductive types [8] may be used for termination. If any function defined over an inductive type θ is restricted in the form of the definition to one using the elimination rule of θ, then the function is known to terminate. By contrast, the approach followed here is to allow the construction of divergent recursive expressions, and then attempt to determine the termination property from a static analysis.

In [20] Thiemann and Giesl apply the size-change principle in the context of term-rewriting, combining it with existing ordering and dependency pairs approaches. The resultant technique is more powerful than the existing approaches, but still PSPACE-complete. Our approach extends the size-change principle in the context of affine constraints over numerical (integer) computations, and liberates the underlying analysis from the traditional well-foundedness restriction.

In the logic programming arena, there is considerable research in the structure of termination proofs, summarized in [10, 3], with techniques to handle mutual recursion, loop detection and so on. In a system for termination of Mercury logic-functional programs [19], a measure of the difference in total size between the input parameters and output arguments is constructed for each procedure. The measures are solved as a set of linear inequalities, and if they reduce then the program terminates. We can view this as a coarse-grained size-change termination analysis argument. In [11], the Mercury termination analysis is refined using convex constraints for inter-argument relationships. In [18], Serebrenik and De Schreye also address termination of numerical (integer) computations and Lindenstrauss and Sagiv [14] use a query-mapping pairs approach. Each of these approaches have some analogy to our previous work [1], but the framework for the termination analysis requires the choice of a termination metric (level-mappings, for

example, ordered linear sums of parameters). By contrast, our framework is independent of such concerns, and derives termination properties for a large range of programs without any selection of metrics or heuristics.

The binary-unfoldings approach in [6] uses program specialization to add structural information to increase the precision of termination analysis. These techniques may complement our analysis, and it will be interesting to see how they can be applied. In [7, 5] various techniques are applied to FOR-loop programs, but we believe that the more intuitive approach of bounded termination will achieve similar results in a language extended with FOR-loops.

This paper presented a significant extension and variation to the analysis of program termination based on the size-change termination method. The approach has its most natural expression in a functional programming style, but the approach is sufficiently general to be applied to other areas. We have complemented the size-change principle with new techniques which allow the analysis technique to be applied to functions in which the return values are relevant to termination. In addition, we have formally expressed the idea that termination can be derived for a changing argument that is bounded. This idea was initially explored in the work on sized typing in [4]. The use of guards associated with the abstract graphs enable us to extend this directly into the termination argument, without the limitations implied by the previous technique of constraints over program arguments. A new test for bounded-termination using this approach subsumes the previous test for (only) reducing parameters, is computationally inexpensive, and captures termination properties for a wide range of guarded recursive calls, including ones where the guard depends on even/odd properties, or combinations of increasing and decreasing parameters.

The techniques presented in this paper can be enhanced in various ways. For instance, the use of a global variable in capturing a returned value can be replaced by multiple variables to capture values returned from different calls. Other information, such as return arguments from nested calls, can be captured in size-change graphs, in addition to return values. Moreover, the definition of abstract graphs in Section 4.2 can be refined to admit abstract graphs of more complicated affine constraints. More investigation is needed to weigh the benefits gained from these enhancements against the analysis complexity.

References

1. H. Anderson and S. Khoo. Affine-based Size-change Termination. In A. Ohori, editor, *APLAS 03: Asian Symposium on Programming Languages and Systems*, pages 122–140, Beijing, 2003. Springer Verlag.
2. H. Anderson and S. Khoo. Affine-Based Size-Change Termination. Technical Report TRA9/03, National University of Singapore, September 2003.
3. K. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
4. W. Chin and S. Khoo. Calculating Sized Types. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 62–72, 2000.
5. M. Codish, S. Genaim, M. Bruynooghe, J. Gallagher, and W. Vanhoof. One Loop at a Time. In *6th International Workshop on Termination (WST 2003)*, June 2003.
6. M. Codish, K. Marriott, and C. Taboch. Improving Program Analyses by Structure Untupling. *Journal of Logic Programming*, 43(3):251–263, 2000.
7. 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.
8. 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.
9. 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.
10. D. De Schreye and S. Decorte. Termination of Logic Programs: the never-ending story. *The Journal of Logic Programming*, 19-20:199–260, 1994.
11. J. Fischer. Termination analysis for mercury using convex constraints. Technical report, University of Melbourne, 2002. (Honours thesis).
12. S. Khoo and K. Shi. Output Constraint Specialization. *ACM SIGPLAN ASIA Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 106–116, September 2002.
13. C. Lee, N. Jones, and A. 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.
14. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In J. Maluszynski, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*. MIT Press, 1997.
15. Z. Manna and J. McCarthy. Properties of Programs and Partial Function Logic. *Machine Intelligence*, 5:27–37, 1970.
16. T. Reps. Program Analysis via Graph Reachability. In *Logic Programming, Proceedings of 1997 International Symposium*, pages 5–19, Port Jefferson, Long Island, N.Y., 1997.
17. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.
18. A. Serebrenik and D. D. Schreye. Inference of Termination Conditions for Numerical Loops in Prolog. *Lecture Notes in Computer Science*, 2250:654–668, 2001.

19. C. Speirs, Z. Somogyi, and H. Sondergaard. Termination Analysis for Mercury. In *Static Analysis Symposium*, pages 160–171, 1997.
20. R. Thiemann and J. Giesl. Size-Change Termination for Term Rewriting. Technical Report AIB-2003-02, RWTH Aachen, Jan. 2003.

A Syntax of language and Presburger formulæ

The language used in this paper is a simple first-order functional language, defined in Table 1.

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 the event that the conditional e_0 for the **`if-then-else`** cannot be reduced to an affine constraint, then the associated guards for both branches will be **`[true]`**.

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

Formulæ:			
$\phi \in$ **F**			⟨ Formulæ ⟩
	ϕ	$::=$ $\psi \mid \{[v_1, \ldots, v_m] \rightarrow [w_1, \ldots, w_n] : \psi\}$	
	ψ	$::=$ $\delta \mid \neg\psi \mid \exists v.\psi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$	
Size Formulæ:			
$\delta \in$ **Fb**			⟨ Boolean expressions ⟩
	δ	$::=$ True $\mid$ False $\mid a_1 = a_2 \mid a_1 \neq a_2$	
		$\mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2$	
$a \in$ **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