Size-Change Termination with Difference Constraints

AMIR M. BEN-AMRAM

This paper considers an algorithmic problem related to the termination analysis of programs. More specifically, we are given bounds on differences in sizes of data values before and after every transition in the program's control-flow graph. Our goal is to infer program termination via the following reasoning ("the size-change principle"): if in any infinite (hypothetic) execution of the program, some size must descend unboundedly, the program must always terminate, since infinite descent of a natural number is impossible.

The problem of inferring termination from such abstract information is not the halting problem for programs and may well be decidable. If this is the case, the decision algorithm forms a "back end" of a termination verifier and it is interesting to find out the computational complexity of the problem.

A restriction of the problem described above, that only uses monotonicity information (but not difference bounds), is already known to be decidable. We prove that the unrestricted problem is undecidable, which gives a theoretical argument for studying restricted cases. We consider a case where the termination proof is allowed to make use of at most one bound per target variable in each transition. For this special case, which we claim is practically significant, we give (for the first time) an algorithm and show that the problem is in PSPACE, in fact that it is PSPACE-complete. The algorithm is based on combinatorial arguments and results from the theory of integer programming not previously used for similar problems.

The algorithm has interesting connections to other work in termination, in particular to methods for generating linear ranking functions or invariants.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; F.2.2 [**analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Abstraction, program analysis, size-change graph, size-change termination, termination analysis

1. INTRODUCTION

Termination analysis is one of the fundamental problems of software verification. The issue also arises in designing certain meta-programs, e.g., interpreters for Logic Programs [Naish 1985; Schreye and Decorte 1994] and partial evaluators [Jones 1988; Jones et al. 1993]. As the general halting problem is undecidable, every method for termination analysis consists, in principle, in identifying some *subproblem* that is decidable. This can be done in a more or less structured manner; a

Author's address: Amir M. Ben-Amram, The Academic College of Tel-Aviv Yaffo, Tel Aviv, Israel. Email: amirben@mta.ac.il.

Part of this work was done while the author was visiting DIKU, the University of Copenhagen, Denmark.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2006 ACM

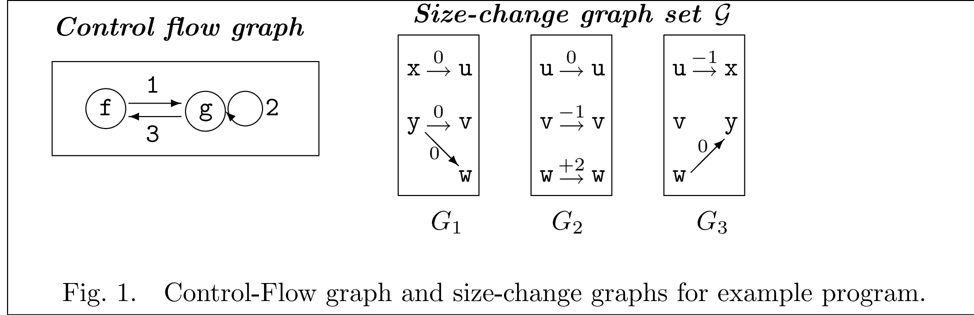


Fig. 1. Control-Flow graph and size-change graphs for example program.

structured work proceeds according to the principles advocated in Theoretical Computer Science—a subproblem is identified and defined in an abstract and concise fashion, then its decidability and complexity are determined. This paper presents such a study that extends previous work on a method called *size-change termination.* In this section, we introduce the problem informally, define the aims of the paper and relate it to previous work. We also summarize the results obtained. A formal presentation is given in Section 2.

1.1 Size-Change termination

The *Size-Change Termination principle* [Lee et al. 2001] breaks the problem of proving that a program always terminates to the following two steps: first, the program is abstracted into a control-flow graph combined with so-called *size-change graphs.* The latter describe changes in the size of data values as they propagate across program transitions. The second step is to verify the following claim:

> Every infinite computation would cause infinite descent in the size of some data.

We assume that "size" is a natural number, hence such infinite descent is impossible, which shows that infinite computation is impossible.

For a concrete example, consider the following program, in a simple first-order functional language with natural numbers as data.

```
f(x,y)   = ¹g(x,y,y)
g(u,v,w) = if uv=0 then 1 else ²g(u,v-1,w+2) + ³f(u-1,w)
```

The three function calls have been labeled 1, 2 and 3 for reference. The *control flow graph* (or *call graph*) is shown in Figure 1. Each call site c is associated with a *size-change graph* G_c that describes how data change in size across the call (here, the size of x is just its value). More precisely, the label δ on an arc in these graphs represents an addition of (at most) δ to the value.

The information in the size-change graphs can be used for a termination proof, as follows: Consider (in order to prove it impossible) any infinite path cs in the control-flow graph.[1] There are two cases to consider. **Case 1:** the path ends with

[1] This analysis treats every path in the control-flow graph as possible. This is a common approach in static program analysis, and does not preclude the utilization of information from conditionals in the *production* of size-change graphs.

an infinite sequence of 2's. By looking at the corresponding size-change graph G_2 we can ascertain that the value of v descends infinitely. **Case 2:** all runs of 2's are finite, i.e., the subpath 13 occurs infinitely many times. In this case, the value of u in function g (equivalently, x in function f) descends infinitely. Since both cases involve infinite descent, we conclude that the program always terminates.

1.2 Abstract and Conquer

An important property of the approach illustrated in the above example is its two-stage structure: in the first stage, the subject program is abstracted to a combinatorial structure, an instance of a system which is much simpler than a programming language. We call it the *abstract system*. In the second stage, we only analyze the abstract system, proving a property that implies the termination of the original program. This construction mitigates an inherent problem in applying the theory of Algorithms and Complexity to software verification: natural verification problems, of which termination is the primary example, are undecidable. But for the problem of deducing termination from the abstract system we may be able to provide a complete algorithm, and the effort in investigating it is rewarded in obtaining a tool that is relatively programming-language independent, as many language-specific features are abstracted away in the first stage. We propose the following research program:

(1) Choose a simple system that captures certain properties of programs that are useful to solving a verification problem. Bread-and-butter examples of such systems are finite-state machines, Petri nets and various constraint domains.

(2) For the system to be interesting, there has to be a way of abstracting ordinary programs into such systems that is convincingly successful in retaining the properties of interest (this stage involves an inherent loss of precision).

(3) Investigate the complexity of the appropriate decision problems over the abstract system. Positive results (algorithms) mean that this system is a tool that, together with tools for the abstraction stage, can provide an approach for doing verification tasks; negative results, such as undecidability, are indicators that the system has to be restricted or further simplified. Thus, research produces a map of abstract systems, drawing the boundaries of decidability and complexity that allow for informed design decisions in approaching the verification tasks.

In this paper we consider the abstract system of size-change graphs, more properly called δSCT graphs (to distinguish them from simpler "size-change graphs" considered in previous work); the precise definitions will be given in the following section. Obtaining size-change graphs from a source program is an already researched topic; for example, [Ullman and Gelder 1988; Brodsky and Sagiv 1991; Lindenstrauss and Sagiv 1997b] all describe methods that can be used to generate such graphs for logic programs. Lindenstrauss and Sagiv's work resulted in a termination analyzer for Prolog programs called Termilog. However, they chose to further simplify the abstract system by ignoring difference values (see the next subsection for further explanation). Thus, they avoided addressing the decision problem for the δSCT abstraction.

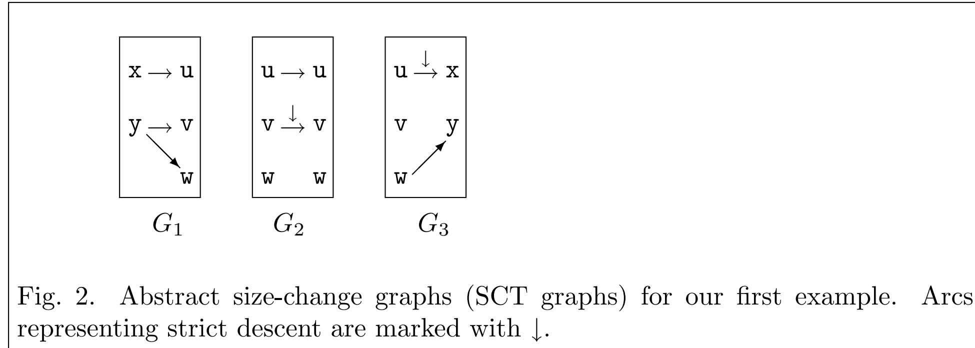


Fig. 2. Abstract size-change graphs (SCT graphs) for our first example. Arcs representing strict descent are marked with ↓.

1.3 A previously-studied variant: SCT

Let us revisit our initial example. It is not hard to observe that for the termination proof, the exact values represented by the labels on the size-change graph arcs are immaterial. We could, in fact, abstract the given size-change information further to a finite domain so that there are only two types of arcs, one representing a non-increase in value, and another standing for strict descent. This abstraction yields *SCT graphs* [Lee et al. 2001], see Figure 2.

The main results in [Lee et al. 2001] addressed the complexity of deducing termination from SCT graphs. That paper can be seen as analyzing the abstract system of SCT graphs in the spirit of our research program. In this case, it turns out that practice preceded theory, in that a decision procedure for SCT graphs already forms part of an algorithm for termination analysis of Datalog programs in [Sagiv 1991], later extended to Prolog in the `Termilog` system [Lindenstrauss and Sagiv 1997b]. In these algorithms, SCT analysis is not isolated from other concerns, notably the treatment of instantiation patterns. Sagiv and Lindenstrauss mention that SCT is a simplification of a more precise abstraction, specifying difference constraints, which they do not handle directly.

The termination analyzer of [Codish and Taboch 1999] implements a closure computation, which is also the basis of the SCT algorithms mentioned above, but in a generic way that allows for various abstract domains to be used. With the domain of size-change graphs, the system decides SCT. In this sytem, graphs with arbitrary size-change labels can be represented, but cannot be analyzed precisely (the generic closure algorithm is inappropriate because the closure set may become infinite).

1.4 δSCT

Our goal is to analyze the complexity of deciding when δSCT graphs imply termination. For a starter, here is a minimal example (due to Amir Pnueli) where the SCT abstraction does not suffice for proving that the program terminates.

```
f(x,y) = if x<2 or y<1 then ... else if ... ¹f(x-2,y+1)
                                 else ... ²f(x+1,y-1)
```

Size-change graphs for the two calls are shown in Figure 3; it is not hard to see that SCT graphs are insufficient to prove termination since they yield no helpful

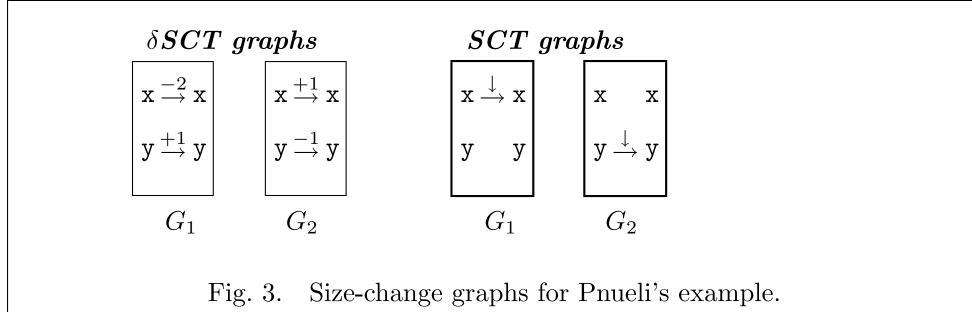


Fig. 3. Size-change graphs for Pnueli's example.

information on the sequence G_1G_2. But, employing the information in the δSCT graphs, it *can* be verified that an infinite sequence must cause unbounded descent; a precise argument is given in the next section.

In general, we claim that it is interesting to investigate the δSCT abstraction because it is more expressive than the SCT one, and existing termination analyzers would be strengthened if the decision procedure at the end of the chain could handle δSCT.

A point of terminology: [Lee et al. 2001] used the term SCT for the decision problem of interest, i.e., the set of systems of SCT graphs (abstractions of programs) that imply termination. Similarly, the set of systems of δSCT graphs that imply termination will be simply called δSCT.

1.5 Main results in this paper

We prove that δSCT is undecidable in general, which gives a theoretical argument for studying restricted cases. SCT, of course, is one such restriction. Here, we consider a different restriction, where the difference values are not ignored as in SCT. Instead, we allow the termination proof to make use of at most one bound (incoming size-change arc) per target variable in each transition. This restriction is motivated, first, by observing that the generality of allowing a conjunction of bounds is often unnecessary with practical examples, and secondly by the fact that the resulting problem is decidable. We give (for the first time) a decision algorithm for the problem and show that it is PSPACE-complete.

2. DETAILED DEFINITIONS AND STATEMENT OF RESULTS

In this section we specify the abstract program representation which we set out to analyze, called an *annotated control-flow graph* or ACG. We then specify and justify the property of interest: what these graphs have to fulfill so that termination can be deduced. Finally we summarize our results on the complexity of this property.

2.1 Annotated control-flow graphs and their semantics

The input to our problem is a directed *control-flow graph* where every arc is annotated with a *size-change graph.* This structure is called an *annotated control-flow graph* (ACG).

For convenience in explaining the relationship of these data to actual programs, we assume that our data represent a first-order pure-functional program. Nodes of the control-flow graph represent program functions, where each function f has a list

of "parameters" denoted $\mathit{Param}(f) = \{f^{(1)}, f^{(2)}, \dots\}$. In this setting, CFG arcs represent function calls. We identify call expressions by labels, as in the previous examples. We write $f \xrightarrow{c} g$ to indicate a call from f to g labeled c.

We should emphasize at this point that the above choice is simplistic and largely a matter of habit: there is nothing in the δSCT formulation itself to fix the type of source programs it applies to or the translation of programs to ACGs. For example, a natural way to represent an imperative program is: CFG nodes are program points, $\mathit{Param}(f)$ is a set of variables, and CFG arcs represent transitions from point to point. As for other programming styles, [Jones and Bohr 2004] and [Thiemann and Giesl 2005] show how to effectively apply this framework to high-level functional programs and term-rewriting systems, respectively, while adaption to Prolog might follow the treatment of either [Lindenstrauss and Sagiv 1997b] or [Codish and Taboch 1999].

It is also worth noting that a given program may be represented as an ACG in different ways, depending on the program analysis used to create the ACG. Even for the first-order functional language, assigning nodes to function names is not necessarily best: assigning them to call sites may be very helpful in refining the size-change information (arcs then denote computation paths that lead from a call site in function f, say to g, through the body of g, up to a call site in g). For a demonstration of the power of this approach see [Manolios and Vroon 2006]. Naturally, it can also be adapted to other programming styles.

A *size-change graph* G associated with a control-flow arc $f \to g$ (notation: $G : f{\to}g$) is a bipartite directed graph with *source* set $A = \mathit{Param}(f)$; *target* set $B = \mathit{Param}(g)$; and a set of labeled arcs $x \xrightarrow{\delta} y$ with $x \in A$, $y \in B$ and $\delta \in \mathbb{Z}$. It can also be presented as a matrix $G_{|A|\times|B|}$ where $G[i,j]$ is the label on arc $f^{(i)} \to g^{(j)}$ (there can be at most one such arc), or ∞ if there is none.

For simplicity's sake, we let the subject program p be fixed for the rest of the discussion. A size-change graph represents a set of assertions, also referred to as constraints, on the relation of parameter values in the calling function and in the call expression. Arc $x \xrightarrow{\delta} y$ indicates that the size of the argument passed for y in the call to g is bounded by δ plus the size of f's argument x. To express this more precisely we introduce *abstract states* and *state transition sequences*; these are in essence descriptions of program behaviour, that are relatively programming-language independent.

An abstract state (state for short) is a pair $(f, \vec{v})$ with f a function name and $\vec{v}$ a vector of natural numbers. The intended meaning of $(f, \vec{v})$ is to represent an assignment of sizes to the arguments of f at some point during computation. That is, $\vec{v}[i]$, the ith coordinate of $\vec{v}$, represents the size of $f^{(i)}$.

A *state transition* $(f, \vec{v}) \xrightarrow{c} (g, \vec{u})$ is a pair of states connected by a call $f \xrightarrow{c} g$. The transition is *reachable* if it can arise in a computation of p on some input.

A *state transition sequence* (STS) is a (finite or infinite) chain of state transitions:

$$\mathit{sts} = (f_0, \vec{v}_0) \xrightarrow{c_1} (f_1, \vec{v}_1) \xrightarrow{c_2} (f_2, \vec{v}_2) \xrightarrow{c_3} \dots,$$

When arguing about a given program p, we are interested in the *reachable state transition sequences*, those sequences that may arise in a computation of p. The *call sequence* associated with *sts* is $c(\mathit{sts}) = c_1 c_2 c_3 \dots$.

Definition 2.1. Let p be a program, and $f \stackrel{c}{\to} g$ a call in p. A size-change graph $G : f \to g$ is *safe for this call* if for every arc $f^{(i)} \stackrel{\delta}{\to} g^{(j)}$ in G, if $(f, \vec{v}) \stackrel{c}{\to} (g, \vec{u})$ is any reachable state transition, then $\vec{u}[j] \le \vec{v}[i] + \delta$.

A safe ACG for program p is one that includes, for every call $f \stackrel{c}{\to} g$ in p, a graph G_c safe for this call.

Remark: The term *safe* is from [Lee et al. 2001]. Using different terminology, we would say that G above is a conservative abstraction of the transition relation associated with call $f \stackrel{c}{\to} g$.

Let $\mathcal{G}$ be an ACG. Following [Lee et al. 2001], we define a *multipath* over $\mathcal{G}$ (for short, a $\mathcal{G}$-multipath) to be a finite or infinite sequence of graphs $G_1 G_2 \dots$ that labels a corresponding directed path in the control-flow graph. For a finite multipath $M = G_1 \dots G_n$, we write $M : f \leadsto g$ if f is the source function of G_1 and g the target of G_n. The length of the multipath is denoted by $|M|$.

A *thread* in such a multipath is a finite or infinite path $x_k \stackrel{\delta_{k+1}}{\to} x_{k+1} \stackrel{\delta_{k+2}}{\to} \dots$ such that for all $j \ge k$, $x_j \stackrel{\delta_{j+1}}{\to} x_{j+1} \in G_j$. Note that a thread does not necessarily start at G_1. For a finite thread t, we write $t : x_k \leadsto x_n$ to indicate the initial and final nodes (parameters) of the thread.

A thread that spans the length of the multipath is called *complete*.

Definition 2.2. Let $t = x_0 \stackrel{\delta_1}{\to} x_1 \stackrel{\delta_2}{\to} \dots$ be a thread of at least n arcs. $\int_0^n t$ is shorthand for $\sum_{i=1}^n \delta_i$. If t is finite, $\int t$ is the sum over all of t.

A finite thread is *descending* if $\int t < 0$; an infinite thread t is *of infinite descent* if $\underline{\lim}_{n\to\infty} \int_0^n t = -\infty$ (the infimum limit is used since the sequence is not monotonic in general).

2.2 δSCT

Definition 2.3. An annotated control-flow graph $\mathcal{G}$ satisfies the *δSCT condition* if every infinite multipath contains at least one thread of infinite descent.

This condition (adapted almost verbatim from [Lee et al. 2001]), is purely combinatorial, in the sense that it does not refer to the program represented by $\mathcal{G}$ or to the semantics of size-change graphs. However, what it intuitively means is the following statement: "Every hypothetical infinite computation by the program would involve infinite descent in the size of some data, therefore infinite computation is impossible". The following theorem formalizes this reasoning.

THEOREM 2.4. *If $\mathcal{G}$ is safe for program* p*, and satisfies δSCT, then no reachable state transition sequence for* p *can be infinite.*

PROOF. Assume in contradition that an infinite reachable STS for p exists,

$$sts = (f_0, \vec{v}_0) \stackrel{c_1}{\to} (f_1, \vec{v}_1) \stackrel{c_2}{\to} (f_2, \vec{v}_2) \stackrel{c_3}{\to} \dots,$$

And let $M = G_{c_1} G_{c_2} G_{c_3} \dots$ be the multipath annotating the path $c(sts)$ in $\mathcal{G}$. By the assumption of the theorem, M has a thread t of infinite descent. A thread does not necessarily start at the beginning of the transition sequence, but to simplify notation, let us assume that it does. Thus

$$t = f_0^{(i_0)} \stackrel{\delta_1}{\to} f_1^{(i_1)} \stackrel{\delta_2}{\to} \dots$$

This simplification loses no generality, because we could consider the suffix of *sts* beginning at the position in which t begins. Consider the sequence of values $\vec{v}_0[i_0], \vec{v}_1[i_1], \ldots$ By definition of safety, we must have

$$\begin{aligned} & \vec{v}_1[i_1] \ \le\ \vec{v}_0[i_0] + \delta_1, \\ & \vec{v}_2[i_2] \ \le\ \vec{v}_1[i_1] + \delta_2, \\ & \quad \ldots \\ \Rightarrow\quad & \vec{v}_n[i_n] \ \le\ \vec{v}_0[i_0] + \delta_1 + \cdots + \delta_n = \vec{v}_0[i_0] + \int_0^n t \end{aligned}$$

for all n. But since t is of infinite descent, there must be some n such that $\vec{v}_n[i_n] < 0$, an impossibility. □

This theorem has a natural converse statement: if $\mathcal{G}$ does not satisfy δSCT, there is a program `p` (in a rudimentary programming language with natural numbers as data) such that $\mathcal{G}$ is safe for `p`, and `p` has nonterminating computations. In fact, the program can be built from $\mathcal{G}$ in a rather straight-forward manner; we leave the details to the interested reader.

For theoretical statements, we use δSCT for the name of the decision problem—formally, the set of (standardized representations of) ACG's that satisfy the above condition.

Having defined δSCT, the SCT condition [Lee et al. 2001] becomes a special case:

Definition 2.5. SCT is the set of instances that satisfy δSCT while including no positive labels.

Arc labels in SCT can clearly be restricted to $\{0, -1\}$; thus there are just two types of arcs. In [Lee et al. 2001] we gave a complexity-theoretic analysis of the SCT problem, concluding that it was complete for PSPACE.

In this work, we analyze the complexity of δSCT. We prove (in Section 3) that the problem is undecidable in general. This is a theoretical motivation for looking at decidable restrictions of the problem; SCT is just such a restriction. In this paper, we present another decidable restriction, which admits positive labels.

Definition 2.6. A size-change graph G is *fan-in free* if

$$x \to y \in G \wedge x' \to y \in G \Longrightarrow x = x'$$

(the notation $x \to y \in G$ indicates that an arc from x to y, of an unspecified label, exists in G).

Intuitively, fan-in in G represents a *conjunction* of different bounds on a target parameter. For example, a call `f(min(x,y))` to function `f(z)` is abstracted as a graph with two arcs: $\mathtt{x} \xrightarrow{0} \mathtt{z}$ and $\mathtt{y} \xrightarrow{0} \mathtt{z}$. Another source of fan-in is the analysis of the guards of a call (the call context). Consider the following program fragment: (from [Lee 2002])

```
f(d,s) = if d=hd(s) then g(hd(s)) else f(d,tl(s))
```

In analyzing the call from `f` to `g`, the condition `d=hd(s)` is known to hold and thus we obtain a graph with arcs $\mathtt{s} \xrightarrow{-1} \mathtt{x}$ as well as $\mathtt{d} \xrightarrow{0} \mathtt{x}$.

Despite their intuitive appeal, it appears that in analysis of "natural" programs such conjunctions play a lesser role than one might suspect. As a (somewhat

anecdotal) evidence, a study [Ben-Amram and Lee 2007] of 123 examples from previous work on Prolog programs [Plümer 1990; Schreye and Decorte 1994; Apt and Pedreschi 1994; Bueno et al. 1994; Lindenstrauss and Sagiv 1997a] revealed material occurrence of fan-in in only a single (contrived) example.[2] In fact, this scarcity of fan-in may be due to the fact that, unlike `min`, other common binary operators do not create fan-in, because of the fact that our constraints only express *upper bounds*. Thus, $\mathtt{max}(\mathtt{x}, \mathtt{y})$ is bounded neither by `x` nor by `y`, and the same goes for `x+y`.

In Section 4 we prove that δSCT instances without material fan-in form a decidable set. In fact we pinpoint its complexity class: it is complete for PSPACE. Section 5 describes a (somewhat surprising) connection of our algorithm to certain termination analyses based on the concept of ranking functions, as well as some other related ideas. Section 6 concludes with some perspectives and open problems.

2.3 Another example

There is a common pattern in programs that gives rise to δSCT instances that are not SCT instances. It involves loops that move elements from list to list, so that while one decreases in size, the other increases; at some points, a full list trades places with an empty one. For an example, consider the following functional-language implementation of QuickSort.

```
sort xs = case xs of
    []     -> []
    (x:xt) -> partition xt [] [] x

partition xs lt rt pvt = case xs of
    []     -> (sort lt) ++ [pvt] ++ (sort rt)
    (x:xt) -> if (x < pvt)
        then partition xt (x:lt) rt pvt
        else partition xt lt (x:rt) pvt
```

An obvious representation of this program's data flow in size-change graph form includes two function names: `sort` with a single input parameter `xs`, and `partition` with input parameters `pvt, xs, lt, rt`. These graphs are as shown in Figure 4. The reader may verify that this size-change graph set does not imply termination: in fact, an infinite multipath such as $G_1G_4G_1G_4G_1G_4\ldots$ has no infinite thread. Interestingly, a size-change termination proof *can* be obtained by employing a somewhat non-obvious trick in constructing the size-change graphs. The trick is to make the constant `[]` appearing in the program into a parameter, so it participates in the size-change graphs, as shown in Figure 5. Specifically, G_1 represents a call in which `[]` is passed for both `rt` and `lt`, hence the arcs connecting these parameters. The calls represented by G_4 and G_5 are dependent on the condition `xs = []`. This

[2] a "material occurrence" is one which cannot be eliminated simply by deleting some arcs without affecting the termination argument. In [Ben-Amram and Lee 2007] we defined a "clean-up" procedure that proved successful in eliminating immaterial fan-in in our SCT benchmark. The interested reader is referred to that article, and encouraged to verify that the same procedure works for δSCT.

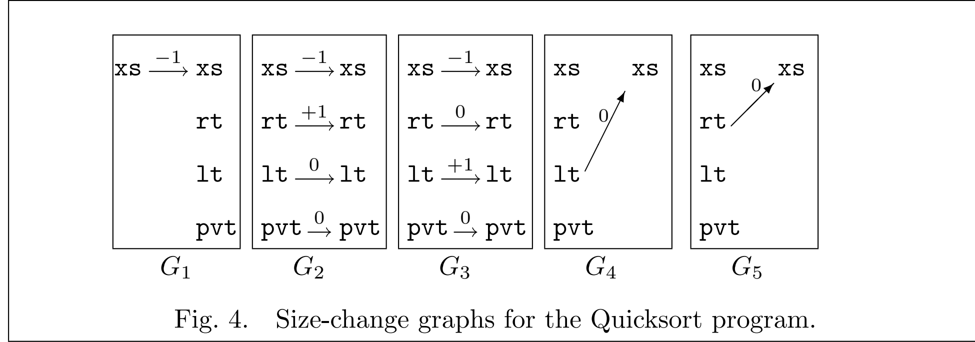


Fig. 4. Size-change graphs for the Quicksort program.

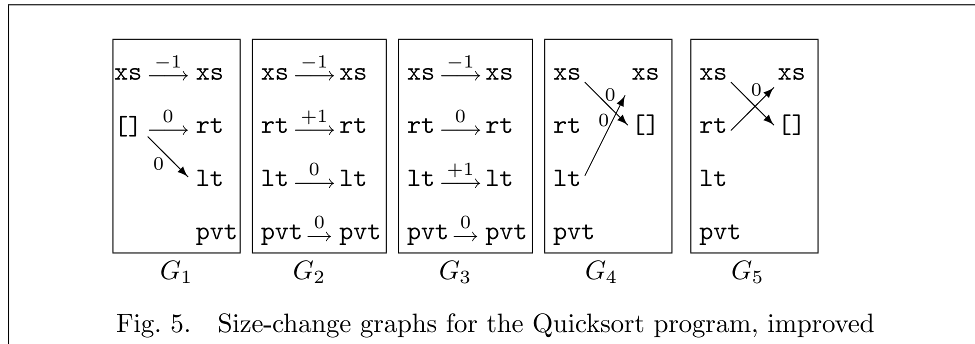


Fig. 5. Size-change graphs for the Quicksort program, improved

allows an analyzer to deduce the arc $\texttt{xs}\xrightarrow{0}\texttt{[]}$. The reader may find it interesting to verify that δSCT is now satisfied.

It is easy to see that this example could also be reduced to SCT by including a node for the *sum* of the lengths of `lt`, `rt` and `xs` in `partition`. This would, of course, necessitate an analyzer that can discover that this particular sum is significant; in Section 5 we show that the mathematics involved in discovering such an invariant is related to solving a restricted case of δSCT, though the methods are not interchangeable in general.

3. UNDECIDABILITY OF THE GENERAL CASE

In this section we prove that δSCT is undecidable in general. We do this by a reduction from a well-known undecidable problem, the halting problem for inattentive (input-free) counter programs, defined next.

A *counter program* is an instruction sequence $p = \texttt{1:I}_1\ \texttt{2:I}_2 \texttt{...m:I}_m$ specifying a computation on variables $\texttt{X}_1,\ldots,\texttt{X}_k$. The variables hold natural numbers. A state of the program is $(\ell, \vec{v})$ where $\ell \in \{1, 2, \ldots, m\}$ represents a program location and $\vec{v}$ represents the values of $\texttt{X}_1,\ldots,\texttt{X}_k$.

Instructions $\texttt{I}_\ell$ have three forms: $\texttt{inc X}_i$, $\texttt{dec X}_i$, and $\texttt{if X}_i\ \texttt{then}\ \ell'\ \texttt{else}\ \ell''$. Here $1 \le i \le k$ and $\ell, \ell', \ell'' \in \{0, 1, 2, \ldots, m\}$.

The program begins at instruction 1 and stops when control flows past instruction m, or when a branch to label 0 is taken. The `if` instruction branches to ℓ' if the value of $\texttt{X}_i$ is positive, or to ℓ'' if it is zero. A `dec` applied to zero may be defined to produce zero, but we prefer to avoid this case completely and assume

that the program never decrements a zero value (tests can be inserted into any given program to ensure this). In an inattentive (input-free) program, the initial value of all variables is 0.

The following set is known to be undecidable:

$$\mathcal{H} = \{p \mid p \text{ is an inattentive counter program that halts}\}.$$

THEOREM 3.1. *$\mathcal{H}$ can be reduced (by a computable many-one reduction) to the complement of δSCT. Hence, δSCT is undecidable.*

The rest of this section describes the reduction. Let program $p = \mathtt{1{:}I_1\ 2{:}I_2 \ldots m{:}I_m}$ with variables $\mathtt{X}_1, \ldots, \mathtt{X}_k$. We translate p into a program p^* in a simple first-order functional language computing over the integers. A δSCT instance is then produced from p^* in a straight-forward way (in fact, for the purpose of this proof, the program is just a convenient manner of specifying the instance).

Program p^* will have functions $\{F_0, F_1, \ldots, F_m\}$, each one of $2k+2$ parameters named $\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z}$. The program is constructed so that in its size-change graph representation, every infinite multipath will have infinite descent, *unless* it can be broken into finite segments that describe terminating computations of p (this logic also underlies the PSPACE-hardness proof in [Lee et al. 2001]).

The roles of the parameters can be described as follows: $\mathtt{X}_i$ is meant to simulate a variable of p. The corresponding $\overline{\mathtt{X}}_i$ is meant to contain the negated value of $\mathtt{X}_i$ (always non-positive). Parameter $\mathtt{E}$ is used as a *ground line (Erdung)* and supplies a value which the simulation treats as zero. Thus, what actually happens is that $\mathtt{X}_i - \mathtt{E}$ simulates p's variable $\mathtt{X}_i$ while $\overline{\mathtt{X}}_i - \mathtt{E}$ is its negation. Parameter $\mathtt{Z}$ is used to count simulated steps (downwards), so that a computation that simulates a nonterminating run of p will yield a thread of infinite descent.

We next write down the program p^*, using an obvious programming language. The program contains an initial function F_0 plus a function for every instruction of p.

Initial function F_0

$$F_0(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z}) = \\ \quad F_1(\mathtt{E}, \mathtt{E}, \ldots, \mathtt{E}, \mathtt{E}, \mathtt{E}, \mathtt{E})$$

Function F_ℓ, for instruction $\ell : \mathtt{inc\ X}_i$

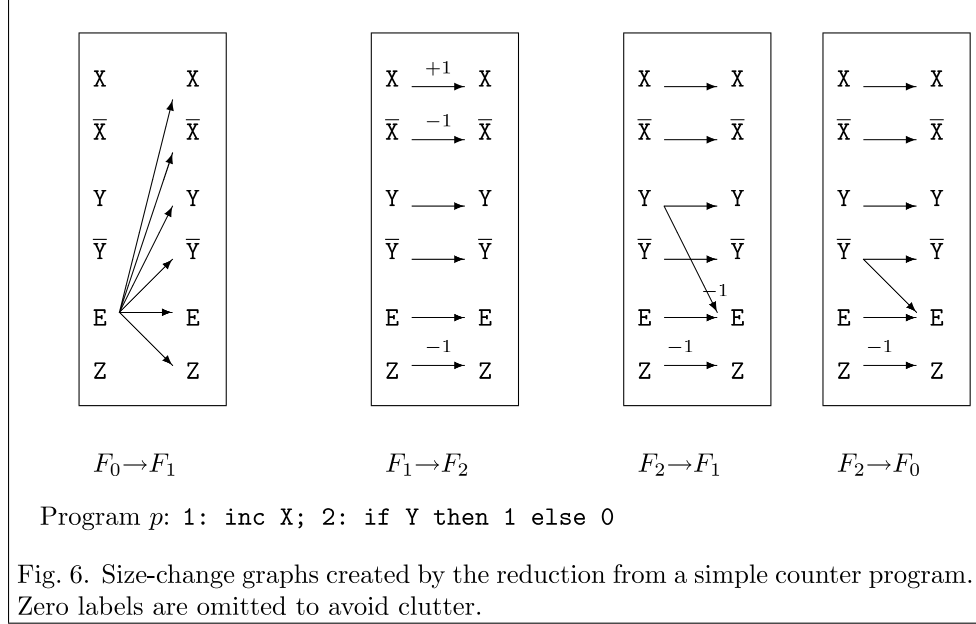
$$F_\ell(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_i, \overline{\mathtt{X}}_i, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z}) = \\ \quad F_{(\ell+1) \bmod (m+1)}(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_i + 1, \overline{\mathtt{X}}_i - 1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z} - 1)$$

Function F_ℓ, for instruction $\ell : \mathtt{dec\ X}_i$

$$F_\ell(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_i, \overline{\mathtt{X}}_i, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z}) = \\ \quad F_{(\ell+1) \bmod (m+1)}(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_i - 1, \overline{\mathtt{X}}_i + 1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z} - 1)$$

Function F_ℓ, for instruction $\ell : \mathtt{if\ X}_i\ \mathtt{then}\ \ell'\ \mathtt{else}\ \ell''$

$$F_\ell(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{E}, \mathtt{Z}) = \ \mathtt{if}\ \mathtt{X}_i > 0 \\ \quad \mathtt{then}\ F_{\ell'}(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{min}(\mathtt{E}, \mathtt{X}_i - 1), \mathtt{Z} - 1) \\ \quad \mathtt{else}\ F_{\ell''}(\mathtt{X}_1, \overline{\mathtt{X}}_1, \ldots, \mathtt{X}_k, \overline{\mathtt{X}}_k, \mathtt{min}(\mathtt{E}, \overline{\mathtt{X}}_i), \mathtt{Z} - 1)$$



Program p: `1: inc X; 2: if Y then 1 else 0`

Fig. 6. Size-change graphs created by the reduction from a simple counter program. Zero labels are omitted to avoid clutter.

We let $\mathcal{G}$ be the set of size-change graphs describing (in the most straight-forward way) the dataflow of p^* (note that `min` is an operator that creates fan-in in the size-change graph). Figure 6 shows the graphs produced for a simple program with two variables. One can easily observe certain properties, such as, all size-change graphs have an arc $\mathtt{E} \stackrel{0}{\to} \mathtt{E}$, and all but the graph for F_0 have $\mathtt{Z} \stackrel{-1}{\to} \mathtt{Z}$. A deeper property is expressed in the following lemma:

LEMMA 3.2. *Suppose that the (unique) computation of program p is described by the STS $\sigma_p = (\ell_1, \vec{v}_1) \to (\ell_2, \vec{v}_2) \to \ldots$, with $(\ell_1, \vec{v}_1) = (1, (0, \ldots, 0))$, and define $\ell_0 = 0$. Then program p^*, started with a call to F_0 in which the value e is passed for* `E`*, goes through the STS*

$$(F_0, \vec{u}_0) \to (F_{\ell_1}, \vec{u}_1) \to (F_{\ell_2}, \vec{u}_2) \to \ldots$$

where

$$\vec{u}_i = (e + \vec{v}_i[1], e - \vec{v}_i[1], e + \vec{v}_i[2], e - \vec{v}_i[2], \ldots, e - \vec{v}_i[k], e, e - i).$$

The corresponding multipath $M = G_1 G_2 \ldots$, with $G_i : F_{\ell_{i-1}} \to F_{\ell_i}$ further satisfies: for all $i \leq |M|$ and $j \leq k$, multipath $M_i = G_1 G_2 \ldots G_i$ has a thread $t_j : \mathtt{E} \rightsquigarrow \mathtt{X}_j$ with $\int t_j = \vec{v}_i[j]$ and a thread $\bar{t}_j : \mathtt{E} \rightsquigarrow \overline{\mathtt{X}}_j$ with $\int \bar{t}_j = -\vec{v}_i[j]$.

Verifying this is straightforward, so we omit a formal proof.

Correctness of the reduction. We prove that $p \in \mathcal{H}$ if and only if $p^* \notin \delta\mathrm{SCT}$. Suppose first that p does terminate. Then program p^*, initialized with $\mathtt{E} = 0$, simulates p until its termination, upon which it reaches F_0 again with 0 for `E` and repeats ad infinitum. Thus an infinite call sequence of p^* is obtained which does not have infinite descent.

Suppose now that p does not terminate, and let σ_p be its (infinite) STS. Consider any infinite p^* call sequence and its corresponding $\mathcal{G}$-multipath. If the sequence does not include F_0 infinitely often, the multipath clearly has infinite descent in Z. Thus, we consider a sequence that consists of infinitely many finite segments, all starting and ending at F_0. We will show that over every such segment there is a thread, starting and ending at E, with a negative integral. Therefore, the concatenation of all these segments has infinite descent. Let $G_1 G_2 \dots G_n$ be such a segment, with $G_i : F_{\ell_{i-1}} \to F_{\ell_i}$, and $\ell_0 = \ell_n = 0$ (and, of necessity, $\ell_1 = 1$). Note that σ_p also begins with $\ell_1 = 1$; but it never reaches location 0. So, consider the first i such that ℓ_i differs from the ith label in σ_p (hence, up to ℓ_{i-1} the sequences match). This indicates that $F_{\ell_{i-1}}$ is a function that has more than one possible successor—necessarily one that represents an `if` instruction. Moreover, ℓ_i is the wrong successor. Suppose that the instruction is `if` $\mathtt{X}_j$ `then` ℓ' `else` ℓ''. Let the $(i-1)$st state in σ_p be $(\ell_{i-1}, \vec{v})$; then either $\vec{v}[j] > 0$ and $\ell_i = \ell''$, or $\vec{v}[j] = 0$ and $\ell_i = \ell'$. In the first case, the lemma provides a thread $\bar{t}_j$ over $G_1 \dots G_{i-1}$ such that $\int \bar{t}_j < 0$; graph $G_i : F_{\ell_{i-1}} \to F_{\ell_i}$ has an arc $\overline{\mathtt{X}}_j \xrightarrow{0} \mathtt{E}$, which gives a descending thread from E back to E. This can be completed with arcs $\mathtt{E} \xrightarrow{0} \mathtt{E}$ up to the end of the segment considered. Similarly, in the other case, we have a thread t_j with $\int t_j = 0$ and an arc $\mathtt{X}_j \xrightarrow{-1} \mathtt{E}$ which forms a descending thread from E back to E.

As previously stated, this shows that the infinite $\mathcal{G}$-multipath has infinite descent, completing the correctness proof for the reduction.

Remark. As 2-counter programs are universal [Jones 1997], the above proof shows that δSCT is already undecidable for subject programs with a constant number (six) of parameters in every function. In contrast, the decidable variants (SCT, and δSCT without fan-in, discussed in the next section) move to a lower complexity class when this number is constant (for more details see Section 4).

4. A DECIDABLE RESTRICTION OF δSCT

The goal of this section is to show that δSCT with fan-in free size-change graphs, or more generally without material fan-in (as defined later), constitutes a decidable problem. Our main conclusion is that the problem is PSPACE-complete.

In the course of studying this problem, it became necessary to introduce numerous auxiliary definitions and lemmata. To ease the reading, the section is internally organized as follows: first, in Section 4.1 we give basic definitions and results which recast the problem from its original infinitary formulation (Definition 2.3) into a finitary form. This opens the way for its solution. The following subsections develop the solution. An outline of the development is deferred to Section 4.2, since it relies on Section 4.1.

4.1 Definitions and the basic theorem

Throughout this section, we mostly assume the subject ACG $\mathcal{G}$ to be fixed (saving the repetitive "let $\mathcal{G}$ be an ACG…"). We also assume the control-flow graph to be strongly connected. This is no loss of generality, since an infinite multipath must eventually stay within a single strongly connected component (SCC) of the control-flow graph. Thus, in general, we can process one SCC at a time.

Size-change graph composition is denoted in [Lee et al. 2001] by a semicolon. We will use the same notation, though we have to redefine it to take care of integer labels. In fact, the matrix notation shows that the composition operation is the familiar matrix "min-sum product".

Definition 4.1. Consider two size-change graphs $G_1 : f_0 \rightarrow f_1$ and $G_2 : f_1 \rightarrow f_2$. The *composition* $G_1; G_2$ is a graph G with source function f_0 and target f_2. Its arcs and labels are defined by

$$G[i,j] = \min_k G_1[i,k] + G_2[k,j].$$

OBSERVATION 4.2. *If G_1, G_2 are fan-in free then so is $G_1; G_2$.*

Let $M = G_1 G_2 \ldots G_n$ be a finite multipath. By $\overline{M}$ we denote the labeled size-change graph resulting from "collapsing" the multipath by composition: $\overline{M} = G_1; G_2; \cdots ; G_n$. Observe that every arc $x \rightarrow y$ in $\overline{M}$ represents one or more complete threads $x \rightsquigarrow y$ in M; conversely, every complete thread $t : x \rightsquigarrow y$ in M is represented by an arc $x \xrightarrow{\delta} y$ in $\overline{M}$, where $\delta \leq \int t$ (more precisely, $\delta = \int t$ unless the same arc also represents another thread, of a smaller sum).

The *empty multipath* beginning and ending at function f is denoted Id_f and we define $\overline{\mathrm{Id}_f}$ to be the size-change graph with arcs $x \xrightarrow{0} x$ for all $x \in \mathit{Param}(f)$.

Concatenation of multipaths is written down by simple juxtaposition, e.g., $M_1 M_2$. This notation implies that the target function of M_1 is the source function of M_2.

We call two multipaths M_1, M_2 *equivalent* if $\overline{M}_1 = \overline{M}_2$. We also write this as $M_1 \equiv M_2$. For size-change graphs G_1, G_2 we write $G_1 \simeq G_2$ if the graphs are identical (labels ignored). We call G *idempotent* if $G; G \simeq G$. We call a multipath M *cyclic* if $\overline{M}$ is idempotent.

Let G be a size-change graph with source and target identical. An arc of the form $x \rightarrow x$ in G is called an *in-situ* arc. By $\lfloor G \rfloor$ we denote the *in-situ part of* G, i.e., the subgraph consisting of all in-situ arcs of G.

LEMMA 4.3. *Let G be idempotent and fan-in free. Then $G; G = \lfloor G \rfloor ; G$.*

PROOF. First, we observe that as $\lfloor G \rfloor$ is a subgraph of G,

$$(\lfloor G \rfloor ; G)[i,j] \geq (G; G)[i,j] \tag{1}$$

for all i, j.

Next, let $(G; G)[i,j] = \delta < \infty$; in graph notation, $i \xrightarrow{\delta} j \in G; G$. Thus there is a k such that $G[i,k] + G[k,j] = \delta$, and in particular $k \rightarrow j \in G$. However since $G; G \simeq G$, and G is fan-in free, we must have $k = i$. Thus, $G[i,i] + G[i,j] = \delta$, but $G[i,i]$ is also $\lfloor G \rfloor [i,i]$. Hence

$$(\lfloor G \rfloor ; G)[i,j] \leq \lfloor G \rfloor [i,i] + G[i,j] = (G; G)[i,j]$$

and by (1) we have equality. □

A *sign graph* is a bipartite graph, similar to a size-change graph, however with labels in $\{\downarrow, \bar{\downarrow}, \updownarrow\}$ rather than $\mathbb{Z}$. A sign graph is an abstraction of a δSCT graph; we denote by α the *abstraction operator*, defined as follows: for every size-change graph G, $\alpha(G)$ is a matrix (that can similarly be interpreted as a bipartite labeled

graph) with

$$\alpha(G)[i,j] = \begin{cases} \downarrow & G[i,j] < 0 \\ \overset{=}{\downarrow} & G[i,j] = 0 \\ \updownarrow & G[i,j] > 0 \\ \infty & G[i,j] = \infty. \end{cases}$$

The underlying graph of $\alpha(G)$ has an arc $i \to j$ whenever $\alpha(G)[i,j] \neq \infty$. It is the same graph as G (ignoring labels), and we call $\alpha(G)$ idempotent if and only if G is (so, in both cases, idempotence ignores labels).

The decision procedure we provide for decidable cases of δSCT is based on the following theorem:

THEOREM 4.4. (Basic Theorem) *A strongly connected ACG with fan-in free size-change graphs satisfies δSCT if and only if for every cyclic multipath M, $\overline{M}$ has an in-situ arc with a negative label.*

PROOF. We begin with the "if" direction. Let $\mathcal{G}$ be an ACG with fan-in free graphs such that for every cyclic $\mathcal{G}$-multipath M, $\overline{M}$ has an in-situ arc with a negative label. We will prove that $\mathcal{G} \in \delta$SCT. To this end, let $\mathcal{I} = G_1 G_2 G_3 \ldots$ be any infinite $\mathcal{G}$-multipath. For every pair $s < t$ of positive integers, let $M_{s,t} = G_s \ldots G_t$. For every possible sign graph G over the functions and parameters of $\mathcal{G}$, define the class C_G of pairs (s,t) by

$$C_G = \{(s,t) \mid \alpha(\overline{M_{s,t}}) = G\}$$

Note that the set of sign graphs over a given set of parameters is finite, hence so is the number of classes just defined. Since the set of pairs (s,t) is infinite, Ramsey's theorem [Ramsey 1930; Graham 1981] shows that there is an infinite set I of positive integers such that all pairs (s,t) with $s,t \in I$ are in the same class C_{G°.

Consider now three elements $r,s,t \in I$, with $r < s < t$; then

$$\alpha(\overline{M_{r,s}}) = \alpha(\overline{M_{s,t}}) = \alpha(\overline{M_{r,t}}) = G^\circ$$

and hence, in particular,

$$\overline{M_{r,s}} \simeq \overline{M_{s,t}} \simeq \overline{M_{r,t}}$$

which shows that $\overline{M_{s,t}}$ is idempotent. By our assumption, it has a descending in-situ arc; i.e., G° has an arc $x \xrightarrow{\downarrow} x$. Thus *every* $\overline{M_{i,j}}$ with $i,j \in I$ has an arc $x \xrightarrow{\delta} x$ with $\delta < 0$. It clearly follows that $\mathcal{I}$ has an infinitely descending thread, as was to be proved.

For the converse implication, assume that $\mathcal{G} \in \delta$SCT, and let M be any cyclic multipath. We shall prove that $\overline{M}$ has a descending in-situ arc.

Consider the infinite multipath M^ω (M repeated infinitely many times). By assumption, it has a thread t of infinite descent. Let $x^{(i)}$ be the parameter visited by this thread at the start of the ith copy of M. Thus $x^{(i)} \to x^{(i+1)}$ is always an arc of $\overline{M}$. As in the proof of Lemma 4.3, we deduce that in fact $x^{(i)} = x^{(i+1)}$. I.e., the thread exits every copy of M at the same parameter x. Moreover, since all size-change graphs in this instance are fan-in free, the endpoints uniquely determine

the thread; thus, there is no doubt that $\int_0^{|M|} t = \delta$ where δ is the label on the arc $x \to x$ of $\overline{M}$.

To complete the proof of the theorem, assume that $\delta \geq 0$, and let Δ be the sum of all *negative* labels in the segment of t passing through a single copy of M. Then for any $n \geq 0$ we have

$$\int_0^n t \geq \left\lfloor \frac{n}{|M|} \right\rfloor \delta + \Delta \geq \Delta,$$

and t can not have infinite descent, a contradiction to the choice of t. We conclude that $\delta < 0$, i.e., $\overline{M}$ has a descending in-situ arc. □

A consequence of this theorem is that an algorithm for δSCT can proceed by searching for a *finite counterexample* to the termination criterion, namely a cyclic multipath with no in-situ descent. A similar theorem holds for the SCT variant [Sagiv 1991; Lee et al. 2001], where we further observe that having restricted the set of labels to $\{0, -1\}$, the set of possible collapsed multipaths $\overline{M}$ becomes finite. This set can be effectively computed, and we obtain the so-called “closure algorithm” for deciding SCT.

In the δSCT problem, label values do matter and therefore the set of collapsed multipaths is infinite. This makes its solution more difficult.

4.2 Solution Outline

Our problem, thus, is to decide whether a counterexample-multipath can be generated by a set of size-change graphs. Since the set of collapsed multipaths that can be generated is typically infinite, we are looking for a finite representation of this infinite set that can be effectively tested for the counterexample.

The complication in this stage stems from the existence of two aspects of size-change graph composition: there is an *arithmetic aspect* in the composition, involving the addition of labels. Then there is a *combinatorial aspect*, having to do with the shape of the graph. The two aspects interact, of course.

In the next subsection we tackle the arithmetic aspect in isolation by restricting the graphs to a simple form, called in-situ graphs (corresponding to diagonal matrices). With such simple graphs, the combinatorial aspect is trivial and the arithmetic aspect can be tackled quite easily since the net effect of a multipath is expressible as a linear form.

Next, in Section 4.4, we develop a way to separate the aspects so that instead of an infinite “closure set” we have a finite set of graphs (ensuing from the combinatorial aspect) plus linear forms (for the arithmetic one). This is achieved by a sort of “factorization” of multipaths. However, a straight-forward application of the idea does not yet reduce the search space to a finite set of multipath-representations. It takes a finer analysis, combining both combinatorial properties (introducing a notion of “reduction” for multipaths) and linear-algebraic properties (bounds on the size of solutions) to show that one needs look no further than a certain effectively-computable bound. With this result, the complexity of the decision problem can be established.

4.3 In-situ graphs

An *in-situ* size-change graph is one that only contains in-situ arcs. Let us consider a very simple case of the δSCT problem: there is a single function and all size-change graphs are in-situ (Pnueli's example from Section 1 has this form, as do Petri Nets [Jones et al. 1977]).

Let $\mathcal{G} = \{G_1, \ldots, G_n\}$ be such an instance. Observe that in-situ graphs always commute, i.e., $G_i; G_j = G_j; G_i$. It follows that every $\mathcal{G}$-multipath has an equivalent "normal-form" multipath $M = G_1^{x_1} \ldots G_n^{x_n}$, where G^x is G repeated x times.

It is easy to see that $\overline{M}$ is very easy to compute, also symbolically when the x_i's are formal variables. Checking whether a non-descending multipath exists boils down to checking whether the following system of inequalities has a nontrivial solution:

$$
\begin{array}{l}
G_1[j,j] \cdot x_1 + \cdots + G_n[j,j] \cdot x_n \geq 0, \quad 1 \leq j \leq |Param(f)| \\
x_0, \ldots, x_n \geq 0.
\end{array}
$$

It follows that this subproblem of δSCT is decidable by Linear Programming techniques.[3] For example, Pnueli's example (Figure 3 on Page 5) yields the following constraints

$$
\begin{array}{rcrl}
-2x_1 & + & x_2 & \geq 0 \\
x_1 & - & x_2 & \geq 0 \\
\multicolumn{3}{r}{x_1, x_2} & \geq 0
\end{array}
$$

having only the trivial solution $(0, 0)$.

4.4 Deciding δSCT with fan-in-free graphs

We begin by presenting some tools for reasoning about multipaths of fan-in-free size-change graphs. The assumption of fan-in-freedom is tacitly assumed in the rest of this subsection whenever a multipath is mentioned.

Definition 4.5. Let $G : f \to f$ be in-situ, and $M : f \rightsquigarrow g$ a finite multipath. We define $\frac{G}{M}$ to be a size-change graph with source and target function g and arcs defined by the following rule: for all $y \in Param(g)$ and $n \in \mathbb{Z}$, $y \xrightarrow{n} y \in \frac{G}{M}$ if and only if there is $x \in Param(f)$ such that $x \xrightarrow{n} x \in G$ and $x \to y \in \overline{M}$.

LEMMA 4.6. *Let $G : f \to f$ be in-situ, and $M : f \rightsquigarrow g$ a finite multipath; then $GM \equiv M\frac{G}{M}$.*

Definition 4.7. For fan-in-free graphs $G : f \to f, H : f \to g$ we write $G \prec H$ if $x \to z \in H \Rightarrow x \to x \in G$ for all parameters x, z.

LEMMA 4.8. *$G \prec H \iff G; H \simeq H$.*

PROOF. ($\Rightarrow$) Assume $G \prec H$. If $x \to z \in G; H$, then for some y, $x \to y \in G$ and $y \to z \in H$, and $G \prec H$ (plus fan-in freedom) implies $x = y$, so the arc $x \to z$ exists in H. If $x \to z \in H$, clearly $x \to z \in G; H$.

---

[3] Essentially we are interested in integer solutions, but since this system is homogenous, any non-trivial solution implies an integer-valued one and so ordinary Linear Programming suffices. Terms of the form $\infty \cdot x_i$, not allowed in ordinary LP formulations, can be handled by separately testing the cases $x_i = 0$ and $x_i > 0$. In the former case, column i can be omitted, and in the latter, we omit a row.

($\Leftarrow$) Assume $G; H \simeq H$ and $x \to z \in H$. Then $x \to z \in G; H$, so there are arcs $x \to y \in G$ and $y \to z \in H$. By fan-in freedom, $y = x$. We conclude that $G \prec H$. □

Definition 4.9. Let M be a multipath and consider a division of M into three segments: $M = ABC$, where B is not empty. We call part B a *redex* if $B : f \rightsquigarrow f$ for some function f and also $\overline{B} \prec \overline{C}$. If B is a redex, *reduction of M using B* produces $M' = AC^{\frac{\lfloor \overline{B} \rfloor}{C}}$.

We remark that A or C (or both) can be empty. If C is empty, $M' = A\lfloor\overline{B}\rfloor$.

LEMMA 4.10. *If M' is obtained from M by reduction, $M \equiv M'$.*

PROOF. The definition of $\prec$ can be used to show $M \equiv A\lfloor\overline{B}\rfloor C$. Lemma 4.6 completes the proof. □

Definition 4.11. A multipath M is *reducible* if M can be expressed as ABC, such that B is a redex. We call M irreducible if it is not reducible.

Reducing a multipath several times can result in collecting several in-situ graphs at its end. We group equal graphs together, using power notation as in the last subsection.

Definition 4.12. An *extended multipath* has the form $MG_1^{x_1} \dots G_n^{x_n}$ where $M : g \rightsquigarrow f$ is a multipath, each $G_i : f \to f$ an in-situ graph and each x_i a positive integer. An extended multipath is *normal* if
(i) M is an irreducible $\mathcal{G}$-multipath,
(ii) for each i, $G_i = \lfloor\overline{B}_i\rfloor$ for some irreducible $\mathcal{G}$-multipath B_i, and
(iii) the G_i are pairwise distinct.

A *formal multipath* is an expression $E = MG_1^{x_1} \dots G_n^{x_n}$, where the exponents are formal variables, that also fulfills Requirements (i)–(iii). For natural numbers $k_0, k_1, \dots, k_n$,

$$E(k_0, k_1, \dots, k_n) \stackrel{def}{=} M^{k_0} G_1^{k_1} \dots G_n^{k_n}.$$

Naturally, k_0 can be greater than 1 only if the source and target functions of M are the same.

LEMMA 4.13. *For every cyclic $\mathcal{G}$-multipath $M : f \rightsquigarrow f$, a formal multipath $E = M' \prod_{i=1}^{n} G_i^{x_i}$ exists such that:*
(i) *There are positive numbers $r_1, \dots, r_n$ such that $E(1, r_1, \dots, r_n) \equiv M$.*
(ii) *There are nonnegative numbers $b_0, \dots, b_n$ such that for all sequences $(l_0, \dots, l_n)$ with $l_0 \geq b_0, \dots, l_n \geq b_n$, $E(l_0, l_1, \dots, l_n)$ is equivalent to some cyclic $\mathcal{G}$-multipath.*

Informally, this lemma shows that a multipath can be "factored" into an irreducible part plus a set of in-situ parts. Further, by modifying the exponents of the in-situ parts, other valid multipaths are obtained. A formal multipath E satisfying the lemma is called *a scalable representation of M*. We also say, more briefly, that E represents M.

We will prove this lemma as part of a stronger proposition (Lemma 4.21) further down this section. First, let us show how it contributes to solving δSCT.

THEOREM 4.14. *Let $\mathcal{G}$ be a strongly connected ACG with fan-in-free size-change graphs. $\mathcal{G}$ satisfies δSCT if and only if for every function f, and every scalable representation $E = M\prod_{i=1}^{n} G_i^{x_i} : f \leadsto f$ of a cyclic $\mathcal{G}$-multipath, there is no solution to the Integer Linear Programming problem:*

$$\begin{array}{l} G_0[j,j]\cdot x_0 + G_1[j,j]\cdot x_1 + \cdots + G_n[j,j]\cdot x_n \geq 0, \quad 1 \leq j \leq |Param(f)| \\ x_0,\ldots,x_n > 0 \end{array} \qquad (2)$$

where $G_0 = \lfloor \overline{M} \rfloor$.

PROOF. The theorem follows easily from Theorem 4.4 (The Basic Theorem) and Lemma 4.13 above, noting that any solution to (2) can be multiplied by any positive integer and remain a solution (this allows for applying Part (ii) of Lemma 4.13). □

Consider a formal multipath $E = M\prod_{i=1}^{n} G_i^{x_i}$. Call E *feasible* if there are numbers r_i such that $M\prod_{i=1}^{n} G_i^{r_i}$ is equivalent to some $\mathcal{G}$-multipath. As we show below, the set of feasible extended multipaths is finite; hence, in particular, the set of scalable representations of $\mathcal{G}$-multipaths is also finite. Intuitively, this provides a method to decide δSCT, as follows:

(1) Construct the set $\mathcal{E}$ of all scalable representations of $\mathcal{G}$-multipaths.

(2) Test each one according to Theorem 4.14.

Example. Consider the following program, with the obvious size-change graphs G_1 and G_2 for calls 1 and 2 respectively.

```
f(x,y) = if x,y>1 then ¹f(x-1,x+1) + ²f(y-2,y-1) else 1
```

There are four irreducible nonempty multipaths: G_1, G_2, G_1G_2 and G_2G_1. All are cyclic. The set $\mathcal{E}$ contains, in addition, each of these four multipaths followed by the in-situ factor $\{\mathtt{x} \stackrel{-1}{\to} \mathtt{x}, \mathtt{y} \stackrel{-1}{\to} \mathtt{y}\}^x$.

Checking these multipaths according to Theorem 4.14, we find that δSCT is satisfied. □

We next derive some combinatorial bounds for irreducible multipaths, leading to the desired finite search space for the decision problem. For simplicity in calculations, we assume that all functions in the subject program have the same number k of parameters. We will use m for the number of functions, and Δ for the maximum absolute value of any size-change label in $\mathcal{G}$ (we may assume that both positive and negative labels are present, for otherwise the problem is at most as hard as plain SCT). We omit the proofs of the easier claims.

LEMMA 4.15. *The number of different unlabeled fan-in-free graphs with k parameters on both sides is $(k+1)^k$.*

LEMMA 4.16. *For any given in-situ graph $G : f \to f$, the set of (labeled) graphs $\frac{G}{M}$ (with M ranging over all multipaths with source f) has at most $(k+1)^k$ elements.*

LEMMA 4.17. *Every multipath of length greater than $L = (k+1)^k \cdot m$ is reducible.*

PROOF. For a multipath longer than L, it suffices to consider its suffix of length L. So let $M = G_1G_2 \ldots G_L$. Let f be an ACG node visited most often in this

multipath. It is visited at least $(k+1)^k+1$ times. Consider *unlabeled* graphs $K_i \simeq G_i;\cdots;G_L$, for all i such that G_i has source function f. By the pigeonhole principle, there are $i<j$ such that $K_i = K_j$, implying (by Lemma 4.8) $\overline{G_i \dots G_{j-1}} \prec \overline{G_j \dots G_L}$. □

LEMMA 4.18. *Let the* extended multipath $MG_1^{x_1}\dots G_n^{x_n}$ *be normal. Then*
(i) *the absolute values of size-change labels in all* G_i *are bounded by* $L\Delta$, *and*
(ii) $n \le (2L\Delta+2)^k$.

PROOF. Claim (i) follows from Definition 4.12 (Part two) and Lemma 4.17. Claim (ii) is obtained by noting that every graph G_i is specified by k arc labels from $\{-L\Delta,\dots,+L\Delta,\infty\}$. □

The last two lemmas suffice for bounding the quantity and length of multipaths in $\mathcal{E}$, but we do not dwell on this point because the bounds obtained are unsatisfactory. Tighter results can be obtained by more elaborate analysis, using two sources of information. First, bounds from the theory of Integer Linear Programming.

We use the following lemma (adapted from [Papadimitriou 1981]).

LEMMA 4.19. *Let* $A_{k\times n}$ *be a matrix with integer entries of absolute value at most* a. *Then if* $Ax \ge 0; x > 0$ *has an integer solution, it also has one satisfying* $\sum_{i=1}^n x_i \le 2(n+k)(ka)^{2k+1}$.

Applying this to the ILP problem (2), we obtain

LEMMA 4.20. *Assume that for some normal extended multipath* $MG_1^{x_1}\dots G_n^{x_n}$ *there is a solution to the Integer Linear Programming problem* (2). *Then there is such a solution that satisfies* $\sum_{i=1}^n x_i \le 2^{2k+1}(m\Delta)^{3k+1}(k+1)^{3k^2+3k+1}$.

PROOF. Problem (2) has the form $Ax \ge 0; x > 0$ with A of dimension $k\times(n+1)$. From Lemma 4.18, the absolute value of entries is bounded by $L\Delta$ and for their number we have the bound $n \le (2L\Delta+2)^k$. The bound of Lemma 4.19 becomes

$$\begin{aligned}
2(n+1+k)(kL\Delta)^{2k+1} &\le 2((2L\Delta+2)^k+1+k)(kL\Delta)^{2k+1} \\
&\le 2(4L\Delta)^k(kL\Delta)^{2k+1} \\
&= 2^{2k+1}k^{2k+1}(m\Delta)^{3k+1}(k+1)^{3k^2+k} \\
&< 2^{2k+1}(m\Delta)^{3k+1}(k+1)^{3k^2+3k+1}.
\end{aligned}$$

□

To make use of this result we need a more precise relation between $\mathcal{G}$-multipaths and normal extended multipaths.

LEMMA 4.21. *For every cyclic* $\mathcal{G}$*-multipath* $M : f \rightsquigarrow f$, *a formal multipath* $E = M'\prod_{i=1}^n G_i^{x_i}$ *exists such that:*
(i) *There are positive numbers* $r_1,\dots,r_n$ *such that* $E(1,r_1,\dots,r_n) \equiv M$.
(ii) *There are nonnegative numbers* $b_0,\dots,b_n$ *such that for all sequences* $(l_0,\dots,l_n)$ *with* $l_0 \ge b_0,\dots,l_n \ge b_n$, $E(l_0,l_1,\dots,l_n)$ *is equivalent to some cyclic* $\mathcal{G}$*-multipath of length bounded by* $L\sum_{i=0}^n l_i$, *where* $L = (k+1)^k\cdot m$. *The numbers* b_i *satisfy:* $\sum_i b_i \le (k+1)^k\cdot m$.

As the proof is lengthy and would distract us from the main theme, it is given in appendix A.

We are now ready to prove a complexity classification of the δSCT problem without fan-in. This is achieved (perhaps surprisingly) via an algorithm that does not use formal multipaths at all.

THEOREM 4.22. *The δSCT problem for fan-in-free graphs is in PSPACE.*

PROOF. Let ACG $\mathcal{G}$ with fan-in free size-change graphs be given. The Basic Theorem shows that deciding δSCT is equivalent to determining if there is a cyclic multipath that testifies against size-change termination by having no descending in-situ arc. Assume that such a multipath M does exist. By the last lemma it has a scalable representation $E = M' \prod_{i=1}^{n} G_i^{r_i}$, such that the corresponding ILP problem (2) has a solution. Now, by Lemma 4.20, there is a solution satisfying:

$$\sum_{i=0}^{n} x_i \leq 2^{2k+1}(m\Delta)^{3k+1}(k+1)^{3k^2+3k+1}.$$

Does this solution yield a $\mathcal{G}$-multipath? Not necessarily, because of the lower bounds b_i in Part (ii) of Lemma 4.21. However, as all exponents are positive, we multiply by $\sum_i b_i$ and obtain one that does. The sum of the exponents thus becomes

$$(\sum_{i=0}^{n} b_i)(\sum_{i=1}^{n} x_i) \leq (k+1)^k \cdot m \cdot 2^{2k+1}(m\Delta)^{3k+1}(k+1)^{3k^2+3k+1}.$$

Lemma 4.21 further ensures that we can convert the extended multipath obtained to a $\mathcal{G}$-multipath whose length is at most L times the sum of the exponents, so, to conclude, we have shown that if a counterexample to termination (according to the Basic Theorem) exists, there is one of length at most

$$B(\mathcal{G}) = (k+1)^{2k} \cdot m^2 \cdot 2^{2k+1}(m\Delta)^{3k+1}(k+1)^{3k^2+3k+1}.$$

A simple nondeterministic algorithm for *the complement of* δSCT is to first compute this bound, then nondeterministically follow a multipath M of at most such length and verify that it is a counterexample to the condition in Theorem 4.4.

The space usage of this algorithm can be optimized by observing that it suffices to keep in memory $\overline{M}$ and the current length of M, rather than the complete multipath. The length requires at most $\lceil \log B(\mathcal{G}) \rceil$ bits of storage, while for $\overline{M}$ we need at most $k(\lceil \log(\Delta B(\mathcal{G})) + 1 \rceil)$ by the familiar argument on the absolute value of labels. Both bounds are polynomial in k, $\log m$ and $\log \Delta$. Thus the algorithm places the problem in NPSPACE, which by Savitch's theorem equals PSPACE. □

We now observe that the PSPACE-hardness proof for SCT in [Lee et al. 2001] uses fan-in-free graphs. Which gives

COROLLARY 4.23. *The δSCT problem for fan-in-free graphs is PSPACE-complete.*

Observe in passing that the above observation means that for SCT, the presence of fan-in does not affect the complexity class of the problem, quite unlike the case for δSCT.

The algorithm suggested by the proof of Theorem 4.22 is not likely to be a good choice for a practical application of δSCT: the use of Savitch's theorem results in regenerating parts of the solution exponentially many times, in order to keep the space bound polynomial. A more useful approach is probably to allow for more space in order to avoid recomputation, i.e., revert to a closure computation as in previous termination analyzers. We now describe this approach in more detail.

Algorithm 4.1. Closure-based algorithm for δSCT over fan-in free graphs.

This algorithm constructs the set $\mathcal{S}$ of all δSCT graphs $\overline{M}$ for some $\mathcal{G}$-multipath M of length bounded by $B(\mathcal{G})$. If an idempotent graph with no in-situ descent is not encountered, δSCT is satisfied.

The algorithm labels each graph G by the length of the corresponding multipath, in order to make use of the length bound. Thus, it actually maintains a set $\mathcal{S}'$ of graphs with lengths:

(1) Initialize $\mathcal{S}'$ to include $(G, 1)$ for every $G \in \mathcal{G}$.
(2) For any $(G, i) : \mathtt{f} \to \mathtt{g}$ and $(H, j) : \mathtt{g} \to \mathtt{h}$ in $\mathcal{S}'$, if $i + j \leq B(\mathcal{G})$, and no pair $(G; H, k)$ exists yet in $\mathcal{S}'$, include $(G; H, i + j)$ in $\mathcal{S}'$. If $(G; H, k)$ is in $\mathcal{S}$, just replace the k with $i + j$ if $i + j$ is smaller.
(3) The above step should be repeated until no more changes can be made to $\mathcal{S}'$.

□

4.5 Some further complexity-theoretic observations

4.5.1 *Fixed-Parameter Complexity.* An interesting property that Algorithm 4.1 has in common with the closure-based SCT algorithm [Lee et al. 2001] can be observed by noticing that the absolute values of labels on the arcs of graphs in $\mathcal{S}$ are bounded by the $B(\mathcal{G}) \cdot \Delta$. As there are at most $(k+1)^k$ different (unlabeled) graphs, we obtain

$$|\mathcal{S}| < (k+1)^k \cdot (B(\mathcal{G}) \cdot \Delta)^k;$$

when the number of parameters k is constant and Δ is at most polynomial in m, $|\mathcal{S}|$ is polynomial in m. It is not hard to verify that this also means that the running time of the algorithm is polynomial. Similarly, the closure-based SCT algorithm is polynomial-time if k is fixed. There is a difference however between the two situations inasmuch as the SCT algorithm is fixed-parameter tractable [Downey and Fellows 1995] for the parameter k, while Algorithm 4.1 is not (due to the m^{k^2} factor in the time bound).

4.5.2 *Graphs with fan-in.* The fact that fan-in freedom is necessary for our algorithm stands in tension with the natural tendency to be greedy while constructing size-change graphs and include as many arcs as possible, since it is not yet known which will be pertinent to termination.

Let δSCT^* be the set of δSCT instances $\mathcal{G}$ such that after deleting some (possibly empty) set of size-change arcs from $\mathcal{G}$, a fan-in free instance remains that satisfies δSCT. We call such instances "positive instances without material fan-in." As an easy corollary of the last theorem, δSCT^* is also in PSPACE: it suffices to perform a brute-force search over sets of arcs for deletion. Obviously, this approach is not

practically appealing. A more practical one would be to make use, when possible, of sound criteria for deleting arcs, as done in [Ben-Amram and Lee 2007] with respect to SCT.

5. δSCT AND OTHER TERMINATION ANALYSES

This section discusses interesting connections between this work and some other ideas from the field of termination analysis (besides SCT).

5.1 Linear ranking functions

The essential characteristic of SCT analysis that sets it apart from much other work on termination (see for example the survey [Schreye and Decorte 1994]) is that we are not trying to synthesize a well-founded order on program states, or explicitly find a "ranking function" that induces such an order. Yet, our problem has interesting connections to such techniques. Sohn and Van Gelder [1991] present a method that aims to find, for every function $f(x_1, \ldots, x_n)$, a linear combination $a_1x_1 + \cdots + a_nx_n$, with $a_i \geq 0$, whose value is guaranteed to decrease in every call-cycle from f to f (to simplify notation, we treat the arguments x_i as natural numbers). To find the coefficient vector $\vec{a}$, their algorithm solves a linear program. In the simple case of a single function with in-situ graphs (Section 4.3), the linear program is exactly the same that we presented. Indeed, it is not hard to verify (using the duality theorem of linear programming) that the linear combination sought by Sohn and Van Gelder's method exists *if and only if* δSCT is satisfied. For example, the example in Figure 3 on Page 5 has the linear ranking function $2x + 3y$.

However, with slightly more complicated instances of δSCT, it becomes possible that no single linear combination can serve as a ranking function. An example is the following program (which even satisfies SCT):

```
f(x,y) = if xy>0 then f(x-1,x-1) + f(y-1,y-1) else 1
```

5.2 Non-linear ranking functions

While the above example does not have a linear ranking function, it can be fitted with the simple non-linear ranking functions $\max(x, y)$. It was proved only recently by Lee that *every* program that terminates by SCT can be provided with a ranking function constructed from the program's variables plus certain constants and the operators min, max and lexicographic tupling [Lee 2006]. The function has the property that for every call $f \stackrel{c}{\to} g$, the size-change graph G_c implies a decrease in the ranking function's value. One might speculate about an extension of this result to δSCT programs, say using linear combinations in addition to the above operators. This may be the case for fan-in free graphs (we do not know), but is certainly not so for δSCT in general. Here's the proof: the ranking functions of a class of this style (even with some other operators) can be recursively enumerated, and checked against the size-change graphs. If every program that terminates by δSCT had such a ranking function, we would deduce that $\delta\text{SCT} \in \text{RE}$. However, the reduction in Section 3 implies the converse, because we reduced from *non-halting* to δSCT.

5.3 Multiple ranking functions

Instead of choosing a single ranking function that should descend over every loop, one can choose a finite set of ranking functions such that every possible loop (cycle in the control-flow graph) decreases one of them (the rest may even increase). That this suffices for termination can be proved by Ramsey's theorem, as in [Dershowitz et al. 2001; Podelski and Rybalchenko 2004]. Such a "disjunctive ranking-function construction" is just what Algorithm 4.1 does—except that it focuses on cycles that yield idempotent size-change graphs.

Codish, Lagoon and Stuckey [2005] show that in a positive SCT instance, it is possible to deduce a linear ranking function for every cycle, not only those associated with idempotent graphs. The same argument applies to δSCT. It follows that when our algorithm succeeds in verifying δSCT, it can also produce a set of linear ranking functions to cover all possible cycles. Thus, both for SCT and δSCT, the local or "disjunctive" approach simplifies the form of ranking functions necessary in a ranking-function based termination proof. Put otherwise, we learn that approaches to termination that rely on disjunctions of ranking functions (e.g., [Cook et al. 2005]) are justified, at least for δSCT instances, in restricting the functions to linear.

5.4 Handling non-well-founded data types

One difficulty in applying SCT analysis to programs in common imperative languages is that loop variables in these programs are frequently of type *integer*, not a well-founded domain. Instead of descent towards the "bottom" of the data type, loop termination follows from either descent or ascent towards a bound determined by the loop conditions and perhaps other variables. Colón and Sipma [2002] illustrate an approach for handling such programs. They use linear-programming techniques both for deducing *bounded expressions* and for determining which of the bounded expressions can be used as a ranking function.

As noted in [Colón and Sipma 2002], the pertinent bounded expressions are often single variables. This indicates that SCT may furnish the termination proof. In fact, of three example programs given in [Colón and Sipma 2002], the first two are SCT instances (once bounded variables have been determined); the third ("a program derived from McCarthy's 91 function") satisfied δSCT.

Avery [2006] presents another, somewhat more involved, combination of bound analysis and SCT, that also captures the two examples mentioned above. In the first stage of his algorithm, the analysis determines bounded linear expressions—not necessarily single variables; this stage does not rely on size-change graphs. In a second stage, the algorithm attempts to prove that for every cyclic SCT graph, one of the bounded expressions descends whenever the correponding cycle in the program is completed. This approach would work with δSCT (in the fan-in free case) as well.[4]

5.5 Partial solutions

Prior to this work, Anderson and Khoo [2003] had proposed to extend the SCT algorithm in a way that allows a more expressive abstraction (general linear con-

[4] Thanks to James Avery for pointing this out.

straints) to be exploited. Since δSCT graphs are more restricted, their algorithm cannot be judged only on its coverage of the δSCT problem, but let us do that for the sake of relating their approach to the purpose of this paper. In this setting, the effect of their method can be roughly described as follows: instead of eagerly replacing every δSCT graph by its SCT approximation, the algorithm composes the given graphs along computation paths, thus obtaining more precise SCT graphs for the CFG cycles. Ultimately, all cycles have to be shown terminating via the SCT abstraction. For a simple example, consider a case where a size-change graph with arc $\mathtt{x} \stackrel{+1}{\to} \mathtt{y}$ is followed by one with $\mathtt{y} \stackrel{-2}{\to} \mathtt{z}$. The algorithm will deduce the SCT arc $\mathtt{x} \stackrel{\downarrow}{\to} \mathtt{z}$. Thus the method is stronger than naïve transformation to SCT, and yet it is not powerful enough to handle δSCT instances where a subtler reasoning is necessary, such as the examples on Pages 5 and 10.

6. CONCLUSION

We investigated the decidability of δSCT, a criterion for inferring program termination from a control-flow graph annotated with size-change information. This criterion seems to be a natural one and at least implicitly indicated in much previous work, though apparently avoided for lack of a decision procedure. We have shown the general case to be undecidable, but provided an algorithm for an important restricted case, namely fan-in free instances.

While we have settled the theoretical question regarding decidability and complexity class of δSCT, the positive result seems far from practical application. Certainly, this paper does not provide algorithms that appear to be anywhere near practical efficiency. There are, however, reasons for optimism:

—Termination of programs is, in general, undecidable, which could be interpreted as a “no entry” sign. Nevertheless, the field of termination checking is thriving, as witnessed by the existence of a workshop devoted to the subject—WST (whose 9th meeting is already scheduled as these lines are being written). The field is well represented in symposia such as TACAS, SAS, RTA and CAV, to name just a few. The work presented in these conferences shows that termination checking (in various forms) is gaining industrial strength (consider, for example, the experiences reported in [Cook et al. 2006; Manolios and Vroon 2006]).

—The paper that presented SCT [Lee et al. 2001] was quite theoretical and only presented a complexity result—PSPACE completeness of the decision problem. Nonetheless, as we soon found out, SCT decision had been implicit in the Prolog termination provers [Lindenstrauss and Sagiv 1997b; Codish and Taboch 1999], which are demonstration tools, but do show practicality of the algorithms to some degree. Recently, Manolios and Vroon [Manolios and Vroon 2006] applied SCT analysis to a vast benchmark suite of over 10,000 functions in the ACL2 functional language, representing all kinds of applications of program verification.

The restriction to fan-in free instances may also appear to be a drawback, noting that the proposal in Section 4.5.2 of trying fan-in free subgraphs by brute force is rather unattractive. However, the program sample we studied in [Ben-Amram and Lee 2007] turned out to exhibit a tiny number of occurrences of fan-in once certain safe “clean-up” steps were taken. Certainly, such a study cannot be conclusive;

but it points out that fan-in may not be such a big hindrance, even in Prolog programs where our intuition was that fan-in would be likely to occur as a result of unifications.

On the theoretical side, this work suggests the challenge of further exploring the range of (abstract) termination analysis problems that can be decided precisely and their computational complexity.

We pose two specific open problems:

(1) Is there an FPT algorithm (say, in (k, Δ)) for δSCT?

(2) Can all programs that satisfy fan-in free δSCT be fitted with a ranking function of some (relatively) simple form, deducible from the size-change graphs?

Acknowledgments

I am obliged to Andreas Podelski and Chin Soon Lee for proposing the problem. Many stimulating discussions with Chin Soon Lee are also gratefully acknowledged, as are his invaluable comments on earlier versions of the manuscript. The QuickSort example originated with Chin Soon Lee and was given its current form by one of the referees. The anonymous referees have done a thorough job and their contributions are much appreciated.

A. APPENDIX: PROOF OF LEMMA 4.21

In this appendix we prove Lemma 4.21. As in Section 4, we assume an ACG of m functions and k parameters in each, and all size-change graphs fan-in free. Let us recite the claims to be proved:

For every cyclic $\mathcal{G}$-multipath M, beginning and ending at any function f, a normal extended multipath $E = M_0 \prod_{i=1}^{n} G_i^{r_i}$ exists such that:
(i) $E \equiv M$ (we say that E represents M).
(ii) There are nonnegative numbers $b_0, \ldots, b_n$ such that for all sequences $l_0 \geq b_0, \ldots, l_n \geq b_n$, $(M_0)^{l_0} \prod_{i=1}^{n} G_i^{l_i}$ is equivalent to some cyclic $\mathcal{G}$-multipath of length bounded by $L \sum_{i=0}^{n} l_i$. The numbers b_i satisfy: $\sum_i b_i \leq (k+1)^k \cdot m$.

We begin by describing the process of obtaining E from M. Briefly, we start with M and apply a series of reductions (Definition 4.9), collecting powers of in-situ graphs at the end. Since the set of in-situ graphs changes along the way, it is inconvenient to use number indices ($G_i^{x_i}$). Instead, we use the size-change graph itself to index the exponent: G^{x_G}. This emphasizes that the order of the G's is immaterial. We use the letter S for the set of in-situ graphs.

Definition A.1. Let $E_1 = M \prod_{G \in S} G^{l_G}$, $E_2 = M' \prod_{G \in S'} G^{r_G}$ be extended multipaths. We write $E_1 \Rightarrow_B E_2$ if: B is a nonempty, irreducible multipath; $M = ABC$ with $\overline{B} \prec \overline{C}$; $M' = AC$; $S' = S \cup \{H\}$ with $H = \frac{\lfloor \overline{B} \rfloor}{C}$; and $r_G = l_G$ for all $G \in S'$ except that $r_H = l_H + 1$ ($r_H = 1$ if $H \notin S$).

We write $E_1 \Rightarrow E_2$ if $E_1 \Rightarrow_B E_2$ for some B.

Let M be a multipath beginning and ending at any function f. By repeatedly reducing M as long as a redex exists, always picking an irreducible redex (e.g., a shortest one), it follows quite easily that $M \Rightarrow^* E$ such that E is normal, proving Claim (i). It is also easy to see that the "stem" M_0 of E is cyclic if M is. We proceed to prove Claim (ii) with the help of some auxiliary definitions.

Definition A.2. A sequence $(M_0, B_1, B_2, \ldots, B_t)$ of irreducible multipaths is *consistent* if there exists a multipath E_t and extended multipaths $E_{t-1}, \ldots, E_0$ such that

$$E_t \Rightarrow_{B_t} E_{t-1} \Rightarrow_{B_{t-1}} \cdots \Rightarrow_{B_1} E_0 = M_0 \prod_{G \in S_0} G^{r_G}.$$

If the above condition holds, we say that the reduction sequence $E_t \Rightarrow \cdots \Rightarrow E_0$ *implements* $(M_0, B_1, B_2, \ldots, B_t)$. There can be different implementations of the same sequence!

Definition A.3. For any multipath M, define $\overline{\overline{M}}$ to be the graph $\overline{M}$ with arc labels removed. For multipaths M and N, let $N \lhd M$ represent the relation "N is a suffix of M." We define $\sigma(M) \stackrel{def}{=} \{\overline{\overline{N}} \mid N \lhd M\}$. If E is an extended multipath, $E = M \prod G_i^{x_i}$, $\sigma(E)$ is defined to be $\sigma(M)$.

In a reduction $ABC \Rightarrow AC\frac{\lfloor \overline{B} \rfloor}{C}$, we have $\sigma(ABC) \supseteq \sigma(AC)$. For if N is a suffix of C, then it is a suffix both of AC and of ABC; while if $N = A'C$, with $A' \lhd A$, then $\overline{N} \simeq \overline{A'BC}$ by Lemma 4.8. Thus, $E_1 \Rightarrow E_2$ implies $\sigma(E_1) \supseteq \sigma(E_2)$.

Definition A.4. Let $(M_0, B_1, B_2, \ldots, B_t)$ be consistent, and implemented by $E_t \Rightarrow \cdots \Rightarrow E_0$. Let the subsequence $K = (i_1 < \cdots < i_s)$ of *critical indices* be defined by: $i \in K$ iff $\sigma(E_i) \supset \sigma(E_{i-1})$.

LEMMA A.5. *Let $(M_0, B_1, B_2, \ldots, B_t)$ be consistent, and implemented by $E_t \Rightarrow \cdots \Rightarrow E_0$. Let $K = (i_1 < \cdots < i_s)$ be the critical indices. Then $(M_0, B_{i_1}, \ldots, B_{i_s})$ is consistent and has an implementation $E'_s \Rightarrow \cdots \Rightarrow E'_1 \Rightarrow E'_0$. We further have:*
(i) $\sigma(E'_j) = \sigma(E_{i_j})$ *for all* j,
(ii) $|E'_s| \leq (k+1)^k \cdot m \cdot s$.

PROOF. We use induction on t. For $t = 0$, K is empty and the claim is trivial. Now, assume correctness for $t-1$. There are now two cases.
Case 1: t is not critical. There is nothing to prove. But note that $\sigma(E_t) = \sigma(E_{t-1})$.
Case 2: $t = i_s$ is critical. I.e., $\sigma(E_t) \supset \sigma(E_{t-1})$. Using the induction hypothesis and the definition of K, we have

$$\sigma(E_{t-1}) = \sigma(E_{t-2}) = \cdots = \sigma(E_{i_{s-1}}) = \sigma(E'_{s-1}). \qquad (3)$$

Let us write the reduction $E_t \Rightarrow_{B_t} E_{t-1}$ explicitly as

$$AB_tC \prod_{G \in S} G^{l_G} \Rightarrow ACH \prod_{G \in S} G^{l_G} \equiv AC \prod_{G \in S'} G^{r_G},$$

with $H = \frac{\lfloor \overline{B_t} \rfloor}{C}$. The conditions for reduction require that $\overline{B_t} \prec \overline{C}$, or equivalently, $\overline{B_t}; \overline{C} \simeq \overline{C}$. By (3), $\overline{\overline{C}} \in \sigma(E_{i_{s-1}}) = \sigma(E'_{s-1})$. I.e., $\overline{\overline{C}} = \overline{\overline{N}}$ with $N \lhd E'_{s-1}$. Thus $\overline{B_t}; \overline{N} \simeq \overline{N}$, and we can insert B_t into E'_{s-1} just before N, obtaining E'_s such that

$$E'_s \Rightarrow_{B_t} E'_{s-1} \frac{\lfloor \overline{B_t} \rfloor}{N} = E'_{s-1} H.$$

Moreover, by the observations following Definition A.3, we see that

$$\sigma(E'_s) = \sigma(E'_{s-1}) \cup \{\overline{\overline{BN}} \mid B \lhd B_t\} = \sigma(E_{t-1}) \cup \{\overline{\overline{BC}} \mid B \lhd B_t\} = \sigma(E_t).$$

By adding H to the end of $E'_{s-2}, \dots, E'_0$ we complete the proof of the induction claim.

The bound on $|E'_s|$ follows from M_0 and the B_i being irreducible. □

LEMMA A.6. *Let* $(M_0, B_1, B_2, \dots, B_t)$ *be consistent, and implemented by* $E_t \Rightarrow \dots \Rightarrow E_0$. *Let* $K = (i_1 < \dots < i_s)$ *be the critical indices. Then* $|K| < (k+1)^k \cdot m$.

PROOF. A simple counting argument, noting that $\sigma(E_i)$ is a set of unlabeled, fan-in free size-change graphs, with varying source functions but a fixed target, that has to grow at each critical index i. □

We are now ready to prove Claim (ii) of Lemma 4.21, as repeated at the top of this appendix. Recall that $E = M_0 \prod_{G\in S} G^{r_G}$ has been obtained from M by a reduction sequence. The last two lemmas allows us to construct a multipath M' that, via a sequence of less than $(k+1)^k \cdot m$ steps, reduces to $E'_0 = M_0 \prod_{G\in S'} G^{b_G}$ where $S' \subseteq S$ (because the redexes are a subset of the original ones). Observe that $\sum_{G\in S'} b_G$ is just the length of the (smaller) reduction sequence and therefore smaller than $(k+1)^k \cdot m$; add $b_0 = 1$ and we have $\sum b_i \le (k+1)^k \cdot m$. Also, $\sigma(M') = \sigma(M) = \sigma(E)$; moreover, $\overline{M} \simeq \overline{M'} \simeq \overline{M_0}$.

Finally, consider any set $L = \{l_G \mid G \in S\}$ with $l_G \ge b_G$ for all G (if $G \in S \setminus S'$, we set $b_G = 0$). And let $l_0 \ge 1$. We claim that $E_L = (M_0)^{l_0} \prod_{G\in S} G^{l_G}$ is equivalent to some $\mathcal{G}$-multipath of length at most $(k+1)^k \cdot m \cdot (l_0 + \sum_G l_G)$. Such a multipath can be obtained by constructing an appropriate reduction sequence in reverse: begin with the reduction sequence that reduces M' to E'_0. Note that E'_0 is embedded inside E_L by the assumption on the l_G values. Extend E'_0 with additional copies of M_0 at the beginning, and graphs G at the end, as necessary, to obtain E_L; by carrying the same additions all along the reduction sequence, we obtain a reduction sequence that begins with $(M_0)^{l_0-1} M' \prod_{G\in S} G^{l_G - b_G}$ and ends with E_L. It rests to add reduction steps (in reverse) that generate the remaining in-situ graphs (i.e., $G \in S$ where $l_G - b_G > 0$). Let G be one such graph. As G appears in our original E, we know that the reduction sequence $M \Rightarrow^* E$ includes a step $E_i \Rightarrow_{B_i} E_{i-1}$ with $G = \overline{B_i}$. Thus $E_i = AB_iC \prod_{G\in S_i} G^{r_G^{(i)}}$ with $\overline{B}_i \prec \overline{C}$. By definition, $\overline{\overline{C}} \in \sigma(E_i) \subseteq \sigma(E) = \sigma(M')$, i.e., $\overline{\overline{C}} = \overline{\overline{N}}$ with $N \lhd M'$. Hence it is possible to create a reverse reduction step that removes G from the trailing set and inserts B_i in M' without affecting $\sigma(M')$ (therefore, also $\overline{\overline{M'}}$ is unaffected). This can be repeated until all the trailing in-situ graphs are removed and we obtain a valid multipath $(M_0)^{l_0-1} M''$ that reduces to E_L. This multipath is cyclic because $\overline{\overline{M_0}} = \overline{\overline{M''}} = \overline{\overline{M'}}$ is idempotent. The bound of the length of the multipath is immediate from the irreducibility of M_0 and the redexes.

REFERENCES

ANDERSON, H. AND KHOO, S.-C. 2003. Affine-based size-change termination. In *Proceedings of the First Asian Symposium on Programming Languages and Systems, APLAS 2003, Beijing, China*, A. Ohori, Ed. Lecture Notes in Computer Science, vol. 2895. Springer, 122–140.

APT, K. R. AND PEDRESCHI, D. 1994. Modular termination proofs for logic and pure prolog programs. In *Advances in Logic Programming Theory*. Oxford University Press, 183–229.

AVERY, J. 2006. Size-change termination and bound analysis. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006*, M. Hagiya and P. Wadler, Eds. Lecture Notes in Computer Science, vol. 3945. Springer.

BEN-AMRAM, A. M. AND LEE, C. S. 2007. Size-change analysis in polynomial time. *ACM Transactions on Programming Languages and Systems 29,* 1.

BRODSKY, A. AND SAGIV, Y. 1991. Inference of inequality constraints in logic programs. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), 1991.* ACM Press, 227–240.

BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *International Symposium on Logic Programming.* MIT Press, 320–336.

CODISH, M., LAGOON, V., AND STUCKEY, P. J. 2005. Testing for termination with monotonicity constraints. In *Logic Programming, 21st International Conference, ICLP 2005*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 326–340.

CODISH, M. AND TABOCH, C. 1999. A semantic basis for termination analysis of logic programs. *The Journal of Logic Programming 41,* 1, 103–123. preliminary (conference) version in LNCS 1298 (1997).

COLÓN, M. AND SIPMA, H. 2002. Practical methods for proving program termination. In *14th International Conference on Computer Aided Verification (CAV).* Lecture Notes in Computer Science, vol. 2404. Springer, 442–454.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2005. Abstraction refinement for termination. In *Static Analysis, Proceedings of the 12th International Symposium, SAS '05, London, UK, Sep 7–9, 2005.* 87–101.

COOK, B., PODELSKI, A., AND RYBALCHENKO, A. 2006. Terminator: Beyond safety. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 2006*, T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 415–418.

DERSHOWITZ, N., LINDENSTRAUSS, N., SAGIV, Y., AND SEREBRENIK, A. 2001. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing 12,* 1–2, 117–156.

DOWNEY, R. G. AND FELLOWS, M. R. 1995. Fixed-parameter tractability and completeness. I. basic results. *SIAM J. Comput. 24,* 4 (Aug.), 873–921.

GRAHAM, R. L. 1981. *Rudiments of Ramsey Theory.* American Mathematical Society.

JONES, N. D. 1988. Automatic program specialization: A re-examination from basic principles. In *Partial evaluation and Mixed Computation*, D. Bjørner, A. P. Ershov, and N. D. Jones, Eds. 225–282.

JONES, N. D. 1997. *Computability and Complexity From a Programming Perspective.* Foundations of Computing Series. MIT Press.

JONES, N. D. AND BOHR, N. 2004. Termination analysis of the untyped lambda calculus. In *Proceedings of the 15th International Conf. on Rewriting Techniques and Applications, RTA'04.* Lecture Notes in Computer Science, vol. 3091. Springer, 1–23.

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice Hall.

JONES, N. D., LANDWEBER, L. H., AND LIEN, Y. E. 1977. Complexity of some problems in Petri nets. *Theoretical Computer Science 4,* 3 (June), 277–299.

LEE, C. S. 2002. Program termination analysis and termination of offline partial evaluation. Ph.D. thesis, University of Western Australia.

LEE, C. S. 2006. ranking functions for size-change termination. submitted for publication.

LEE, C. S., JONES, N. D., AND BEN-AMRAM, A. M. 2001. The size-change principle for program termination. In *Proceedings of the Twenty-Eigth ACM Symposium on Principles of Programming Languages, January 2001.* Vol. 28. ACM press, 81–92.

LINDENSTRAUSS, N. AND SAGIV, Y. 1997a. Automatic termination analysis of logic programs (with detailed experimental results). `http://www.cs.huji.ac.il/~naomil/`.

LINDENSTRAUSS, N. AND SAGIV, Y. 1997b. Automatic termination analysis of Prolog programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed. MIT Press, Leuven, Belgium, 64–77.

MANOLIOS, P. AND VROON, D. 2006. Termination analysis with calling context graphs. In *Proceedings, Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA*. LNCS, vol. 4144. Springer-Verlag, 401–414.

NAISH, L. 1985. Automating control for logic programs. *Journal of Logic Programming 2,* 3 (Oct.), 167–183.

PAPADIMITRIOU, C. H. 1981. On the complexity of integer programming. *Journal of the ACM 28,* 4, 765–768.

PLÜMER, L. 1990. *Termination Proofs for Logic Programs.* Lecture Notes in Artificial Intelligence, vol. 446. Springer-Verlag.

PODELSKI, A. AND RYBALCHENKO, A. 2004. Transition invariants. In *LICS'04: Logic in Computer Science*, H. Ganzinger, Ed. IEEE Computer Society, 32–41.

RAMSEY, F. P. 1930. On a problem of formal logic. In *Proceedings of the London Mathematical Society.* The London Mathematical Society, 264–286.

SAGIV, Y. 1991. A termination test for logic programs. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA*, V. Saraswat and K. Ueda, Eds. MIT Press, 518–532.

SCHREYE, D. D. AND DECORTE, S. 1994. Termination of logic programs: The never-ending story. *Journal of Logic Programming 19-20*, 199–260.

SOHN, K. AND VAN GELDER, A. 1991. Termination detection in logic programs using argument sizes (extended abstract). In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SOGART Symposium on Principles of Database Systems (PODS), May 1991, Denver, Colorado.* ACM Press, 216–226.

THIEMANN, R. AND GIESL, J. 2005. The size-change principle and dependency pairs for termination of term rewriting. *Applicable Algebra in Engineering, Communication and Computing 16,* 4 (Sept.), 229–270.

ULLMAN AND GELDER, V. 1988. Efficient tests for top-down termination of logical rules. *Journal of the ACM 35,* 2, 345–373.