Program Termination Analysis in Polynomial Time

AMIR M. BEN-AMRAM
The Academic College of Tel-Aviv-Yaffo, Israel
and
CHIN SOON LEE
Max-Planck-Institut für Informatik, Saarbrücken, Germany

A *size-change termination algorithm* takes as input abstract information about a program in the form of *size-change graphs* and uses it to determine whether any infinite computation would imply that some data decrease in size infinitely. Since such an infinite descent is presumed impossible, this proves program termination. The property of the graphs that implies program termination is called SCT. There are many examples of practical programs whose termination can be verified by creating size-change graphs and testing them for SCT.

The size-change graph abstraction is a useful one because the graphs often carry sufficient information to deduce termination, and at the same time are simple enough to be analyzed automatically. However, there is a trade-off between the completeness and efficiency of this analysis, and complete algorithms in the literature can easily be pushed to an exponential combinatorial search by certain patterns in the graph structures.

We therefore propose a novel algorithm to detect common forms of parameter-descent behavior efficiently. Specifically, we target lexicographic descent, multiset descent and min- and max-descent. Our algorithm makes it possible to verify practical instances of SCT while guarding against unwarranted combinatorial search. It has worst-case time complexity cubic in the input size, and its effectiveness is demonstrated empirically using a test-suite of over 90 programs.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Algorithms, Design, Performance

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

1. INTRODUCTION

The work presented in this paper deals, generally speaking, with algorithms for verifying program termination. The termination problem is a cornerstone of program verification; this is obvious and probably sufficient to motivate its research. Work on program termination also grew out of research on evaluation strategies in Logic Programming, since a program may be terminating under one evaluation strategy and non-terminating under another. Finally, a different area which sparked research on termination questions is program transformation. For example, consider off-line partial evaluation (specialization). The termination of program specialization is not necessarily guaranteed even if one is willing to assume that the subject program always terminates.

How does one verify program termination? The classic technique [Turing 1948; Floyd 1967] is that of *ranking* program states, i.e., the prover has to come up with a function that maps program states into a *well-founded set*, such that any program transition (or, at least, any cycle in the flow-chart) causes this "ranking" to decrease. Well-foundedness means that there can be no infinite decreasing chain, hence the program must terminate[1].

Various techniques of constructing such functions have been proposed; however, much of this work is difficult to automate. Intuitively, it may require ingenuity to find the right ranking function. The earliest example we know of, by Turing, involves a program with nested loops. Here is a similar (well-known) example:

Example 1.1. *Ackermann's function*[2].

```
ack(m,n) = if m=0 then n+1 else
           if n=0 then ¹ack(m-1,1) else
                       ²ack(m-1,³ack(m,n-1))
```

One can prove termination of this recursive program by arguing that the pair of parameters (m, n) descends lexicographically in every recursive call. Thus one uses the set of pairs of non-negative integers, with lexicographic order, as the well-founded set. An essentially equivalent idea is to use transfinite ordinals, mapping the pair of parameters to $\omega m + n$.

There is, however, another formulation of the termination proof for this program which avoids the introduction of ranking functions and arguably intricate well-founded sets. Consider some (hypothetical) infinite chain of recursive calls. If Call 1 or Call 2 occurs infinitely often in the chain, then the values of m form a non-increasing sequence that decreases infinitely often. Otherwise, these calls occur only finitely in the chain. In this case, the values of n eventually form a strictly decreasing sequence. Both cases are impossible since m and n only assume non-negative integer values.

This is an example of the so-called *size-change principle*. The reference to *size* is due to the fact that the variables considered often hold structured data, such as a list, and it is their size that is decreasing; thus a general formulation of the principle is: Non-terminating program execution is impossible, if every infinite computation would give rise to infinite descent in the size of some data in the program. With simple functional programs, we can replace "infinite computation" by "infinite chain of function calls," by König's lemma.

In the next section, we give more precise definitions and a handful of examples to demonstrate that this principle captures a wide variety of program behaviors. Further evidence to its strength is given by examples from other sources in our bibliography. The feature of only *analyzing* changes in the program data, avoiding the *synthesis* of ranking functions, suggests that this approach is attractive

[1]Note that our objective in this paper is to prove a program terminating for *any* input.
[2]Most examples in this paper are simple functional programs, operating on lists or natural numbers. The labels on function calls are used to identify the call sites for future reference.

for automation; indeed, it has been proposed and implemented in different contexts. First, [Sagiv 1991] and [Lindenstrauss and Sagiv 1997b] introduced it in the context of Prolog; later in work that grew out of research on the termination of partial evaluators, [Lee et al. 2001] rediscovered the technique, presenting it in the context of functional programming. Subsequently, [Thiemann and Giesl 2005] adapted the size-change reasoning for term rewriting systems. These works have led to various implementations (see the Related Work section). Nonetheless, the road to industrial-strength size-change termination analysis still has obstacles, one of which is efficiency.

An important contribution of [Lee et al. 2001] is the complexity analysis of the *size-change termination problem* (SCT). Obviously termination analysis, like any "interesting" program analysis, is an open-ended goal (we are essentially approximating the undecidable). However, the above paper defines the SCT problem in terms of an abstraction of the program, called *size-change graphs* (for a precise definition, see Section 2); this structure is programming-language independent and whether termination can be deduced from it is algorithmically decidable.

The formulation allows for studying the complexity of this problem. An efficient algorithm will remove at least one obstacle from the grand project of termination analysis. The complexity result of [Lee et al. 2001] is that SCT is complete for PSPACE. This means (under a standard complexity-theoretic assumption) that the running time of a decision procedure must be exponential in the worst case. While the result does not argue against the use of SCT in practice per se, it has compelled us to consider the following question: Does the performance of the SCT decision procedures in the literature degrade gracefully with respect to the complexity of the termination argument found? What we have observed is that these decision procedures can *easily* be pushed to an exponential combinatorial search by certain patterns in the graph structures. Thus the purpose of the abstraction—to discard possibly irrelevant information so as to render the termination problem tractable—has only been partially achieved.

Instead of introducing safeguards against unwarranted combinatorial search, in this paper, we present an algorithm that decides a subset of SCT (thus conservatively approximating it) in polynomial time. Our algorithm is designed to detect the most common parameter-descent behaviors efficiently. Specifically, we target lexicographic descent, multiset descent, and min- and max-descent. We propose that this algorithm is the better choice for scalable termination analysis. Besides theoretic arguments expanded in this paper, we have also attempted an empirical investigation. We have used test suites from previous work, consisting of programs small enough to be handled by the precise (exponential-time) algorithm as well. Thus, this is not a benchmark to demonstrate the scalability of the algorithm (future work will have to fill this gap), but it shows that the algorithm is practical (i.e., its polynomial efficiency is not "only asymptotic") and that it is precise enough (in fact, there was not a single program in the test suite where our algorithm was not able to decide or refute SCT).

A precursor to the current paper is [Lee 2002], where some of the ideas in this paper already appeared. That paper should now be considered obsolete. The algorithm presented in the current paper is strictly stronger and better motivated, and much of the theory is new.

Structure of the paper. This paper is meant to be self-contained, which is why we have kept the introduction rather informal. In the next section we define size-change graphs and the SCT condition. This allows us to give examples and elaborate on issues of computational complexity. Sections 3–4 describe an efficient algorithm to recognize a subset of SCT. In Sections 5 and 6 we discuss the precision of the algorithm: what kind of termination proofs does it capture, and for what sub-problems do we know it to be complete? Finally, Section 7 presents our empirical investigation, and Section 8 concludes.

2. SIZE-CHANGE TERMINATION

This section includes the definitions of the size-change graph abstraction and the SCT condition, and recalls the main results of [Lee et al. 2001].

2.1 Preliminaries

Let L be a set of *arc labels*; *an annotated directed graph (digraph) with labels from L* is a pair $D = (V, A)$ where V is the set of nodes and $A \subseteq V \times L \times V$ is a set of labeled arcs. The property $(u, \lambda, v) \in A$ will also be written as $u \xrightarrow{\lambda} v \in D$. A *bipartite* digraph (X, Y, A) has disjoint node sets X, Y and labeled arcs $A \subseteq X \times L \times Y$. We refer to X and Y as the *source* and *target* nodes of the bipartite graph respectively. The graph D is *uniquely labeled* if for every u, v there is at most one λ such that $u \xrightarrow{\lambda} v \in D$. We write $u \to v \in D$ to indicate that $u \xrightarrow{\lambda} v \in D$ for *some* λ.

2.2 Abstraction of Program Behavior

SCT analysis deals with the following structures, representing information about the dynamic aspects of a program: a *control-flow graph* and a collection of *size-change graphs*. Both are annotated directed graphs.

The *control-flow graph* (CFG) represents accessibility between pairs of program points. In a simple functional language, program points are function names, and an arc of the control-flow graph represents a call site. Therefore, we refer to this graph as a *call graph*.

Example 2.1. Figure 1 shows the control-flow graph for the following program to multiply two numbers. The labels on the CFG arcs correspond to the numbering of calls sites in the program text.

```
mult(m,n) = ¹loop1(m,0,n,0)
loop1(i,j,n,a) = if i=0 then a else
                             ²loop2(i-1,n,n,a)
loop2(i,j,n,a) = if j=0 then ³loop1(i,j,n,a) else
                             ⁴loop2(i,j-1,n,a+1)
```

More specific to our topic is the concept of *size-change graphs*. These graphs convey information on the transformation of data values during a state transition; in our functional-programming setting, they describe relations between parameter values in the caller and in the callee. We use the notation $Param(f)$ for the set of parameters of function f. We assume that parameters are named in a unique way across the program and denote the set of all parameters by Par.

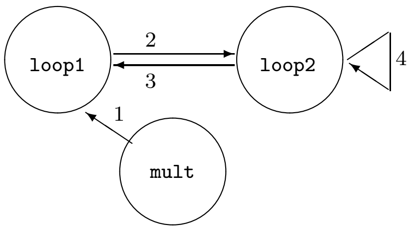


Fig. 1. Control-flow graph, annotated with call labels, for example program 2.1

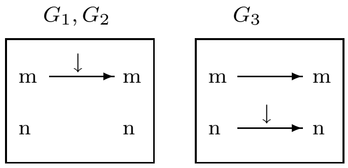


Fig. 2. Size-change graphs for Example 1.1 (`ack`)

Definition 2.1. A *size-change graph* G for a call from function f to function g (written $G : f \to g$) is a bipartite annotated digraph $(\mathit{Param}(f), \mathit{Param}(g), A)$, whose arcs (directed from $\mathit{Param}(f)$ to $\mathit{Param}(g)$) are labeled with either $\downarrow$ or $\overline{\downarrow}$.

An arc $x \xrightarrow{r} y$ of the size-change graph represents a relation between the value of parameter x in f and the value passed for parameter y of g in the given call. A *strict arc* $x \xrightarrow{\downarrow} y$ indicates that a value *definitely decreases* in size, while a *non-strict arc* $x \xrightarrow{\overline{\downarrow}} y$ indicates that the size does not increase. The absence of an arc between a pair of parameters means that neither of these relations is asserted (although they may hold). Figure 2 shows the size-change graphs for Example 1.1 (`ack`). In diagrams, the labels of non-strict arcs are omitted to avoid clutter.

Extracting information about size-changes in a program is a static analysis task. The analysis goal is to produce graphs containing as much information as possible while being *safe*, which means that the assertions about size relations must hold in every occurrence of the call during program execution. In other words, size-change graphs should be a conservative abstraction of the transition relation of the program.

A simple construction of size-change graphs is to include a non-strict arc if a parameter is simply copied, and a strict arc if the parameter is subjected to *destructive operators* (such as `hd` and `tl` for lists or $\bullet$`-1` for natural numbers).

In more complicated programs including nested function calls, *size analysis* may be necessary for incorporating knowledge about the results of subcomputations. For example, in the program below, the analysis would deduce that the return value of ^{2}`sub(m,n)` is smaller than the value of `m` (because `n` must be positive in this call). Thus an arc `m` $\xrightarrow{\downarrow}$ `m` can be safely included in the size-change graph for Call 1.

Example 2.2.

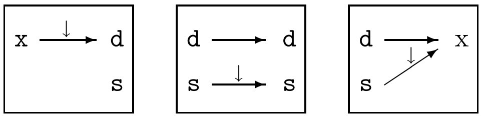
```
quot(m,n) = if n=0 then #error else
            if m<n then 0 else
               ^1quot(^2sub(m,n),n)+1
sub(a,b) = if b=0 then a else
               ^3sub(a-1,b-1)
```

Another source for safe size-change information is the conditions of `if` expressions. The code below originated from a program transformed by a binding-time improvement called The Trick [Lee 1999]. In analyzing Call 3 from `f` to `g`, the condition `d=hd(s)` is known to hold and accounts for the arc `d` $\to$ `x` in the size-change graph G_3.

Example 2.3.

```
g(x) = if x=0 then 0 else
          ^1f(x-1,[0,1,2])+1

f(d,s) = if d=hd(s) then
            ^3g(hd(s)) else
            ^2f(d,tl(s))
```



G_1 : g $\to$ f G_2 : f $\to$ f G_3 : f $\to$ g

In analyzing programs over structured data, the choice of the notion of "size" (often called *norm* in the termination literature) may also be important for obtaining useful size-change graphs. In fact, it may even be useful to measure a single parameter with multiple norms, represented by multiple nodes in the size-change graph. For example, the following tail-recursive program to calculate the tree-size of an S-expression can be proved size-change terminating if we include two nodes in the size-change graphs, one representing the *list length* of `tr` and one representing its tree size.

Example 2.4.

```
size(tr) = if tr=[] then 1 else
           if tl tr=[] then ^1size(hd tr)+2  else
              ^2size(((hd tr) : hd tl tr) : tl tl tr)
```

This topic is further discussed in [Genaim and Codish 2002]. Observe that pursuing this strategy causes a blow-up in the number of size-change graph nodes[3], which presents a greater challenge to a precise SCT analysis.

More generally, size-change graphs can be used to represent relationships among expressions whose valuations are known to be well-founded. One can imagine defining, for a particular application context, a large set of expressions to be tracked (although to do so in a fruitful manner is in practice a highly non-trivial task!). At any rate, the extraction of the size-change graphs is separate from its SCT analysis in our modular 2-stage strategy[4]. For this paper, we expect to be given a set of

[3]Thanks are due to an anonymous referee for pointing this out.
[4]The appeal of this modularity is in the applicability of SCT to logic programming, functional programming, and even term rewriting [Thiemann and Giesl 2005].

possibly complex graphs that embed the size-change information of interest. So, we now leave behind the discussion of program analysis, and proceed to define the SCT condition over the abstract program description.

Definition 2.2. A *size-change annotated call graph* (ACG) for program p is an annotated digraph whose node set represents the function names in p, and to every call c from function f to g corresponds an arc $f \xrightarrow{G_c} g$ such that G_c is a safe size-change graph for c.

In what follows we regard the program p and its ACG as fixed.

2.3 SCT

Definition 2.3. A *multipath* $\mathcal{M}$ over an ACG $\mathcal{G}$ (a $\mathcal{G}$-multipath) is a non-empty, finite or infinite sequence of size-change graphs $G_1 G_2 \dots$ that label a (finite or infinite) path in the call graph.

A more explicit way to write down a multipath, naming the functions involved, is $f_0 \stackrel{G_1}{\to} f_1 \stackrel{G_2}{\to} f_2 \dots$ such that for $i = 1, 2, \dots,$ $f_{i-1} \stackrel{G_i}{\to} f_i$ is an arc of the ACG.

The multipath is often viewed as the single (finite or infinite) annotated digraph obtained by concatenating the sequence of size-change graphs, i.e., identifying the target nodes of G_i with the source nodes of G_{i+1}.

Definition 2.4. A *thread* in a multipath $\mathcal{M} = f_0 \stackrel{G_1}{\to} f_1 \stackrel{G_2}{\to} f_2 \dots$ is a (finite or infinite) directed path in $\mathcal{M}$ (viewed as a concatenated digraph). A thread can be written down in the form $x_0 \stackrel{r_1}{\to} x_1 \stackrel{r_2}{\to} x_2 \dots$, where $x_{i-1} \stackrel{r_i}{\to} x_i \in G_{s+i}$ for some $s \geq 0$ (the starting position of the thread).

A thread *has infinite descent* if $r_i = \downarrow$ for infinitely many i.

A thread is *complete* if it starts at the beginning of the multipath, and has the same length as the multipath.

Definition 2.5. A multipath *has infinite descent* if it has a thread with infinite descent.

We often identify (annotated) digraphs with the set of their (annotated) arcs. In this vein, instead of the annotated call-graph of a program, we may refer to a set of size-change graphs.

Definition 2.6. A set of size-change graphs $\mathcal{G}$ satisfies *size-change termination* (SCT) if every infinite $\mathcal{G}$-multipath has infinite descent.

The size-change principle for program termination reads:

> Let $\mathcal{G}$ be a safe ACG for p. If $\mathcal{G}$ satisfies SCT, then p is terminating.

This is proved in [Lee et al. 2001] for the simple functional language that we are using for our examples. Its correctness for Prolog programs is implicit in [Sagiv 1991; Lindenstrauss and Sagiv 1997b; Codish and Taboch 1997]. We expect it to be routine to justify it for other languages.

All the example programs given so far can be shown terminating by the size-change principle. Below we give a few additional examples. The first two demonstrate how SCT supplies termination proofs for programs previously handled in

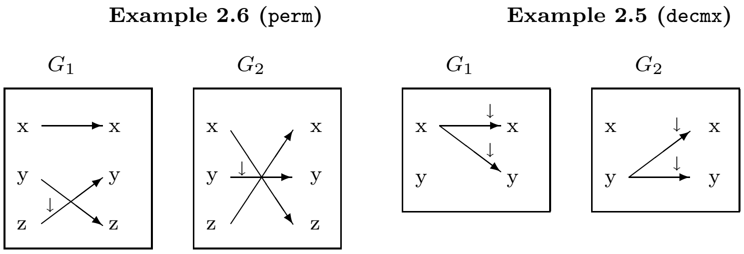


Fig. 3. Size-change graphs showing typical descent behaviors

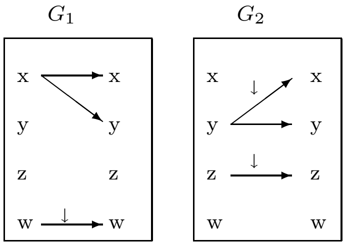


Fig. 4. Size-change graphs for Example 2.7

other ways. Therefore, we give in each case the "alternate" termination argument. The size-change graphs for the examples are shown in Figure 3 and 4.

Example 2.5. In the following program, the value $max(sx, sy)$, where sx and sy are the sizes of `x` and `y`'s values, decreases on every call. This form of descent has been observed in a type-inference algorithm [Frederiksen 2001].

```
decmx(x,y) = if hd(hd(x)) or hd(hd(y)) then true else
             ¹decmx(hd(tl(x)),tl(tl(x))) and
             ²decmx(hd(tl(y)),tl(tl(y)))
```

Example 2.6. In the next program, the three-element multiset $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ is decreased in every call in the sense of having one of its elements replaced by a smaller value (a simple case of multiset descent [Dershowitz and Manna 1979]). At the same time, values are permuted, which is an obstacle to some simple analyses.

```
perm(x,y,z) = if z=[] then x else
              if y=[] then ¹perm(x,tl(z),y) else
                           ²perm(z,tl(y),x)
```

Example 2.7. The following program does not yield to a simple termination argument as the previous examples, but nonetheless it is not difficult to verify that its size-change graphs satisfy SCT. This shows that SCT extends beyond the "natural" set of termination arguments.

```
f(x,y,z,w) = if w=[] then x else
             if y=[] or z=[] then ¹f(x,x,1:z,tl(w)) else
                                  ²f(tl(y),y,tl(z),1:w)
```

2.4 Computational Complexity of SCT

The focus of the current work is on deciding whether a given $\mathcal{G}$ satisfies SCT. We first recall a decision procedure given in [Lee et al. 2001].

Definition 2.7. Consider two size-change graphs $f_0 \xrightarrow{G_1} f_1$ and $f_1 \xrightarrow{G_2} f_2$. Their *composition* $G_1;G_2$ has source function f_0 and target f_2. Its arcs are given by $G^{\downarrow} \cup G^{\overline{\downarrow}}$, where

$$\begin{aligned} G^{\downarrow} &= \{x \xrightarrow{\downarrow} z \mid x \xrightarrow{r_1} y \in G_1,\ y \xrightarrow{r_2} z \in G_2,\ r_1 \text{ or } r_2 \text{ is } \downarrow\} \\ G^{\overline{\downarrow}} &= \{x \xrightarrow{\overline{\downarrow}} z \mid x \xrightarrow{\overline{\downarrow}} y \in G_1,\ y \xrightarrow{\overline{\downarrow}} z \in G_2,\ x \xrightarrow{\downarrow} z \notin G^{\downarrow}\}. \end{aligned}$$

Definition 2.8. For a set $\mathcal{G}$ of size-change graphs, the *composition closure* of $\mathcal{G}$, denoted $\overline{\mathcal{G}}$, is the least set satisfying: $\overline{\mathcal{G}} = \mathcal{G} \cup \{G_1;G_2 \mid f_0 \xrightarrow{G_1} f_1, f_1 \xrightarrow{G_2} f_2 \in \overline{\mathcal{G}}\}$.

THEOREM 2.9. ([Lee et al. 2001]) *$\mathcal{G}$ satisfies SCT if and only if for every size-change graph $f \xrightarrow{G} f$ in $\overline{\mathcal{G}}$ such that $G;G = G$ (an* idempotent *size-change graph), $x \xrightarrow{\downarrow} x \in G$ for some x.*

It follows that SCT can be decided by constructing $\overline{\mathcal{G}}$ and checking the above condition. Such a procedure obviously requires exponential time in the worst case, although it can be implemented in polynomial space. This seems to be the best we can hope for, as indicated by the following hardness result:

THEOREM 2.10. ([Lee et al. 2001]) *SCT is complete for* PSPACE.

The source of the exponential-time complexity in the algorithm is the combinatorial explosion of the closure set. As noted in [Lee et al. 2001], this occurs when values move around between parameter positions. For an illustration of this exponential behavior, consider the program shown in Figure 5. It is not hard to see that the closure set of its natural size-change graphs has size proportional to $n!$. This is due to tracing the values of the sorted data, which are in fact irrelevant to the termination proof. The algorithm presented in this paper is meant to avoid the situation where a program which intuitively is a simple case of SCT defeats the algorithm. We remark that in contrast with previous examples, this one is written in imperative style; we believe that imperative programs are more likely to suffer from such problems than pure-functional ones.

3. THE SCP ALGORITHM

In this section we develop a polynomial-time algorithm (more precisely, cubic in the worst-case), which we name SCP (for Size-Change termination in Polynomial time). It approximates SCT in the conservative sense, i.e., detects a subset of terminating instances, and escapes the exponential-time worst-case of any precise

```
#define swap(a,b) {tmp=a; a=b; b=tmp;}

void sort() {
  int tmp, i;

  i = n-1;
  while (i > 0) {
    if (a1 > a2) swap(a1,a2);
    if (a2 > a3) swap(a2,a3);
    if (a3 > a4) swap(a3,a4);
    .
    .
    .
    if (a_{n-1} > a_n) swap(a_{n-1},a_n);
    i = i - 1;
  }
}
```

Fig. 5. A bubble sort program, with the inner loop unrolled.

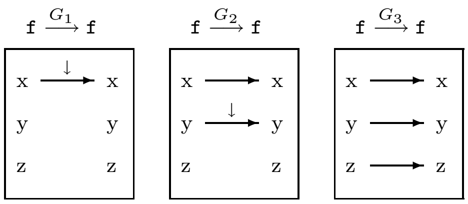


Fig. 6. Graphs not satisfying SCT, but with anchors

SCT procedure. We give some of the motivation for its design, and provide some examples to illustrate the algorithm. A deeper discussion of why we believe that this algorithm is a good choice for scalable SCT analysis is given in Section 5.

3.1 The Basic Strategy

Our polynomial-time algorithm is built around a simple and well-known strategy: identify arcs in the program's flowchart that are "loop anchors," that is, they prevent loops containing them from running forever. This concept is formalized below.

Definition 3.1. The *infinity set* of an infinite sequence $s_1 s_2 \ldots$ is the set of its infinitely repeating elements, namely $\{s \mid \{i \mid s_i = s\}$ is infinite$\}$.

Definition 3.2. Let $\mathcal{H}$ be a set of size-change graphs. We call $G \in \mathcal{H}$ an *SCT anchor* for $\mathcal{H}$ if every $\mathcal{H}$-multipath whose infinity set contains G has infinite descent.

Examples of anchors. The size-change graphs $\mathcal{G}$ in Figure 6 do not satisfy SCT. However, G_1 fulfills the definition of an anchor (consider the complete thread passing through just `x` in a $\mathcal{G}$-multipath—it clearly has infinite descent if G_1 occurs infinitely often). This means that any infinite multipath *without* infinite descent must end in a suffix of G_2 and G_3. But then, for $\mathcal{H} = \{G_2, G_3\}$, G_2 is an anchor (consider the thread passing through y in any $\mathcal{H}$-multipath). Therefore any infinite multipath without infinite descent must end in G_3. In other words, if $\mathcal{G}$ does not satisfy SCT, it must be due to G_3^ω (G_3 repeated infinitely).

Definition 3.3. A strongly connected component of a digraph D is a maximal strongly connected subgraph. A subgraph is *non-trivial* if it contains at least one arc. By SCC(D) we denote the set of non-trivial strongly connected components of D.

The following lemmas follow easily from the definitions.

LEMMA 3.4. *If G is an anchor for $\mathcal{G}$, it is also an anchor for every $\mathcal{H} \subseteq \mathcal{G}$ such that $G \in \mathcal{H}$.*

LEMMA 3.5. *$\mathcal{G}$ satisfies SCT if and only if every non-trivial strongly connected subgraph $\mathcal{H}$ has an anchor.*

Recall that we regularly identify a digraph with its arc set.

Algorithm 3.1. (Generic SCP)
Call *SCP*($\mathcal{H}$) below with each $\mathcal{H} \in$ SCC($\mathcal{G}$) in turn.

Procedure *SCP*($\mathcal{H}$)

1. Determine a set $\mathcal{A}$ of anchors for $\mathcal{H}$.
2. If $\mathcal{A}$ is empty, fail (this terminates the algorithm).
3. Otherwise call *SCP* on each member of SCC($\mathcal{H} \setminus \mathcal{A}$).

THEOREM 3.6. *Assuming the termination of the procedure used to determine anchors, Algorithm 3.1 always terminates. If it does not fail on $\mathcal{G}$, then $\mathcal{G}$ satisfies SCT.*

PROOF. The algorithm terminates, as $\mathcal{H}$ is reduced on each recursive invocation. If $\mathcal{G}$ does *not* satisfy SCT, Lemma 3.5 guarantees a non-empty strongly connected $\mathcal{H}^\star \subseteq \mathcal{G}$ without anchors. Consider the application of SCP to the strongly connected component $\mathcal{H}$ that contains $\mathcal{H}^\star$. By Lemma 3.4, the set $\mathcal{A}$ of anchors found cannot intersect with $\mathcal{H}^\star$. Hence, unless the algorithm fails, $\mathcal{H}^\star$ will be entirely contained in one of the components to which SCP is applied recursively. Thus, unless the algorithm fails, an infinite chain of recursive calls arises. We have already argued that this never occurs. □

How can we locate anchors? A necessary and sufficient condition for a graph to be an anchor can be adapted from Theorem 2.9:

THEOREM 3.7. *A graph $G \in \mathcal{G}$ is an SCT anchor for $\mathcal{G}$ if and only if every idempotent size-change graph $G^\circ = G_1; G_2; \cdots; G_n$, with each $G_i \in \mathcal{G}$ and $G = G_1$, includes an arc $x \xrightarrow{\downarrow} x$ for some x.*

We omit the proof, which is a straight-forward adaption of the proof from [Lee et al. 2001]. Clearly, this theorem does not yield any algorithmic advantage. The SCP algorithm hinges on discovering anchors faster. In the next subsections we show how to do this.

3.2 Anchor-Hunting with Thread Preservers

In this subsection we give the core of our algorithm: two basic tests used to determine anchors. After a few preliminaries, we define the concept of *thread preserver*, which underlies these tests, and present the basis of the tests—theorems that give sufficient conditions for a size-change graph to be an anchor.

Definition 3.8. Let G be a size-change graph.

(1) G is *fan-out free* if the outdegree of any node in G is at most one.

(2) G is *fan-in free* if the indegree of any node in G is at most one.

(3) G *has strict fan-out* if for any node x in G with outdegree larger than one, all arcs $x \to y \in G$ are strict.

(4) G *has strict fan-in* if for any node y in G with indegree larger than one, all arcs $x \to y \in G$ are strict.

We describe a set of size-change graphs as fan-in free, fan-out free etc. if all its graphs have the said property.

We observe that the straight-forward generation of size-change graphs for functional programs, described in Section 2.2, produces fan-in free graphs, but a more advanced analysis as in Example 2.3 may produce fan-in. For Prolog programs, experiments using Terminweb's size analysis [Codish and Taboch 1997] indicate that both its polyhedron analysis and the simpler analysis using monotonicity and equality constraints frequently generate fan-in free graphs. This is despite the fact that equality constraints among variables (due to unifications) are common in Prolog. These observations motivate special attention to the fan-in free case as a starting point. In fact, we will present a simple anchor criterion for the more general class of graphs with strict fan-in.

The two basic anchor criteria to be presented both hinge on the notion of *thread preservers*, introduced next.

Let $Par = \bigcup_f Param(f)$, the combined set of parameters in all the size-change graphs for the subject program.

Definition 3.9. Let $\mathcal{H}$ be a set of size-change graphs. A set $P \subseteq Par$ is called a *thread preserver* for $\mathcal{H}$ if for every $G \in \mathcal{H}$ where $G : f \to g$, it holds that whenever $x \in (Param(f) \cap P)$, there is $x \to y \in G$ for some $y \in P$.

LEMMA 3.10. *Let P be* any *thread preserver for $\mathcal{H}$. Let $\mathcal{M}$ be any infinite $\mathcal{H}$-multipath. For every node in $\mathcal{M}$ that corresponds to a parameter of P, $\mathcal{M}$ contains at least one infinite thread starting at this node and staying within P.*

It is easy to see that the set of thread preservers of a given ACG is closed under union. Hence, there is always a unique maximal thread preserver (MTP) for $\mathcal{H}$, which we denote by MTP($\mathcal{H}$). We will see later that the MTP is easy to compute.

A thread preserver represents a subset of parameters intuitively expected to be useful to size-change termination: these parameters maintain a continuous dataflow over any execution path. Consider Figure 6 on Page 10 again, and observe that MTP($\{G_1, G_2, G_3\}$) $= \{\mathtt{x}\}$, while MTP($\{G_2, G_3\}$) $= \{\mathtt{x}, \mathtt{y}\}$.

Definition 3.11. For a size-change graph G and a set of parameters P, we denote by G^P (*G restricted to P*) the subgraph of G induced by the nodes of G corresponding to parameters in P. For a set $\mathcal{H}$ of size-change graphs, $\mathcal{H}^P \stackrel{\text{def}}{=} \{G^P \mid G \in \mathcal{H}\}$.

We now give a couple of lemmas followed by our first criterion—Theorem 3.14. This criterion assumes that the size-change graphs, when restricted to some thread-preserver P, have a simple structure—specifically, the restricted graphs should have strict fan-in (see Definition 3.8).

LEMMA 3.12. *If size-change graphs $G_1, \ldots, G_n$ have strict fan-in, so does their composition $G_1; \cdots; G_n$.*

LEMMA 3.13. *Suppose $\mathcal{H}$ has strict fan-in, and the outdegree of any source node of its size-change graphs is exactly 1. Let $G \in \mathcal{H}$. If G contains a strict arc, then G is an anchor for $\mathcal{H}$.*

PROOF. Suppose that G contains the strict arc $x \stackrel{\downarrow}{\to} y$. Consider any idempotent graph $G^\circ = G_1; G_2; \cdots; G_n$, with each $G_i \in \mathcal{H}$ and $G = G_1$. Observe that in G° the outdegree of each source node is 1. Thus there is a unique z such that $x \stackrel{\downarrow}{\to} z \in G^\circ$. Consider the case $z = x$; then we have the strict arc $x \stackrel{\downarrow}{\to} x$, as required by Theorem 3.7. Now assume that $z \neq x$, and consider the composition $G^\circ; G^\circ$. By idempotence, this composition must yield an arc $x \to z$ as well. Since all outdegrees are 1, this arc must result of the arc $x \to z$ in the first copy of G° followed by $z \to z$ in the second copy. Thus G° includes two arcs entering z. By fan-in strictness, the arc $z \to z$ is strict, fulfilling once again the condition in Theorem 3.7. Hence G is an anchor. □

THEOREM 3.14. *Suppose P is a thread preserver for $\mathcal{H}$ such that $\mathcal{H}^P$ has strict fan-in. Let $G \in \mathcal{H}$. If G^P contains a strict arc, then G is an anchor for $\mathcal{H}$.*

PROOF. Suppose that G^P contains the strict arc $x \stackrel{\downarrow}{\to} y$. Now, as P is a thread preserver, any source node of a size-change graph of $\mathcal{H}^P$ has an outdegree greater than 0. We will delete certain arcs from the size-change graphs of $\mathcal{H}^P$ so that every outdegree is reduced to 1. For the source node x in G^P, delete every arc leaving this node other than $x \stackrel{\downarrow}{\to} y$. For any other source node (across $\mathcal{H}^P$), simply pick an arc leaving the node and delete the rest. It is easy to see that P remains a thread preserver under such deletion, and still has strict fan-in. Denote by G' the reduced copy of G^P and by $\mathcal{H}'$ the set of reduced graphs. Clearly, if G' is an anchor for $\mathcal{H}'$, then so is G for $\mathcal{H}$. But the previous lemma guarantees that G' is an anchor. □

Definition 3.15. Let P be a thread preserver for $\mathcal{H}$ such that $\mathcal{H}^P$ has strict fan-in. A size-change graph $G \in \mathcal{H}$ is a *Type 1 anchor with respect to P* if G^P contains a strict arc.

To locate Type 1 anchors, we require a thread preserver. Our basic strategy (refined later) is to compute the MTP and check whether any size-change graph is a Type 1 anchor with respect to it.

An example. Consider the size-change graphs $\mathcal{G}$ for Example 1.1 (Figure 2 on Page 5). Clearly, $\mathcal{G}$ is strongly connected. We work out that MTP($\mathcal{G}$) = $\{\mathtt{m}\}$.

We deduce that G_1 and G_2 are anchors for $\mathcal{G}$, as they have the arc $\mathtt{m} \stackrel{\downarrow}{\to} \mathtt{m}$. This means that the occurrence of either graph in the infinity set of a $\mathcal{G}$-multipath implies infinite descent in $\mathtt{m}$. Removing these graphs from consideration then leaves $\mathcal{H} = \{\mathtt{ack} \stackrel{G_3}{\to} \mathtt{ack}\}$. We work out that $\mathrm{MTP}(\mathcal{H}) = \{\mathtt{m}, \mathtt{n}\}$, and deduce that G_3 is an anchor for $\mathcal{H}$, as it has the arc $\mathtt{n} \stackrel{\downarrow}{\to} \mathtt{n}$. We conclude that $\mathcal{G}$ satisfies SCT.

A Second Anchor Criterion. Theorem 3.18 below characterizes a different type of anchor that can be found efficiently.

An intuitive reasoning leading to Theorem 3.18 is the following. Any thread preserver contains an infinite thread in every infinite multipath. An infinite thread does *not* have infinite descent only if it eventually stays inside a strongly connected subgraph of non-strict arcs ("a subgraph of no descent"). Thus, if there is a thread preserver that avoids such subgraphs, a thread of infinite descent is ensured.

For an example, see Figure 7. The only strongly connected subgraph of no descent is induced by the parameters $\{\mathtt{a}, \mathtt{b}, \mathtt{x}\}$. Now, there is a thread preserver $P = \{\mathtt{b}, \mathtt{y}\}$ that avoids all arcs of this subgraph; this is sufficient to guarantee an infinitely descending thread within P. The following definitions make this intuitive reasoning precise.

Definition 3.16. The *size-relation graph due to $\mathcal{H}$*, denoted $SRG_{\mathcal{H}}$, is the annotated digraph with node set *Par* and arc set formed by the union of all arc sets in $\mathcal{H}$. Each arc is labeled, in addition to the size-change label it already has, by an identification of the graph giving rise to it. Formally, the arc set is $\{x \xrightarrow[G]{r} y \mid x \stackrel{r}{\to} y \in G \in \mathcal{H}\}$.

Note that $SRG_{\mathcal{H}}$ is simply a different presentation of the information in $\mathcal{H}$. A thread of an $\mathcal{H}$-multipath induces a directed path in $SRG_{\mathcal{H}}$. Specifically, consider a multipath $\mathcal{M} = G_1 G_2 \ldots$ and a thread $x_0 \stackrel{r_1}{\to} x_1 \stackrel{r_2}{\to} x_2 \ldots$, beginning (without loss of generality) at the beginning of $\mathcal{M}$. Then $x_0 \xrightarrow[G_1]{r_1} x_1 \xrightarrow[G_2]{r_2} x_2 \ldots$ is a path in $SRG_{\mathcal{H}}$. Conversely, every (directed) path in $SRG_{\mathcal{H}}$ corresponds to a thread in some $\mathcal{H}$-multipath. Specifically, the path $x_0 \xrightarrow[G_1]{r_1} x_1 \xrightarrow[G_2]{r_2} x_2 \ldots$ corresponds to a thread in the multipath $\mathcal{M} = G_1 G_2 \ldots$.

Definition 3.17. Let $\mathcal{H}$ be a set of size-change graphs. The *no-descent graph* $NDG_{\mathcal{H}}$ is the subgraph of $SRG_{\mathcal{H}}$ consisting of just the non-strict arcs.

The *interior* of the no-descent graph, $NDG^{\circ}_{\mathcal{H}}$, is the subgraph of $NDG_{\mathcal{H}}$ consisting of all the arcs internal to a strongly connected component of $NDG_{\mathcal{H}}$.

THEOREM 3.18. *Let $G \in \mathcal{H}$, and $G^{\downarrow}$ be G minus any arc belonging to $NDG^{\circ}_{\mathcal{H}}$. Consider $\mathcal{H}' = (\mathcal{H} \setminus \{G\}) \cup \{G^{\downarrow}\}$. If $\mathcal{H}'$ has a non-empty MTP, then G is an anchor for $\mathcal{H}$.*

PROOF. Assume, for a contradiction, that there is an $\mathcal{H}$-multipath $\mathcal{M} = G_1 G_2 \ldots$ without infinite descent, whose infinity set contains G. Let $\mathcal{M}'$ be obtained by replacing in $\mathcal{M}$ every occurrence of G with $G^{\downarrow}$. Clearly, every thread of $\mathcal{M}'$ is also a thread of $\mathcal{M}$. Let $P = \mathrm{MTP}(\mathcal{H}')$. By Lemma 3.10, $\mathcal{M}'$ has a complete thread *th* among P parameters. By assumption, *th* does not have infinite descent. Thus *th* eventually stays within $NDG^{\circ}_{\mathcal{H}'}$. Since $G^{\downarrow}$ has no arc in $NDG^{\circ}_{\mathcal{H}}$ and hence no arc

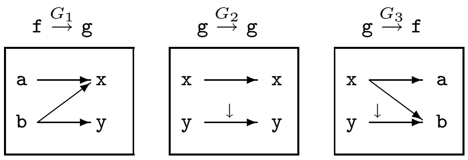


Fig. 7. Example to Theorem 3.18

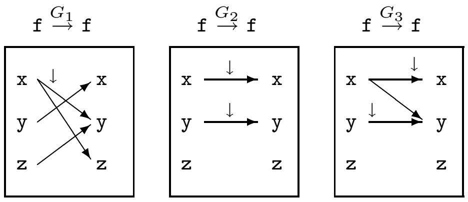


Fig. 8. An example without strict fan-in.

in $NDG^{\circ}_{\mathcal{H}'}$, this implies that *th* cannot pass infinitely often through $G^{\downarrow}$ (which it does, since $G^{\downarrow}$ is in the infinity set of $\mathcal{M}'$). □

Definition 3.19. Graph $G \in \mathcal{H}$ is called a *Type 2 anchor* if $(\mathcal{H} \setminus \{G\}) \cup \{G^{\downarrow}\}$ has a non-empty MTP, where $G^{\downarrow}$ is G minus any arc belonging to $NDG^{\circ}_{\mathcal{H}}$.

Comparison with Type 1 anchors. We now give three examples to illustrate the relative strengths of the two anchor tests.

(1) The SCT instance in Figure 7 shows an advantage of the anchor test due to Theorem 3.18 over the anchor test due to Theorem 3.14. In this instance, we *discover* the parameter set $P = \{\mathtt{b}, \mathtt{y}\}$, giving rise to the termination proof, by computing the maximal thread preserver of $\{G_1^{\downarrow}, G_2, G_3\}$, when checking the conditions of a Type 2 anchor with respect to G_1. Note that G_1 is also a Type 1 anchor with respect to P, since $\mathcal{G}^P$ has strict fan-in; but Theorem 3.14 provides no efficient approach for uncovering this thread preserver (in general, simply deciding whether a thread preserver with strict fan-in exists is NP-hard; for a proof see Appendix A).
(2) Next, consider the graphs of Figure 8. When checking the conditions of a Type 2 anchor for G_2, we compute the MTP of $\{G_1, G_2^{\downarrow}, G_3\}$ and discover $P = \{\mathtt{x}, \mathtt{y}\}$. This time, P happens to be also the MTP of the original graph set; but the graphs restricted to it do not have strict fan-in, which renders Theorem 3.14 unusable.
(3) Finally, consider the graphs of Example 2.6 in Figure 3 on Page 8. The reader may verify that $G_1^{\downarrow}$ contains the arcs $\mathtt{y} \xrightarrow{\overline{\downarrow}} \mathtt{z}$ and $\mathtt{z} \xrightarrow{\downarrow} \mathtt{y}$, and $G_2^{\downarrow}$ contains the single arc $\mathtt{y} \xrightarrow{\downarrow} \mathtt{y}$. Neither G_1 nor G_2 are Type 2 anchors, but both are Type 1 anchors with respect to the MTP.

To conclude, the two anchor tests do not subsume each other in general and it is desirable to apply both.

3.3 Applying the Anchor Tests: Transposition

Definition 3.20. Let $G = (Param(f), Param(g), A)$ be a size-change graph. Its *transposition* is $G^t = (Param(g), Param(f), A^t)$ with $A^t = \{x \xrightarrow{r} y \mid y \xrightarrow{r} x \in A\}$. For a set $\mathcal{H}$ of size-change graphs, $\mathcal{H}^t = \{G^t \mid G \in \mathcal{H}\}$.

Note that the control-flow graph underlying $\mathcal{H}$ is also transposed in $\mathcal{H}^t$.

Our algorithm applies the anchor tests both to $\mathcal{H}$ and to $\mathcal{H}^t$. First, let us see why this is sound:

LEMMA 3.21. *SCT is closed under transposition. Furthermore, $G \in \mathcal{H}$ is an anchor for $\mathcal{H}$ if and only if G^t is an anchor for $\mathcal{H}^t$.*

PROOF. This property is evident from the formulation of the SCT condition in terms of the composition closure (Theorems 2.9 and 3.7). □

Next, why is it useful to transpose? Consider Example 2.5 (with size-change graph set $\mathcal{G}$ shown in Figure 3 on Page 8). There is no non-empty thread preserver, so our anchor-finding approach is doomed. But $\mathcal{G}^t$ does have a thread preserver– its MTP $\{\mathtt{x}, \mathtt{y}\}$– and the restriction of $\mathcal{G}^t$ to the MTP has strict fan-in. Both $\mathcal{G}^t$ graphs can be established as Type 1 anchors with respect to the MTP, or as Type 2 anchors. In general, by always applying the anchor tests to the transposed graphs as well, we compensate for a weakness of the thread-preserver based approach—the lack of symmetry with respect to transposition. In the following subsection, we see another reason for testing the transposed graphs.

3.4 Further Motivation for the Anchor Tests

The definitions of Type 1 and Type 2 anchors can be motivated by noticing that they describe certain “descent patterns”—multiset descent and min-/max-descent respectively, as explained below.

Type 1 anchors and multiset descent. Multiset ordering was introduced as a tool for termination proofs in [Dershowitz and Manna 1979]. Under this ordering, multisets M, M' of elements from a given ordered universe satisfy $M \le M'$ if M can be obtained from M' by the removal of a sub-multiset X, and the introduction of any finite number (possibly zero) of new elements, each of which is smaller than one of the elements removed. Naturally, $M < M'$ holds if the multiset X is not empty.

Given size-change graphs for a program, we may identify multiset descent in a call $f \to g$ as follows: a subset $X = \{x_1, \dots, x_k\}$ of source parameters are related to a subset $Y = \{y_1, \dots, y_\ell\}$ of target parameters such that

(1) for each $y \in Y$, there is an arc $x \to y$ from some $x \in X$;

(2) if the arc is *not* strict, then it is the only arc from x into Y.

If a subset P of the program's parameters can be identified such that every call exhibits multiset descent among P parameters, we may use multiset-descent arguments for a termination proof.

With a bit of reflection, the reader may observe that these conditions imply that P is a thread preserver for the *transposed* graphs $\mathcal{G}^t$, and that $\mathcal{G}^P$ has strict fan-out. Thus, Type 1 anchors for $\mathcal{G}^t$ express, in some sense, the argument that a loop must terminate because of multiset descent.

We found it more intuitive to introduce the anchors in the "forward" direction, considering $\mathcal{G}$ rather than $\mathcal{G}^t$ and strict fan-in rather than strict fan-out, because this direction matches the intuitive significance of thread preservers. The significance of Type 1 anchors is investigated in greater depth in Section 5.

Type 2 anchors and min- and max-descent. A simpler observation related to the significance of thread preservers is the following: the thread preserver's parameters hold a multiset of values such that its minimum never increases. This is so because the parameter that happens to hold the minimum always has an outgoing size-change arc. If, in the course of completing a loop in the program, the minimum actually decreases, this can be used to prove that the loop must terminate.

To establish such a fact, our strategy is to show that as the minimum value moves among parameters, it must (within a certain number of transitions) traverse a strict arc. This is the essential consideration leading to the definition of Type 2 anchors; a more precise explanation of their significance in terms of "descent in the minimum" is given in Section 5.

Our argument above relates to the thread preserver in the forward direction, but it is not hard to observe that if it is $\mathcal{G}^t$, rather than $\mathcal{G}$, that has a thread preserver, we can similarly deduce that the *maximum* among the values of those parameters does not increase.

4. IMPLEMENTATION

In this section we give an algorithm for computing the Maximal Thread Preserver in linear time. We then present the complete SCP algorithm based on the detection of Type 1 and Type 2 anchors and analyze its running time.

4.1 Input Representation

The input to our algorithm is a concrete representation of the set $\mathcal{G}$ of size-change graphs. The graphs are represented internally in the standard linked-list structure. Function names (of the functional subject program) as well as parameters are represented as unique objects. A size-change graph $G : f \to g$ has pointers to f and g, and an arc $x \xrightarrow{r} y$ has pointers to x and y.

We also assume all linking to be bidirectional, e.g., a size-change graph node has a list of all arcs entering it, and an arc points back at the size-change graph where it belongs.

Building such a representation from a textual representation of the data can be accomplished in linear time using standard methods (Paige and Yang [1997] discuss this issue with much generality). By the *size* of a structure we mean the number of objects it contains. For example, the size of a set of size-change graphs is the sum of the numbers of size-change graph nodes and arcs over all size-change graphs in the set.

Finally, we assume that the objects in our data structure accommodate some extra fields that are necessary in the following algorithms.

4.2 Computing the MTP

LEMMA 4.1. *Let $\mathcal{H} \subseteq \mathcal{G}$ be given as a set of pointers to size-change graphs in $\mathcal{G}$, and let N be the size of $\mathcal{H}$. Then MTP($\mathcal{H}$) can be computed in time $O(N)$.*

PROOF. The following algorithm supports this lemma.

Algorithm 4.1. (MTP)

The algorithm maintains a count for each source node of each size-change graph of $\mathcal{H}$. This count initially records the node's out-degree. If the out-degree is 0 in some graph, mark the parameter associated with this node. Make a work-list of all marked parameters. Then process elements from the work-list in turn. When processing x, inspect the size-change arcs entering x. For each such arc $y \rightarrow x$, reduce the count of the source node. If the count becomes 0, and the parameter y is unmarked, mark y and insert it into the work-list. Stop when the work-list is empty. We claim that upon termination, the unmarked parameters form MTP($\mathcal{H}$).

Correctness. Let S be the set of unmarked parameters at the completion of the algorithm. We claim:
(i) S is a thread preserver.
(ii) If P is any thread preserver, then $P \subseteq S$.

To show (i), let $x \in S$, i.e., x is unmarked when the algorithm terminates, and suppose that $x \in \mathit{Param}(f)$. Consider graph $G \in \mathcal{H}$ such that $G : f \rightarrow g$ for some g. The count for the source node x in G was initialized to its out-degree, and remains non-zero (since $x \in S$). For every arc $x \rightarrow y \in G$ such that $y \notin S$ (i.e., y is marked), the count was decreased when y was removed from the work-list, so if the count for x is non-zero on termination, there is an arc $x \rightarrow y \in G$ such that $y \in S$. Thus S fulfills the definition of a thread preserver.

To show (ii), assume the contrary. Let x be the first element of P that was removed from S, i.e., marked, during the algorithm. If x was marked in the first stage, it had to have out-degree 0 in one of the size-change graphs, but then P would not be a thread preserver. Hence, x was marked at a later stage; and it was marked because all the arcs leaving x in some graph G led to parameters already marked. As P is a thread preserver, G has some arc $x \rightarrow y$ such that $y \in P$. But then y would be an element of P marked before x, a contradiction.

Efficiency. The algorithm spends linear time (in the size of the data structure) on initializing the counts and the work-list. Every parameter x is removed from the work-list at most once. During the processing of x, constant time is spent on each arc entering nodes named x. Thus, the total of this work is $O(N)$ over all iterations. □

4.3 The Complete Algorithm

Algorithm 4.2. (SCP, complete)

Call $\mathit{SCP}(\mathcal{H})$ below with each $\mathcal{H} \in \mathrm{SCC}(\mathcal{G})$ in turn.

Procedure $\mathit{SCP}(\mathcal{H})$

1. Compute $\mathit{SRG}_{\mathcal{H}}$ and break into strongly connected components. Remove any arc that is not internal to a component C such that $\mathcal{H}^C$ contains at least one strict arc.
2. Call AnchorFind (Figure 9) on $\mathcal{H}$ to obtain anchors $\mathcal{A}$.
3. Call SCP on each member of $\mathrm{SCC}(\mathcal{H} \setminus \mathcal{A})$.

Step 1, which we call "SRG cleanup," is used to reduce $\mathcal{H}$ by removing arcs that are *a priori* useless for SCT (based on the observation that any infinite thread eventually stays within a single component). Besides potentially reducing the running time in the rest of the procedure, this preprocessing may increase the precision of SCP, as far as Type-1 anchors are concerned, by clearing irrelevant fan-in. In the following example, our algorithm uncovers multiset descent where restriction to the MTP without cleanup fails to find it.

Example 4.1. Consider again Example 2.6 (`perm`), with the size-change graphs shown in Figure 3 on Page 8. Extend the parameter set of function `perm` to $\{\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{a}, \mathtt{b}\}$, and suppose that program analysis reveals that `a` (resp. `b`) bounds the values of $\mathtt{x}, \mathtt{y}, \mathtt{z}$ from above (resp. below) throughout every computation (for example, `a` could be a list supplying initial values to `x`, `y` and `z` while `b` could be the empty list). Thus, non-strict arcs are included in both size-change graphs from `a` to every target node and from every source node to `b`.

Now, neither $\mathcal{G}$ nor $\mathcal{G}^t$ has strict fan-in within the MTP, so no Type 1 anchors are detected, but deleting the irrelevant arcs as in Step 1 above uncovers the anchors as in the original example.

Running time. It is well-known that breaking a graph into strongly connected components can be done in linear time (see, for example, [Cormen et al. 2001, Chapter 22]). Using appropriate data structures, SRG cleanup can be performed with the same efficiency. We will show below that the running time of our implementation of AnchorFind is at most quadratic in the size of $\mathcal{H}$, more precisely $O(Nn)$, where N is the size of $\mathcal{H}$ and n is the number of size-change graphs in $\mathcal{H}$. It follows that the running time of *SCP*($\mathcal{H}$), excluding the recursive calls, is $O(Nn)$.

Consider a complete execution of the SCP algorithm on input $\mathcal{G}$ of size N, containing n size-change graphs. The recursive calls can be seen as forming a recursion tree. Different calls at the same level of the tree process disjoint subsets of $\mathcal{G}$. Thus their total running time remains $O(Nn)$. The height of the tree is limited by n, because at least one anchor is removed before every recursive call performed. We conclude that the total running time is $O(Nn^2)$, i.e., at most cubic in the input size (clearly, in many cases this is an overestimate). Observe that for a fixed control-flow graph, the time grows linearly in the number of size-change graph nodes; thus, the algorithm pays minimal penalty for analyzing many parameters (or parameters that are duplicated to represent different norms), in contrast to the closure-based SCT algorithm, where it is precisely the number of nodes that dominates the exponential growth.

Efficiency of SRG Cleanup and AnchorFind. SCP begins by building a linked representation of the SRG. This is straight-forward, given a representation of $\mathcal{G}$ as described earlier in this section. The SRG representation is coupled with the representation of $\mathcal{G}$ by using a single object for each size-change arc, which participates in both structures. This makes operations like finding the SCC's of the SRG and removing redundant arcs from the size-change graphs easy to perform in linear time.

Using the linear-time MTP algorithm, the running time of Type1Anchors is $O(N)$, but the loop in Type2Anchors, which tests each size-change graph in turn, increases the worst-case running time of AnchorFind to $O(Nn)$.

Procedure AnchorFind($\mathcal{H}$)

1. If $\mathcal{A} = \text{Type1Anchors}(\mathcal{H})$ is non-empty, return $\mathcal{A}$.
2. If $\mathcal{A} = \text{Type1Anchors}(\mathcal{H}^t)$ is non-empty, return $\mathcal{A}$.
3. If $\mathcal{A} = \text{Type2Anchors}(\mathcal{H})$ is non-empty, return $\mathcal{A}$.
4. If $\mathcal{A} = \text{Type2Anchors}(\mathcal{H}^t)$ is non-empty, return $\mathcal{A}$.
5. Fail (this terminates SCP).

Procedure Type1Anchors($\mathcal{H}$)

1. Compute $P = \text{MTP}(\mathcal{H})$.
2. If $\mathcal{H}^P$ has strict fan-in, let $\mathcal{A} = \{G \in \mathcal{H} \mid \exists\, x \xrightarrow[G]{\downarrow} y \text{ in } \mathcal{H}^P\}$; else let $\mathcal{A} = \{\,\}$.
3. Return $\mathcal{A}$.

Procedure Type2Anchors($\mathcal{H}$)

1. Compute $P = \text{MTP}(\mathcal{H})$ and $D = NDG^{\circ}_{\mathcal{H}^P}$.
2. For each $G \in \mathcal{H}$
   a) Work out $G^{\downarrow}$ from G^P and D, and let $\mathcal{H}' = \mathcal{H} \setminus \{G\} \cup \{G^{\downarrow}\}$.
   b) Compute $\text{MTP}(\mathcal{H}')$. If it is non-empty, return $\{G\}$.
3. Return $\{\,\}$.

Fig. 9. AnchorFind procedure

5. ON TERMINATION ARGUMENTS CAPTURED BY SCP

The purpose of this section it to provide better insight on the design of SCP, and explain why we expect it to be successful in practice (an expectation sustained by out experiments, reported in Section 7).

Let us contrast SCP with a precise SCT algorithm. We cannot expect the precise algorithm to handle all programs we encounter, because the exponential runtime of the algorithm will force us to cut off a certain set of instances: those of high combinatorial complexity (i.e., a large closure set). Of course, SCP cannot handle all programs either, but its advantage is that its cutoff is based on another measure: the complexity of the termination argument (in a very specific sense). In fact, SCP is built to encompass a certain set of termination proofs that together can handle a wide variety of programs. The point is that termination of real-world programs, even large, is likely to be provable by a low-complexity proof. In this section we make precise the connection of SCP to familiar forms of termination proofs using ranking functions.

We will need a notation for program states. As in our previous examples, we assume a functional-programming setting and identify a program point with a function. A *program state* is thus a pair $(f, \vec{v})$ where f is a function name and $\vec{v} = (v_1, \ldots, v_{\text{arity}(f)})$ is a valuation of f's parameters. A *state transition sequence* is a sequence (finite or infinite) of form:

$$sts = (f_0, \vec{v}_0) \xrightarrow{c_0} (f_1, \vec{v}_1) \xrightarrow{c_1} (f_2, \vec{v}_2) \xrightarrow{c_2} \ldots,$$

where for each $t = 0, 1, \ldots$, the label c_t is a call in function f_t causing the transition from state $(f_t, \vec{v}_t)$ to $(f_{t+1}, \vec{v}_{t+1})$.

Definition 5.1. Suppose a program p is given. Let $\mathcal{S}$ be a well-founded ordered set. Let $f(x_1, \ldots, x_k)$ be a function of p. Let $\rho(\vec{x})$, where $\vec{x} = (x_1, \ldots, x_k)$, be a function that yields a value in $\mathcal{S}$. Let c be a call in p (not necessarily from function f). We call ρ *a ranking function for f anchored at call c* (for short, an (f, c)-ranking function) if the following hold:
(1) If two states $(f, \vec{v}_0), (f, \vec{v}_1)$ appear in a state transition sequence in this order, then $\rho(\vec{v}_0) \geq \rho(\vec{v}_1)$.
(2) If, moreover, the call c is taken between these two points, then $\rho(\vec{v}_0) > \rho(\vec{v}_1)$.

Definition 5.2. A *global* (or *unanchored*) ranking function for a program function f is a function $\rho(\vec{x})$ as above such that if two states $(f, \vec{v}_0), (f, \vec{v}_1)$ appear in a computation in this order, then $\rho(\vec{v}_0) > \rho(\vec{v}_1)$, irrespective of the computation history between these states.

The use of ranking functions for termination proofs goes back to [Turing 1948] and [Floyd 1967]. Different kinds of ranking functions have been employed in termination proofs; intuitively, these correspond to different types of termination arguments, appropriate on different occasions. In the next subsections, we show how SCP relates to some of these techniques.

5.1 Lexicographic ranking functions

The use of lexicographically-ordered lists as the well-founded set for termination proofs goes back to Recursion Theory (witness Ackermann's function, or more generally the Multiply-Recursive Function Definitions [Péter 1967]).

Thiemann and Giesl [2005] argue that SCT analysis is useful for termination proofs of Term Rewriting Systems, among other reasons, because it captures lexicographic descent of a tuple of variables. They illustrate this claim by analyzing a TRS that represents a sorting program. In fact, SCP handles their example. We aim to show that this case is representative.

How does lexicographic ordering arise in a termination proof? We will use the pair $\langle x, y \rangle$ as a ranking function to prove termination when our reasoning takes the following form:

(1) Certain (call) transitions in the program involve a decrease in x (which, in addition, is never increased). This prevents infinite looping in cycles that contain such transitions.

(2) For infinite runs that do not induce descent in x, i.e., do not include the transitions mentioned above, there will be infinite descent in y.

In general, x and y need not be program parameters, but can be any functions: note that y need only be a ranking function for computations that do not have infinite descent in x.

The straight-forward case, where x and y above are just program parameters, is demonstrated by the `ack` function (Example 1.1). Observe that in this example, SCP implicitly constructs the lexicographic ranking function. It begins by identifying {`m`} as a thread preserver, then considers the control-flow graph without the calls that decrease m, and identifies the descent in `n`. We will show that this simple case generalizes to more complicated termination proofs by SCP, because whenever

SCP identifies an anchor, it identifies a ranking function. First, we give a general lemma that relates the way SCP constructs a termination proof (by removing anchors) and lexicographic descent.

LEMMA 5.3. *Let f be a function in a strongly connected ACG $\mathcal{G}$, which is proved terminating by Generic SCP (Algorithm 3.1 on page 11). Let $\mathcal{G} = \mathcal{H}_0 \supset \mathcal{H}_1 \supset \cdots \mathcal{H}_m$ be the sequence of strongly connected ACG components to which SCP is recursively applied, with f in each $\mathcal{H}_k$. Let $G_{c_1}, G_{c_2}, \ldots, G_{c_m}$ be the sequence of anchors identified in these SCP calls. Assume that every anchor G_{c_i} identified is associated with a ranking function $\rho_i(\vec{x})$ for f, anchored at c_i. Then the m-tuple $\langle \rho_1(\vec{x}), \ldots, \rho_m(\vec{x}) \rangle$ is a global ranking function for f under lexicographic order.*

We omit the straight-forward proof. The less-obvious observations, associating SCP anchors with ranking functions, are discussed in the following subsections.

5.2 Multiset Ordering and Its Dual

In Thiemann and Giesl's paper, capturing instances of *multiset descent* is the second strength claimed for SCT analysis. As described in Section 3, the connection of multiset descent to SCP is to be found in Theorem 3.14—when applied to transposed graphs. Recall that the theorem requires strict fan-in; hence for applying it to $\mathcal{H}^t$, we require *strict fan-out* of the original graphs.

THEOREM 5.4. *Let P be a thread preserver for $\mathcal{H}^t$ such that $\mathcal{H}^P$ has strict fan-out. Let $G_c \in \mathcal{H}$ be the size-change graph for Call c and suppose that G_c^P contains a strict arc. Then for every program function f in $\mathcal{H}$, the function ρ mapping a valuation of $Param(f)$ to the multiset of values of $P \cap Param(f)$ is an (f, c)-ranking function under multiset ordering.*

PROOF. Let $\mathcal{M} = G_{c_1} \ldots G_{c_t}$ be any multipath over $\mathcal{H}$, beginning and ending at f. Let G be the composition $G_{c_1}; \cdots; G_{c_t}$. It follows from our assumptions that G^P has strict fan-out. Let M' be the multiset of values of $P \cap Param(f)$ prior to a transition sequence corresponding to the calls $c_1 \ldots c_t$, and M the posterior values. Note that because P is a thread preserver for $\mathcal{H}^t$, every node at the end of $\mathcal{M}$ which corresponds to some $y \in P$ is the endpoint of a complete thread beginning at some $x \in P$ and staying within P. This thread is represented by an arc $x \xrightarrow{r} y$ in G^P. We distinguish two cases:

(1) $r = \overset{=}{\downarrow}$. In this case, by fan-out strictness, there is no parameter y' other than y such that $x \to y' \in G^P$. Thus the prior value of x is either to be found unmodified in y, or can be considered to have been removed and replaced with the smaller value now in y.

(2) $r = \downarrow$. In this case, the posterior value of y is smaller than the prior value of x. By fan-out strictness, every arc $x \xrightarrow{r'} y' \in G^P$ has $r' = \downarrow$ as well. Thus, the value of x has been removed from the multiset and replaced with one or more values that are smaller.

We conclude that $M \leq M'$ under multiset ordering. Moreover, if the multipath contains G_c, it easily follows that G^P contains at least one strict arc. Hence, M is smaller than M'. $\square$

By this theorem, an anchor identified by applying Theorem 3.14 to $\mathcal{H}^t$ implies multiset descent in a subset of parameters. What about the application of the theorem as stated (without transposition)? This implies a related type of descent, which we refer to as the dual of multiset descent. Under this ordering of multisets, M is smaller than M' if M can be obtained from M' by the removal of a sub-multiset $X \neq \{\ \}$, and the introduction of a finite number (possibly zero) of new elements, among which there is at least one smaller than every element removed. When the size of the multisets is bounded, this ordering becomes well-founded and is just what Theorem 3.14 implies (by an analysis as in the above theorem).

Dershowitz and Manna have observed that if the elements of our multisets are taken from a total order, then multiset descent is equivalent to lexicographic descent of the list of multiset elements, sorted in descending order. This connection is nicely complemented by the fact that dual multiset descent is equivalent to lexicographic descent in the upwards-sorted list.

5.3 Theorem 3.18 and slow ranking functions

Sometimes it is easier to prove termination by assigning to a program point a function that is not guaranteed to descend each time the program point is visited, but only after a given number of iterations. We call this a slow ranking function. It is easy to expand Definition 5.1 to accommodate this change. We call function ρ an (f, c, k)-ranking function if the following hold:
(1) If states $(f, \vec{v}_0), (f, \vec{v}_1)$ appear in a computation in this order, then $\rho(\vec{v}_0) \geq \rho(\vec{v}_1)$.
(2) If, moreover, the call c occurs at least k times between the two states, then $\rho(\vec{v}_0) > \rho(\vec{v}_1)$.

THEOREM 5.5. *Let $G_c \in \mathcal{H}$ be the size-change graph for Call c. Let $G_c^{\downarrow}$ be G_c minus any arc belonging to $NDG^{\circ}_{\mathcal{H}}$. Suppose that $\mathcal{H}' = \mathcal{H} \backslash \{G_c\} \cup \{G_c^{\downarrow}\}$ has the non-empty thread preserver P. Then for every program function f in $\mathcal{H}$, the function ρ mapping a valuation of $Param(f)$ to the minimum of the values of $P \cap Param(f)$ is an (f, c, k)-ranking function, where k is the number of source nodes in G_c.*

PROOF. It is easy to see that if P is a thread preserver, and G any size-change graph with source parameters X and target parameters Y, the minimum value among $P \cap X$ is an upper bound on the minimum of $P \cap Y$ following the call corresponding to G. Therefore, function ρ never increases.

To see that it decreases at the rate stated, consider any finite $\mathcal{H}$-multipath $\mathcal{M} = G_{c_1} \ldots G_{c_t}$ beginning and ending at function f, and including G_c at least k times. Let *sts* be a transition sequence corresponding to the calls $c_1 \ldots c_t$, and u the minimum among the values of $P \cap Param(f)$ in the initial state of *sts*. Let x be a parameter of P where this minimum occurs. Consider the multipath $\mathcal{M}'$, obtained by replacing in $\mathcal{M}$ every occurrence of G_c with $G_c^{\downarrow}$. Clearly, every thread of $\mathcal{M}'$ is also a thread of $\mathcal{M}$. Since P is a thread preserver for the graphs of $\mathcal{M}'$, there is a complete thread of $\mathcal{M}'$ that begins at x, stays within P, and ends at some $y \in P \cap Param(f)$. This thread has to contain a strict arc; for otherwise, by the pigeon-hole principle, we can easily demonstrate that an arc of $G_c^{\downarrow}$ participates in an NDG cycle. Let v be the minimum among the values of $P \cap Param(f)$ at the end of the transition sequence *sts*. Then $v < u$, i.e., the value of ρ at the end of *sts* is smaller than the value of ρ at the beginning. □

We have established that an anchor identified by Theorem 3.18 is related to descent in the minimum value among a set of parameters. What about the application of the theorem to the transposed graphs? We leave it to the reader to verify that an anchor obtained this way is related to descent in the *maximum* among the values of P parameters.

5.4 Summary

We conclude that the termination proofs constructed by SCP embody the following strategies: lexicographic ordering, multiset descent and its dual, and (slow) descent in the minimum or maximum of a set of parameter values. Due to its restriction to Maximal Thread Preservers, SCP cannot capture *every* SCT instance that could be handled by these methods; achieving this would be intractable. Nonetheless, the program examples we have studied (see Section 7) suggest that this may be a minor handicap in practice.

6. A COMPLETENESS RESULT FOR TYPE 1 ANCHORS

Another way to evaluate the coverage of the algorithm is to characterize subproblems for which it is complete. In this section we give a certain completeness result for the algorithm (in fact only relying on Type 1 anchors). It implies that a natural subproblem of SCT can be decided precisely in polynomial time. We start with a couple of lemmas.

LEMMA 6.1. *Let $\mathcal{H}$ be strongly connected and P a thread preserver for $\mathcal{H}$, and suppose that $\mathcal{H}^P$ is fan-in free. Then $Param(f) \cap P$ has the same cardinality for every f in $\mathcal{H}$.*

PROOF. Since $\mathcal{H}$ is strongly connected, there exists some $\mathcal{H}$-multipath $\mathcal{M} = f_0 \stackrel{G_1}{\to} f_1 \dots \stackrel{G_{n+1}}{\to} f_{n+1}$ with $f_0 = f_{n+1}$ containing every graph of $\mathcal{H}$. Let $P_i = Param(f_i) \cap P$. By definition of a thread preserver and the assumption that $\mathcal{H}^P$ is fan-in free, $|P_i| \le |P_{i+1}|$. It follows that $|P_0| \le \dots \le |P_{n+1}| = |P_0|$. Hence $|P_i| = |P_0|$ for every i. □

LEMMA 6.2. *Suppose that $\mathcal{H}$ is strongly connected, has strict fan-in, and is fan-out free. Let $P = MTP(\mathcal{H})$. Assume that $\mathcal{H}^P$ contains no strict arcs. Consider any finite $\mathcal{H}$-multipath $\mathcal{M} = f_0 \stackrel{G_1}{\to} f_1 \dots \stackrel{G_{n+1}}{\to} f_{n+1}$, and suppose that $x \notin P$ is a source node of G_1 in $\mathcal{M}$. It is possible to extend $\mathcal{M}$ (by adding size-change graphs) to a finite multipath $\mathcal{M}'$ with no complete thread from x.*

PROOF. Assume that a complete thread *th* from some $x \notin P$ exists in $\mathcal{M}$, otherwise the claim is trivial. First, we show that *th* never passes through P. Since $\mathcal{H}$ has strict fan-in and we have assumed that $\mathcal{H}^P$ contains no strict arcs, $\mathcal{H}^P$ is actually fan-in free. By Lemma 6.1, the set $P_i = P \cap Param(f_i)$ has the same cardinality for every i. It is not hard to see that the restriction of $\mathcal{M}$ to P nodes consists of exactly $|P_0|$ node-disjoint complete threads, formed from the (non-strict) arcs of $\mathcal{H}^P$, and every P node in $\mathcal{M}$ participates in one of these threads. It follows from the fan-in strictness of $\mathcal{H}$ that no G_i has fan-in at a P node, so the thread *th* must be completely disjoint from P. In particular, it ends at some $z \notin P$.

Now P is the g.f.p. of the function $F : Par \to Par$ given by:

$$F(S) = \{x \mid f \stackrel{G}{\to} g \text{ in } \mathcal{H} \text{ and } x \in Param(f) \cap S \Rightarrow x \to y \in G \text{ for some } y \in S\}$$

Since $z \notin P$, there is a minimal $n \geq 1$ such that $z \notin F^n(Par)$. From the definition of F, it follows easily that a multipath $\mathcal{N}$ of length n exists, with z among its source nodes, such that the (unique) thread from z does not reach a node at the end of $\mathcal{N}$. We let $\mathcal{M}'$ be $\mathcal{M}$ concatenated with $\mathcal{N}$. ☐

THEOREM 6.3. *Suppose that $\mathcal{G}$ has strict fan-in and is fan-out free. Then $\mathcal{G}$ is size-change terminating if and only if every strongly connected $\mathcal{H} \subseteq \mathcal{G}$ has a thread-preserver P such that $\mathcal{H}^P$ contains a strict arc.*

PROOF. The "if" direction follows by applying the SCP algorithm to $\mathcal{G}$ (note that by our assumptions, the MTP will have strict fan-in). To prove the converse direction, suppose that $\mathcal{G}$ is size-change terminating. Let $\mathcal{H}$ be a strongly connected subset of $\mathcal{G}$ and $P = MTP(\mathcal{H})$. We will assume that $\mathcal{H}^P$ contains no strict arc and derive a contradiction.

Let $\mathcal{M}$ be an arbitrary finite multipath $f_0 \stackrel{G_1}{\to} f_1 \ldots$ over $\mathcal{H}$. Repeatedly extend $\mathcal{M}$ as in Lemma 6.2, preventing the non-MTP parameters of f_0 (one at a time) from starting complete threads. Ultimately, the only complete threads in the resulting multipath are within P, and therefore include no strict arcs. Now, extend this multipath so that it ends at f_0. Refer to the result as $\mathcal{M}^\star$. This multipath has no complete thread that includes a strict arc. It immediately follows that the infinite multipath $(\mathcal{M}^\star)^\omega$ does not have infinite descent. Thus, $\mathcal{G}$ does not satisfy SCT, and the proof is complete. ☐

To conclude, if $\mathcal{G}$ has strict fan-in and no fan-out (or, taking the transpose, strict fan-out and no fan-in), SCP is complete. We use this result in our implementation by allowing a negative answer to be qualified with a statement that it is precise, i.e., SCT is *not* satisfied. This happens if a graph set $\mathcal{H}$ on which the SCP procedure fails has one of the above forms, after redundant arcs have been eliminated (Step 1 of Algorithm 4.2).

Note also that for $\mathcal{G}$ of the above forms, any anchor discovered must be due to the subprocedure Type1Anchors. Consequently, SCP is guaranteed to terminate in quadratic time. We have already mentioned that fan-in free graphs arise naturally from a simple analysis to relate the source and target parameter values in a function call. Fan-out strictness signifies that any duplication of data involves some descent. Thus size-change graphs with no fan-in and strict fan-out are arguably a natural subclass to consider.

From a complexity-theoretic viewpoint, the conclusion of Theorem 6.3 is that the subproblem of SCT described is polynomial-time decidable. Indeed, the instances created in the PSPACE hardness proof of [Lee et al. 2001] have non-strict fan-out (though they are fan-in free!), showing that it suffices to allow either non-strict fan-out or non-strict fan-in to raise the decision problem's complexity. Naturally, some of these instances will trip SCP.

In [Lee et al. 2001], a couple of claims on polynomial-time solvability were stated without any proof. These claims are given next, as easy corollaries of Theorem 6.3.

Definition 6.4. A set of size-change graphs $\mathcal{G}$ is *stratifiable* if there exists a labeling of the parameters $\ell : Par \to I\!N$ such that the following hold.

(1) For every function f of the subject program and $x, y \in Param(f)$, $\ell(x) \neq \ell(y)$.
(2) For every $G \in \mathcal{G}$ and $x \to y \in G$, $\ell(x) \geq \ell(y)$.

COROLLARY 6.5. *Let $\mathcal{G}$ be a set of size-change graphs.*

(1) *If $\mathcal{G}$ is fan-in free and fan-out free, then SCT for $\mathcal{G}$ is polynomial-time decidable.*
(2) *If $\mathcal{G}$ is stratifiable, then SCT for $\mathcal{G}$ is polynomial-time decidable.*

PROOF. For the first claim, simply observe that graphs with no fan-in have strict fan-in, by definition. For the second claim, it follows from the stratifiability of $\mathcal{G}$ that each component $C \in SCC(SRG_{\mathcal{G}})$ contains at most one parameter from each $Param(f)$. As noted before, SCT is invariant under the deletion of arcs $x \to y$ where x and y belong to different strongly connected components of $SRG_{\mathcal{G}}$, and such arcs are removed in Step 1 of Algorithm 4.2. In a reduced size-change graph $G' : f \to g$, the presence of an arc $x \to y \in G'$ implies that x and y are in the same component of $SRG_{\mathcal{G}}$. If $x \to z$ is also in G', then z and y are in the same SRG component. Since g has only one parameter in the component, $z = y$. Similarly, if $z \to y \in G'$, then $z = x$. We conclude that G' is fan-in free and fan-out free, and SCT can once again be decided in polynomial-time. □

7. EXPERIMENTS

We have conducted experiments to demonstrate an efficient implementation of SCP and to establish the algorithm's effectiveness in verifying SCT instances encountered in actual termination analysis. Our benchmark is produced using test suites from previous work and consists of SCT instances small enough to be handled by the precise (exponential-time) algorithm as well. Thus, it does not demonstrate the scalability of SCP, but it does show that Algorithm 4.2 is practical, in that it is efficient already for small examples. Of primary concern to us is the precision of SCP. Here, we find that SCP is able to verify or refute SCT for every input in the benchmark.

7.1 Generation of Test Inputs

For the convenient generation of test inputs, we turn to Terminweb [Codish and Taboch 1997], a current, fully-automated Prolog analyzer that uses the size-change reasoning. It has been tested on standard Prolog test suites [Lindenstrauss and Sagiv 1997a] and demonstrates the applicability of the size-change reasoning in practice.

Terminweb takes as input a Prolog program and a *query.* The latter is a pattern representing the procedure call whose termination will be checked. It consists of a procedure name and a tuple indicating which of the call's arguments are *bound* and which are *free.* The termination analysis of the query is carried out in two stages:

(1) An *instantiation analysis* and *size analysis* are first performed. The instantiation analysis computes a a set of argument instantiation patterns which describe possible procedure invocations. Each pattern indicates which of the procedure's arguments will have an instantiated size according to the norm selected (such arguments are said to be *instantiated enough* [Lindenstrauss and Sagiv 1997b]). The size analysis computes size relationships among instantiated arguments in a possible procedure call.
(2) Terminweb employs an abstraction that combines information about argument instantiation and size relations to summarize a procedure call. A "composition

closure" is computed for such abstract descriptions, much as in the standard test for SCT, and checked for possible non-termination.

The above approach incorporates a great amount of context sensitivity in tracking argument instantiation. For simple programs (such as the ones in the test suites), this degree of precision does not appear necessary. We therefore attempt a much coarser analysis of the test suites. We begin with the abstract descriptions produced by Terminweb for possible procedure calls that follow from the input query. From each such description, it is straight-forward to extract a conservative size-change graph. Satisfaction of SCT by the graphs then implies the query's termination. Following this strategy, we generate size-change graphs for each query in the test suites.

We find that in all but 3 instances, a query verified by Terminweb leads to size-change graphs that satisfy SCT. In each of the 3 cases where the graphs do not satisfy SCT, it is due to our overly conservative approximation of argument instantiation. For instance, a typical call to `perm/2` [Apt and Pedreschi 1994] (this is a common Prolog procedure for computing list permutations) leads to invocations of `app/3` (append) in 2 distinct modes: with the first 2 arguments bound (to compute the concatenation of 2 lists) and with the last argument bound (to compute decompositions of a list). Our extraction of size-change graphs conservatively assumes that none of the arguments of `app/3` is instantiated enough here, which results in an empty graph for the self-recursive call in `app/3`. To overcome this lack of context sensitivity, we manually duplicate a number of procedures invoked with different modes. In the case of `perm/2`, the 2 calls to `app/3` are replaced with calls to distinct copies of `app/3`. Among the test suites, such a manipulation has been performed only for the append procedure in the permutation program [Apt and Pedreschi 1994; Plümer 1990] and procedure `q` in the program `pql` [Lindenstrauss and Sagiv 1997a]. This is enough to allow an SCT instance to be extracted from every query verified by Terminweb.

Our purpose in isolating SCT analysis from the analysis of a query is to make it possible to test the precision of SCP. While it appears that a straight-forward combination of instantiation analysis, size analysis, and SCP is adequate to handle simple programs, such as the ones in the test suites, further research will be required before we have an effective, automatic size-change termination analyzer for Prolog.

7.2 Size-change Analysis of Test Inputs

Our implementation of SCP follows Algorithm 4.2 closely, except that after the cleanup step, it is checked whether the current size-change graphs or the transposed graphs satisfy the conditions of Theorem 6.3. In this case, failure of AnchorFind causes the analyzer to declare that SCT has been refuted. In general, an SCP analysis has 3 possible outcomes: verification of SCT, refutation of SCT, or an indication that the analysis was inconclusive.

We have also implemented a precise (exponential-time) SCT test as a reference. While the programs in the test suites are too small to allow us to draw conclusions about the performance of SCP relative to a precise SCT procedure, we have attempted to be fair to SCT in the generation of our benchmark.

There are many ways to improve on the basic algorithm suggested by Theorem 2.9. For example, long and non-branching cycles in the ACG will cause a wasteful increase of the closure set in this algorithm. However, collapsing non-branching cycles introduces an overhead that is unnecessary for the test suites, so we have not included such a step. We have, however, implemented a natural variation of the closure procedure aimed at trimming the closure set. It is based on the observation that a size-change graph generated during the closure computation need not be considered further if a graph seen earlier conservatively approximates it. Additionally, we test every graph generated regardless of idempotence, as in [Lindenstrauss and Sagiv 1997b].

Definition 7.1. Let $G : f \to g$, $G' : f \to g$ be size-change graphs. We write $G' \leq G$ if for each $x \xrightarrow{r'} y \in G'$ we have $x \xrightarrow{r} y \in G$, and if $r' = \downarrow$, then $r = \downarrow$.

Algorithm 7.1. (SCT for $\mathcal{H}$)

(1) Initialize accumulator $\mathcal{S}$ to empty and work-list W to $\mathcal{H}$.
(2) If W is empty, exit, else remove some $G : f \to g$ from W.
(3) If $\mathcal{S}$ includes $G' : f \to g$ such that $G' \leq G$, go back to Step 2.
(4) If $f = g$, and G, considered as a standard digraph (by identifying source and target nodes with the same name), has no SCC with a strict arc, fail.
(5) Insert G into $\mathcal{S}$.
(6) For each $H : g \to h$ in $\mathcal{H}$, insert $G; H : f \to h$ into W.
(7) Go back to Step 2.

The following are all linear-time operations: comparing and composing size-change graphs, and checking the failure condition at Step 4. The running time of the algorithm is dependent on the size of the closure $\mathcal{S}$ computed. Checking $G' \leq G$ at Step 3 instead of $G' = G$ sometimes avoids a blowup in the size of $\mathcal{S}$. For a proof of correctness of Algorithm 7.1, see Appendix B.

7.3 Results

Our benchmark is produced using 123 queries for 97 programs (although certain queries are repeated across different test suites); see [Lindenstrauss and Sagiv 1997a]. The size-change graphs generated lead to 294 strongly connected components, which are fed as inputs to our size-change analyses; for our benchmark, we have elected to treat each strongly connected component as a separate input to the analyzer.

Efficiency of SCP. Tables I to III (at the end of this article) compare timings for our SCP and SCT analysis. For each query in the Prolog test suites, we indicate the following: the number of size-change graph SCC's produced (column *scc* in the tables), whether all the SCC's are size-change terminating (column *res*), the total number of arcs in the SCC's (column *arcs*), and the total time taken to analyze the SCC's using our implementation of Algorithm 7.1 (column *sct*) and Algorithm 4.2 (column *scp*). Timings are given in microseconds (10^{-6} seconds). In the tables, the test suites [De Schreye and Decorte 1994; Apt and Pedreschi 1994; Plümer 1990;

Bueno et al. 1994; Lindenstrauss and Sagiv 1997a] are abbreviated as DD, AP, Pl, B+, and LS respectively.

The experiments have been conducted on a P4 2.6 GHz machine with over 500 MB of RAM, running Linux 2.6.11. The size-change analyses have been coded in C, and compiled with gcc 3.3.5. The approximate time for an SCT or SCP analysis has been worked out from the time taken for $1,000,000$ repetitions.

The timings indicate that SCP is a practical algorithm.

For the interested reader, our implementation of the analyses, as well as instructions on how to repeat the experiments, are available online via the URL `http://www2.mta.ac.il/~amirben/sct.html` (document title: *The Size-Change Termination Project*).

The experiments reveal that our collection of size-change graph SCC's does not require very difficult analysis. After the cleanup step, every SCC or its transposition fulfills the conditions of Theorem 6.3, except for one SCC due to the artificial program `vangelder`, which is nonetheless verified by SCP. Thus, in *every* case, SCP is able to either verify or refute SCT. This supports our belief that SCP is a viable alternative to a precise SCT analysis in practice. It defeats the exponential asymptotic running time of an exact procedure, while performing well on small inputs.

8. CONCLUSION

8.1 Related Work

The automatic verification of program termination is a topic that has been addressed across diverse areas, and relevant discussions range from the innermost normalization of orthogonal term rewrite systems [Arts 1997] to the role of dependent types [Xi 2002]. For an appreciation of the breadth of the subject, one only has to refer to recent proceedings of the Workshop on Termination (WST). Here we will focus on works directly connected to the development of the size-change termination analysis.

Historically, the size-change reasoning, formalized and investigated in [Lee et al. 2001], was present already in the Prolog analyzers Termilog [Lindenstrauss and Sagiv 1997b] and Terminweb [Codish and Taboch 1997]. In fact, the *query-mapping pairs* used in Termilog can be seen as size-change graphs augmented with information about argument instantiation. The abstraction used in Terminweb is strictly more expressive, since argument size relations are captured by linear constraints, but in many cases the termination reasoning can be expressed as size-change termination reasoning.

The work on size-change termination [Lee et al. 2001] can therefore be seen as a distillation of the termination principle in Termilog and Terminweb, or an adaptation of these analyses to a simple first-order functional language. However, the clean formulation of the SCT condition has been helpful from both a theoretical as well as practical standpoint. Theoretically, it has led to results concerning the scope of the size-change reasoning [Ben-Amram 2002] and how efficiently this reasoning can be implemented [Lee et al. 2001]. Practically, the simplicity of SCT has encouraged its application in different contexts. We describe some of them below.

Wahlstedt [2000] has implemented SCT analysis as part of the AGDA theorem prover, while Thiemann and Giesl [2003] have incorporated it in the AProVE analyzer for term rewrite systems. Jones and Bohr [2004] have developed the necessary support for analysis of the pure lambda calculus with a call-by-value semantics. In the area of program transformation, Glenstrup and Jones [2004] have modeled a termination analysis for the offline partial evaluation of a first-order functional language after SCT analysis.

For dealing with integer-valued parameters, Anderson and Khoo [2003; 2005] have refined the size-change graph abstraction to track domain information together with affine relations among parameter values, while Avery [2005] proposes combining a global boundedness analysis with the size-change abstraction. These works use abstractions that add affine constraints to size relations, bringing them closer to Terminweb's analysis or techniques based on predicate abstraction [Podelski and Rybalchenko 2005]. Codish et al. [2005] have recently shown that monotonicity constraints, which are more expressive than size-change graphs, nevertheless admit a complete analysis using the same closure technique.

Finally, we note that complementary analyses can be important to enhance the effectiveness of the size-change method. Terminweb, for example, has been extended with various techniques, including structure untupling [Codish et al. 2000] and automatic norm inference [Genaim and Codish 2002].

8.2 Concluding Remarks

The SCT condition has proven itself useful for the termination analysis of functional and logic programs, and even term rewrite systems. In [Lee et al. 2001], it was proved that the SCT property is complete for PSPACE. Thus, we either have to give up on an exact decision procedure, or we must justify that some SCT test will scale in practice. Unfortunately, we see that the standard tests are easily pushed to an exponential combinatorial search by certain patterns in the size-change graphs.

Thiemann and Giesl [2003] have observed that "a major advantage of the size-change principle is that it can simulate the basic ingredients of RPOS, i.e., the concepts of *lexicographic* and of *multiset* comparisons." This suggests designing an efficient analysis to detect these and other common forms of parameter-descent behavior in a given set of size-change graphs. The SCP algorithm in this paper specifically targets lexicographic descent, multiset descent, and min- and max-descent. It has worst-case complexity cubic in the input size. Theoretically, we have shown that SCP is complete for size-change graphs that have strict fan-out and no fan-in. Thus it solves a natural subproblem of SCT exactly. Empirically, we have tested the precision of SCP using standard benchmark programs and obtained satisfactory results.

SCP has been designed to cope with size-change graphs containing a lot of information that is not relevant to the termination argument. We consider it a piece of the puzzle in the scaling-up of SCT analysis.

Acknowledgement

We thank the TOPLAS referees for their very thorough reports. Their comments contributed substantially to the improvement of this paper.

REFERENCES

ANDERSON, H. AND KHOO, S. C. 2003. Affine-based size-change termination. In *Proceedings of the 1st Asian Symposium on Programming Languages and Systems, APLAS'03*, A. Ohori, Ed. Lecture Notes in Computer Science, vol. 2895. Springer, 122–140.

ANDERSON, H. AND KHOO, S. C. 2005. Bounded size-change termination. Tech. Rep. TRB6/05, National University of Singapore, Singapore.

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.

ARTS, T. 2001. Automatically proving termination and innermost normalisation of term rewriting systems. Ph.D. thesis, Universiteit Utrecht, Netherlands.

AVERY, J. 2005. The size-change termination principle on non well founded data types. Tech. Rep., Datalogisk Institut Københavns Universitet, Denmark.

BEN-AMRAM, A. 2002. General size-change termination and lexicographic descent. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, T. Mogensen, D. Schmidt, and H. Sudborough, Eds. Lecture Notes in Computer Science, vol. 2566. Springer, 3–17.

BUENO, F., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *Proceedings of the International Logic Programming Symposium, ILPS'94*. MIT Press, 320–336.

CODISH, M., LAGOON, V., AND STUCKEY, P. 2005. Testing for termination with monotonicity constraints. In *Proceedings of the 21st International Conf. on Logic Programming, ICLP'05*, to appear.

CODISH, M., MARIOTT, K., AND TABOCH, C. 2000. Improving program analyses by structure untupling. *The Journal of Logic Programming 43,* 3, 251–263.

CODISH, M. AND TABOCH, C. 1997. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *Proceedings of the 6th International Conf. on Algebraic and Logic Programming, ALP'97*, M. Hanus, J. Heering, and K. Meinke, Eds. Lecture Notes in Computer Science, vol. 1298. Springer, 31–45.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 2001. *Introduction to Algorithms.* MIT Press.

DERSHOWITZ, N. AND MANNA, Z. 1979. Proving termination with multiset orderings. *Communications of the ACM 22,* 8 (august), 465–476.

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

FLOYD, R. W. 1967. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics XIX*, 19–32.

FREDERIKSEN, C. C. 2001. A simple implementation of the size-change principle. Tech. Rep. D-442, Datalogisk Institut Københavns Universitet, Denmark.

GENAIM, S. AND CODISH, M. 2002. Combining norms to prove termination. In *Proceedings of the International Workshop on Verification, Model Checking and Abstract Interpretation, VMCAI'02*. Springer.

GIESL, J., THIEMANN, R., SCHNEIDER-KAMP, P., AND FALKE, S. 2004. Automated termination proofs with AProVE. In *Proceedings of the 15th International Conf. on Rewriting Techniques and Applications, RTA'04*. Lecture Notes in Computer Science, vol. 3091. Springer, 210–220.

GLENSTRUP, A. AND JONES, N. 2004. Termination analysis and specialization-point insertion in off-line partial evaluation. Tech. Rep. D-498, Datalogisk Institut Københavns Universitet, Denmark.

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.

LEE, C. S. 1999. Partial evaluation of the Euclidean algorithm, revisited. *Higher-order and Symbolic Computation 12,* 2 (Sep), 203–212.

LEE, C. S. 2001. Program termination analysis and termination of offline partial evaluation. Ph.D. thesis, UWA, Australia.

Lee, C. S., Jones, N. D., and Ben-Amram, A. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages, POPL'01*. ACM.

Lee, C. S. 2002. Program termination analysis in polynomial time. In *Proceedings of the 1st International Conf. on Generative Programming and Component Engineering, GPCE'02*. Lecture Notes in Computer Science, vol. 2487. Sringer, 218–235.

Lindenstrauss, N. and Sagiv, Y. 1997a. Automatic termination analysis of logic programs (with detailed experimental results). This article is available on the internet at `http://www.cs.huji.ac.il/~naomil/`.

Lindenstrauss, N. and Sagiv, Y. 1997b. Automatic termination analysis of Prolog programs. In *Proceedings of the 14th International Conf. on Logic Programming, ICLP'97*, L. Naish, Ed. MIT Press, 64–77.

Paige, R. and Yang, Z. 1997. High level reading and structure compilation. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages, POPL'97*. ACM, 456–469.

Péter, R. 1967. *Recursive Functions*. Academic Press.

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

Podelski, A. and Rybalchenko, A. 2005. Transition predicate abstraction and fair termination. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages, POPL'05*. ACM.

Sagiv, Y. 1991. A termination test for logic programs. In *Proceedings of the International Logic Programming Symposium, ILPS'91*, V. Saraswat and K. Ueda, Eds. MIT Press, 518–532.

Thiemann, R. and Giesl, J. 2003. Size-change termination for term rewriting. In *Proceedings of the 14th International Conf. on Rewriting Techniques and Applications, RTA'03*. Lecture Notes in Computer Science, vol. 2706. Springer, 264–278.

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*, to appear. On-line at Springer Verlag, doi 10.1007/s00200-005-0179-7

Turing, A. M. 1948. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. 67–69. reprinted in: The early British computer conferences, vol. 14 of Charles Babbage Institute Reprint Series For The History Of Computing, MIT Press, 1989.

Wahlstedt, D. 2000. Detecting termination using size-change in parameter values. Master's thesis, Göteborgs Universitet, Sweden.

Xi, H. 1999. Dependent types for program termination verification. *Higher-order and Symbolic Computation 15,* 1 (Mar), 91–131.

Table I. Comparison of timings for SCT and SCP analysis

cf.	*program*	*query*	*scc*	*arcs*	*res*	*sct*	*scp*
DD	dds1.1	`append(b,b,f)`	1	2	Y	5	3
DD	dds1.1	`append(f,f,b)`	1	1	Y	4	2
DD	dds1.1	`append(f,b,f)`	1	1	N	3	2
DD	dds1.2	`permute(b,f)`	2	2	Y	6	4
DD	dds3.8	`reverse(b,f,b)`	1	1	Y	3	2
DD	dds3.13	`duplicate(b,f)`	1	1	Y	3	2
DD	dds3.14	`sum(b,b,f)`	1	2	Y	5	3
DD	dds3.15	`merge(b,b,f)`	1	4	Y	13	4
DD	dds3.17	`dis(b)`	1	5	Y	19	7
DD	dds3.17	`con(b)`	1	5	Y	19	7
AP	list	`list(b)`	1	1	Y	3	2
AP	member	`member(f,b)`	1	1	Y	4	2
AP	subset	`subset(b,b)`	2	4	Y	10	6
AP	subset	`subset(f,b)`	2	2	N	5	4
AP	append	`app(b,f,f)`	1	1	Y	3	2
AP	append	`app(f,f,b)`	1	1	Y	3	2
AP	select	`select(f,b,f)`	1	1	Y	4	2
AP	sum	`sum(f,b,f)`	1	1	Y	3	2
AP	sum	`sum(f,f,b)`	1	1	Y	3	2
AP	lte	`goal`	2	2	Y	6	4
AP	naive_rev	`reverse(b,f)`	2	3	Y	8	5
AP	dc_schema	`dcsolve(b,f)`	3	5	Y	12	8
AP	permutation	`perm(b,f)`	3	4	Y	11	7
AP	quicksort	`qs(b,f)`	3	5	Y	13	8
AP	overlap	`overlap(b,b)`	1	1	Y	3	2
AP	mergesort	`mergesort(b,f)`	3	5	N	18	8
AP	mergesort_ap	`mergesort(b,f,b)`	3	9	Y	26	12
AP	curry_ap	`type(b,b,f)`	2	5	Y	13	7
AP	map	`map(b,f)`	1	1	Y	3	2
AP	gtsolve	`gtsolve(b,f)`	2	2	Y	6	4
AP	ordered	`ordered(b)`	1	1	Y	4	2
AP	fold	`fold(b,b,f)`	1	3	Y	6	3
AP	SS_map_t	`test_color(b,f)`	4	6	Y	17	10
Pl	pl1.1	`append(b,b,f)`	1	2	Y	5	3
Pl	pl1.1	`append(f,f,b)`	1	1	Y	4	2
Pl	pl1.1	`append(f,b,f)`	1	1	N	3	2
Pl	pl1.2	`perm(b,f)`	3	4	Y	11	7
Pl	pl1.2_t	`perm(b,f)`	3	3	Y	9	6
Pl	pl2.3.1	`p(f,b)`	1	1	N	3	2
Pl	pl3.5.6	`p(f)`	1	0	N	1	1
Pl	pl3.5.6a	`p(f)`	1	1	Y	3	2

Table II. Comparison of timings for SCT and SCP analysis (continued)

cf.	*program*	*query*	*scc*	*arcs*	*res*	*sct*	*scp*
Pl	pl4.01	`append3(b,b,b,f)`	1	2	Y	5	3
Pl	pl4.01	`append3(f,b,b,b)`	1	1	N	2	2
Pl	pl4.4.3	`merge(b,b,f)`	1	4	Y	13	4
Pl	pl4.4.6a	`perm(b,f)`	2	3	Y	8	5
Pl	pl4.5.2	`s(b,f)`	1	1	N	3	3
Pl	pl4.5.3a	`p(b)`	1	0	N	1	1
Pl	pl4.5.3b	`goal`	1	0	N	1	1
Pl	pl4.5.3c	`goal`	1	0	N	1	1
Pl	pl5.2.2	`turing(b,b,b,f)`	2	3	N	7	5
Pl	pl6.1.1	`qsort(b,f)`	3	5	Y	13	8
Pl	pl7.2.9	`mult(b,b,f)`	2	4	Y	10	6
Pl	pl7.6.2a	`reach(b,b,b)`	2	4	N	8	5
Pl	pl7.6.2b	`reach(b,b,b,b)`	2	4	N	8	6
Pl	pl7.6.2c	`reach(b,b,b,b)`	3	8	Y	17	10
Pl	pl8.2.1	`mergesort(b,f)`	3	5	N	18	8
Pl	pl8.2.1a	`mergesort(b,f)`	3	6	Y	19	8
Pl	mergesort_t	`mergesort(b,f)`	3	7	Y	25	11
Pl	pl8.3.1	`minsort(b,f)`	3	3	N	11	8
Pl	pl8.3.1a	`minsort(b,f)`	3	4	Y	12	8
Pl	pl8.4.1	`even(b)`	1	2	Y	9	4
Pl	pl8.4.1	`odd(b)`	1	2	Y	9	4
Pl	pl8.4.2	`e(b,f)`	1	5	Y	30	9
B+	aiakl.pl	`init_vars(b,b,f,f)`	4	10	Y	29	13
B+	ann.pl	`go(b)`	26	13	N	49	43
B+	bid.pl	`bid(b,f,f,f)`	7	13	Y	31	18
B+	boyer.pl	`tautology(b)`	3	0	N	6	5
B+	browse.pl	`main`	12	10	N	26	21
B+	deriv.pl	`d(b,b,f)`	1	2	Y	5	3
B+	fib_t	`fib(b,f)`	2	5	Y	16	6
B+	grammar.pl	`goal`	0	0	Y	0	0
B+	hanoiapp.suc	`shanoi(b,b,b,b,f)`	2	10	Y	62	10
B+	mmatrix.pl	`mmultiply(b,b,f)`	3	6	Y	15	9
B+	mmatrix.pl	`trans_m(b,f)`	2	1	N	5	4
B+	money.pl	`money(f,f,f,f,f,f,f,f)`	2	2	Y	6	4
B+	occur.pl	`occurall(b,b,f)`	3	6	Y	15	9
B+	peephole.pl	`peephole_opt(b,f)`	8	14	N	64	29
B+	progeom.pl	`pds(b,f)`	6	11	N	26	17
B+	qplan.pl	`qplan(b,f)`	21	75	N	198	99
B+	qsortapp.pl	`qsort(b,f)`	3	5	Y	13	8
B+	query.pl	`query(f)`	0	0	Y	0	0
B+	rdtok.pl	`goal`	9	14	N	45	33

Table III. Comparison of timings for SCT and SCP analysis (concluded)

cf.	*program*	*query*	*scc*	*arcs*	*res*	*sct*	*scp*
B+	read.pl	`goal`	5	3	N	26	17
B+	serialize.pl	`serialize0(b,f)`	4	6	Y	17	10
B+	tak.pl	`tak(b,b,b,f)`	1	6	N	10	7
B+	tictactoe.pl	`play(f)`	3	19	N	32	23
B+	warplan.pl	`test2`	11	3	N	22	17
B+	zebra.pl	`zebra(f,f,f,f,f,f,f)`	2	2	Y	6	4
B+	zebra.pt	`houses(f)`	2	2	Y	6	4
LS	p	`p`	1	0	N	1	1
LS	sublist	`sublist(b,b)`	1	0	N	2	1
LS	sicstus1	`concatenate(b,b,f)`	1	2	Y	5	3
LS	sicstus1	`concatenate(b,f,f)`	1	1	Y	3	2
LS	sicstus1	`concatenate(f,f,b)`	1	1	Y	3	2
LS	sicstus1	`member(f,b)`	1	1	Y	4	2
LS	sicstus1	`reverse(b,f)`	1	1	Y	3	2
LS	sicstus1	`concatenate(f,b,f)`	1	1	N	2	2
LS	sicstus1	`member(b,f)`	1	1	N	3	2
LS	sicstus1	`reverse(f,b)`	1	1	N	2	2
LS	sicstus2	`descendant(b,b)`	1	2	N	3	3
LS	sicstus3	`put_assoc(b,b,b,f)`	1	12	Y	52	13
LS	sicstus3	`get_assoc(b,b,f)`	1	8	Y	35	10
LS	sicstus4	`d(b,b,f)`	1	2	Y	5	3
LS	ack	`ack(b,b,f)`	1	3	Y	12	6
LS	pql	`p(b,f)`	1	2	Y	9	5
LS	pql	`q(b,f)`	1	3	Y	11	6
LS	pql	`q(f,b)`	1	3	Y	11	6
LS	pql	`p(f,b)`	1	0	N	3	2
LS	pql	`q(f,f)`	1	0	N	3	2
LS	vangelder	`q(b,b)`	1	24	Y	236	140
LS	deep_rev	`deep(b,f)`	1	3	Y	12	6
LS	huffman	`huffman(b,f)`	2	2	Y	6	4
LS	huffman	`code(b,f,f)`	1	1	Y	3	2
LS	queens	`queens(b,f)`	4	5	Y	15	9
LS	arit_exp	`e(b)`	1	7	Y	36	10
LS	associative	`normal_form(b,f)`	2	2	Y	6	4
LS	game	`win(b)`	1	1	Y	3	2
LS	yale_s_p	`holds(b,b)`	1	8	Y	23	9
LS	yale_s_p	`holds(f,b)`	1	4	Y	15	7
LS	yale_s_p	`holds(b,f)`	1	3	N	5	4
LS	plan	`transform(b,b,f)`	4	7	N	19	12
LS	blocks	`tower(b,b,b,f)`	2	1	N	4	3
LS	credit	`credit(b,f)`	4	9	Y	21	12

A. HARDNESS OF FINDING A FAN-IN FREE THREAD PRESERVER

This appendix provides the proof for a statement made in Section 3.

THEOREM A.1. *The set of size-change-annotated control-flow graphs that possess a non-empty, fan-in free thread preserver is NP-complete.*

PROOF. Verifying that a subset P of parameters constitutes a fan-in free thread preserver is a straight-forward, efficient procedure. Inclusion of the problem in NP is, therefore, obvious. For NP-hardness, we show a reduction from the well-known problem of *3-coloring an undirected graph.*

Let $G = (V, E)$ be a given undirected graph. We construct an annotated control-flow graph $\mathcal{G}$ such that $\mathcal{G}$ has a non-empty, fan-in free thread preserver if and only if G is 3-colorable.

(1) The node set of $\mathcal{G}$ is $V \cup \{c\}$, where c is called *the center node.*

(2) The parameter sets associated with ACG nodes are as follows: for $v \in V$, we have $Param(v) = \{v^1, v^2, v^3\}$. Informally, these represent the three possible colorings of v. The center node has a single parameter, x_c.

(3) The arcs of $\mathcal{G}$ are: two directed arcs $u \to v$ and $v \to u$ for each edge $\{u, v\} \in E$, as well as arcs $c \to v$ and $v \to c$ for all $v \in V$.

(4) For each $v \in V$, the size-change graph for arc $c \to v$ connects x_c to each of v^1, v^2 and v^3. The size-change graph for arc $v \to c$ is exactly its transpose. Informally, the role of the center node is to force a non-empty thread-preserver to contain exactly one parameter out of $Param(v)$, for every v.

(5) The size-change graph for an ACG arc $u \to v$ connects every u parameter u^i to the two parameters v^j with $j \neq i$. Informally, this represents the coloring constraint implied by the G edge.

The correctness of the reduction is now probably obvious. □

B. CORRECTNESS OF ALGORITHM 7.1

THEOREM B.1. *Algorithm 7.1 correctly decides SCT.*

PROOF. First note that the failure condition at Step 4 is satisfied for G just when G^ω has no thread with infinite descent.

Thus, if the algorithm fails, we know that $\overline{\mathcal{H}}$ (the composition closure) contains some $G^\star : f \to f$ such that $(G^\star)^\omega$ has no infinite descent. Hence, $\mathcal{H}$ does not satisfy SCT.

Conversely, suppose that the algorithm does not fail. We first argue that it terminates. Observe that $\mathcal{S}$ never decreases. Since the size of $\mathcal{S}$ is bounded, it eventually stabilizes. Subsequently, Step 3 in the algorithm never proceeds to Step 4, so W is repeatedly reduced until it is empty.

Now assume, for a contradiction, that SCT is not satisfied. By Theorem 2.9, $\overline{\mathcal{H}}$ contains an idempotent graph $G^\star : f \to f$ without any arc of the form $x \xrightarrow{\downarrow} x$. By the definition of $\overline{\mathcal{H}}$, $G^\star = G_1; \ldots; G_n$ for some $\mathcal{H}$-multipath $G_1 \ldots G_n$. It is not difficult to prove, by induction on n, that there is some G in the final $\mathcal{S}$ such that $G \leq G^\star$.

For $(G^\star)^\omega$ to have infinite descent, it would have to contain, for some x, a thread from x to x with a strict arc. Since $G^\star$ is idempotent, this would mean $x \xrightarrow{\downarrow} x \in G^\star$, contradicting the assumption about $G^\star$. Thus, $(G^\star)^\omega$ has no infinite descent. Since G can be obtained from $G^\star$ by removing arcs and $\downarrow$ labels, G^ω has no infinite descent either. But then the algorithm should have failed on testing G at Step 4. $\square$