Program Termination Analysis in Polynomial Time

Chin Soon Lee[⋆]

Datalogisk Institut, Copenhagen University, Denmark

Abstract. Recall the "size-change principle" for program termination: An infinite computation is *impossible*, if it would give rise to an infinite sequence of size decreases for some data values. For an actual analysis, *size-change graphs* are used to capture data size changes in possible program state transitions. A graph sequence that respects program control flow is called a *multipath*. Multipaths describe possible state transition sequences. To show that such sequences are finite, it is sufficient that the graphs satisfy *size-change termination (SCT)*: Every infinite multipath has infinite descent, according to the arcs of the graphs. This is an application of the size-change principle.
SCT is decidable, but complete for PSPACE. In this paper, we explore efficient approximations that are sufficiently precise in practice.
For size-change graphs that can loop (i.e., they give rise to an infinite multipath), and that satisfy SCT, it is usually easy to identify among them an *anchor*— some graph whose infinite occurrence in a multipath causes infinite descent. The SCT approximations are based on finding and eliminating anchors as far as possible. If the remaining graphs cannot loop, then SCT is established.
An efficient method is proposed to find anchors among *fan-in free* graphs, as such graphs occur commonly in practice. This leads to a worst-case quadratic-time approximation of SCT. An extension of the method handles general graphs and can detect some rather subtle descent. It leads to a worst-case cubic-time approximation of SCT. Experiments confirm the effectiveness and efficiency of the approximations in practice.

1 Introduction

1.1 Background and Motivation of This Work

This work began as an investigation into the problem of non-termination for the offline partial evaluation of functional programs. To achieve finite function specialization, various techniques have been proposed to ensure that static variables assume boundedly many values during specialization (see [7] in this volume and references therein). The intuition behind these techniques is: Arbitrarily many increases in a static variable are *impossible*, if that would imply arbitrarily many size decreases for some bounded-variable values.

[⋆] This research is supported by DAEDALUS, the RTD project IST-1999-20527 of the IST Programme within the European Union's Fifth Framework Programme.

The conditions for variable boundedness seen in the literature are complicated. To understand better the forms of descent detected by them, Neil Jones proposed studying a related, but simpler condition for *program termination*, based on the reasoning: Infinite program execution is *impossible*, if it would give rise to an infinite sequence of size decreases for some data values.

In joint work with Jones and Ben-Amram, the *size-change termination (SCT)* condition, based on the above principle, was formulated and studied for a first-order functional language [11]. SCT is stated in terms of simple structures called *size-change graphs*. These graphs capture the data size changes in possible program state transitions (in this case, function-call state transitions), and are obtained by dataflow analysis. SCT is decidable, but complete for PSPACE. In [11], we argued that this called for "sufficiently strong" approximations. They form the topic of the present article.

Fast and scalable methods for termination analysis are not widely discussed in the literature. The need for complicated supporting analyses in a practical analyzer makes exponential-time procedures difficult to avoid. For instance, consider the Prolog analyzers Termilog [12] and Terminweb [3]. These analyzers implement procedures similar to a decision method for SCT, except that they manipulate procedure-call descriptions that capture the instantiatedness of variables as well as argument size relations. The abstract information can be exponentially large in the program size.

Interestingly, when this is *not* the case, fast analysis still cannot be guaranteed. This is because SCT can be tested using Termilog or Terminweb by a straightforward encoding. Such generality is not required in practice. In this article, we develop effective and efficient approximations of SCT. By studying fast supporting analyses, it might then be possible to design practical analyzers based on the approximations.

Below are different reasons for this research, connected to the study of automatic program manipulation.

- **Finite function specialization** requires static variables to be bounded. This can be ensured by *adapting* our techniques [9, 10, 7].
- **Termination of unfolding** during the offline partial evaluation of functional and logic programs is directly addressed by our techniques [9, 14, 7].
- **More aggressive program transformation** may be possible given the termination of certain computations. These computations need not be retained in the transformed program simply for their termination behaviour.
- **Choice of evaluation strategy** for some Prolog compilers may be guided by the results of termination analysis [13].
- **Independent verification of machine-generated programs** that are not completely trusted is desirable.

1.2 Generality of SCT

The SCT condition subsumes natural ways to reason about the termination of programs that manipulate well-founded data. It is particularly suited to functional and logic programs, for which the size relations among *inputs* and *outputs* of subcomputations can be approximated by size analysis. The following untyped functional programs, operating on lists and natural numbers, are all terminating because every infinite sequence of function-call state transitions would give rise to a sequence of parameter values whose sizes decrease infinitely. We label the function calls for easy reference.

1. **Non-recursive calls** do not affect size-change termination, although they may cause descent to begin late. Consider call 1 below.

```
rev0(zs) = ¹rev1(zs,[])
rev1(xs,acc) = if xs=[] then acc else
                  ²rev1(tl(xs),hd(xs):acc)
```

2. **Indirect recursion**, as exhibited by the following tail-recursive program to multiply two numbers, is handled naturally.

```
mult(m,n) = ¹loop1(m,0,n,0)
loop1(i1,j1,n1,a1) = if i1=0 then a1 else
                        ²loop2(i1-1,n1,n1,a1)
loop2(i2,j2,n2,a2) = if j2=0 then ³loop1(i2,j2,n2,a2) else
                        ⁴loop2(i2,j2-1,n2,a2+1)
```

3. **Nested calls** are handled by *size analysis.* Size analysis deduces that the return value of `sub(m,n)` below always has smaller size than the value of `m`.

```
quot(m,n) = if n=0 then false 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)
```

4. **Complex descent** in the *program states*: SCT induces a well-founded ordering on the program states in which every possible state transition decreases.
   (a) *Lexical descent.* The parameters of `ack` below have sizes that descend lexically in every program state transition. To show termination, program state `ack`(m, n) is ordered below `ack`(m', n') whenever the sizes of m and n is a lexically smaller pair than the sizes of m' and n'.

```
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))
```

   In Ex. 2, the sizes of `i` and `j`'s values also descend lexically in every sequence of program state transitions from `loop2` to `loop2`.
   (b) *Descent in a sum of parameter sizes.* For the next program, the number $sx + sy + sz$, where sx, sy and sz are the sizes of `x`, `y` and `z`'s values, decreases on every program state transition. Plümer identifies this as an important form of descent [13].

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

(c) *Descent in the maximum over parameter sizes.* The number $max(sx, sy)$, where sx and sy are the sizes of x and y's values, decreases on every program state transition for the following program. This descent has been observed in a type inference algorithm [5, 6].

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

(d) *Combining different descent.* It is complicated to express the descent for the next program. Such descent is not expected in practice.

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

1.3 This Article

Following our earlier paper [11], we will work with a minimal first-order functional language with eager evaluation. In Sect. 2, we present its syntax and semantics. We define program state transitions (more precisely than in [11]), and relate the existence of an infinite state transition sequence to non-termination. *Size-change graphs* are bipartite graphs that describe the data size changes in possible program state transitions. In Sect. 3, we review the *size-change termination (SCT)* condition for a program's size-change graphs. As SCT is complete for PSPACE, we study viable PTIME approximations in Sect. 4. Section 5 contains some concluding remarks.

2 Programs and Abstraction of Their Behaviour

2.1 Syntax and Notations

Programs in our minimal subject language are generated by the following grammar, where $x, x_i \in Par$, $f \in Fun$, Par and Fun are mutually exclusive sets of identifiers, and $op \in Op$ is a primitive operator.

$$\begin{array}{lll} p \in Prog & ::= & d_1 \ \ldots \ d_m \\ d, d_i \in Def & ::= & f(x_1, \ldots, x_n) = e^f \\ e, e_i, e^f \in Expr & ::= & x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \\ & & op(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n) \end{array}$$

The *definition* of function f has the form $f(x_1, \ldots, x_n) = e^f$, where e^f is called the *body* of f. The number $n \geq 0$ of parameters is its *arity*, written $ar(f)$. Notation: $Param(f) = \{f^{(1)}, \ldots, f^{(n)}\}$ is the set of f parameters. In examples they are named by identifiers. Parameters are assumed to be in scope when they are used. Call expressions are sometimes labelled, e.g., ${}^c f(e_1, \ldots, e_n)$. Without loss of generality, function names, parameter names and call labels are assumed to be distinct in a program. Constants are regarded as 0-ary operators.

2.2 Programs and Their Semantics

Programs in the subject language are untyped, and interpreted according to the call-by-value semantics in Fig. A.1. The semantic operator $\mathcal{E}$ is defined as usual for a functional language: $\mathcal{E}[\![e]\!]\boldsymbol{v}$ is the value of expression e in the environment $\boldsymbol{v} = (v_1, \ldots, v_n)$, a tuple containing the values of parameters $f^{(1)}, \ldots, f^{(n)}$.

$\mathcal{E}$ has type $\mathit{Expr} \to \mathit{Val}^* \to \mathit{Val}^\sharp$, where Val^* is the set of finite value sequences. Domain $\mathit{Val}^\sharp = \mathit{Val} \cup \{\bot, \mathit{Err}\}$ includes values, plus Err to model runtime errors, and $\bot$ to model non-termination. Function $\mathit{lift} : \mathit{Val} \to \mathit{Val}^\sharp$ is the natural injection. Any input $\boldsymbol{v}$ to f is assumed to have length $ar(f)$.

Definition 1. *(Termination) Function f of program p* terminates on input $\boldsymbol{v}$ *if $\mathcal{E}[\![e^f]\!]\boldsymbol{v} \neq \bot$. Function f is* terminating *if it terminates on all inputs. A program p is* terminating *if all of its functions are terminating.*

Definition 2. *(Program states and computations)*

1. *A* local program state *is a triple $(e, f, \boldsymbol{v}) \in \mathit{Expr} \times \mathit{Fun} \times \mathit{Val}^*$ such that e is a subexpression of e^f and $\boldsymbol{v}$ has length $ar(f)$.*
2. *For local program states $(e, f, \boldsymbol{v})$ and $(e', g, \boldsymbol{u})$, write $(e, f, \boldsymbol{v}) \rightsquigarrow (e', g, \boldsymbol{u})$ to express a* subcomputation. *Define the relation $\rightsquigarrow$ by case analysis of e.*
   - *(a)* Case $e \equiv$ **if** e_1 **then** e_2 **else** e_3
     - $(e, f, \boldsymbol{v}) \rightsquigarrow (e_1, f, \boldsymbol{v})$
     - $(e, f, \boldsymbol{v}) \rightsquigarrow (e_2, f, \boldsymbol{v})$ *if $\mathcal{E}[\![e_1]\!]\boldsymbol{v} = \mathit{lift}\ u$ where $u = \mathit{True}$.*
     - $(e, f, \boldsymbol{v}) \rightsquigarrow (e_3, f, \boldsymbol{v})$ *if $\mathcal{E}[\![e_1]\!]\boldsymbol{v} = \mathit{lift}\ u$ where $u \neq \mathit{True}$.*
   - *(b)* Case $e \equiv op(e_1, \ldots, e_n)$
     - $(e, f, \boldsymbol{v}) \rightsquigarrow (e_k, f, \boldsymbol{v})$ *if for $1 \leq i < k : \mathcal{E}[\![e_i]\!]\boldsymbol{v} \notin \{\bot, \mathit{Err}\}$.*
   - *(c)* Case $e \equiv g(e_1, \ldots, e_n)$
     - $(e, f, \boldsymbol{v}) \rightsquigarrow (e_k, f, \boldsymbol{v})$ *if for $1 \leq i < k : \mathcal{E}[\![e_i]\!]\boldsymbol{v} \notin \{\bot, \mathit{Err}\}$.*
     - $(e, f, \boldsymbol{v}) \rightsquigarrow (e^g, g, \boldsymbol{u})$ *if each $\mathcal{E}[\![e_i]\!]\boldsymbol{v} = \mathit{lift}\ u_i$ and $\boldsymbol{u} = (u_1, \ldots, u_n)$.*
3. *A* program state *is a pair $(f, \boldsymbol{v}) \in \mathit{Fun} \times \mathit{Val}^*$ where $\boldsymbol{v}$ has length $ar(f)$.*
4. *A* program state transition (or function-call state transition) *is a pair of program states $(f, \boldsymbol{v}) \hookrightarrow (g, \boldsymbol{u})$ such that there is a sequence of local program states $(e_0, f, \boldsymbol{v}) \rightsquigarrow \ldots \rightsquigarrow (e_T, f, \boldsymbol{v})$, where $e_0 = e^f$, expressions $e_0, \ldots, e_{T-1}$ are* not *function calls, e_T is a function call to g, each $(e_t, f, \boldsymbol{v}) \rightsquigarrow (e_{t+1}, f, \boldsymbol{v})$ is a local state transition, and $(e_T, f, \boldsymbol{v}) \rightsquigarrow (e^g, g, \boldsymbol{u})$.*
5. *A* state transition sequence *has the form $(f_0, \boldsymbol{v}_0) \hookrightarrow (f_1, \boldsymbol{v}_1) \hookrightarrow (f_2, \boldsymbol{v}_2) \hookrightarrow \ldots$ such that each $(f_t, \boldsymbol{v}_t) \hookrightarrow (f_{t+1}, \boldsymbol{v}_{t+1})$ is a program state transition.*

Remark: Clearly, neither $\rightsquigarrow$ nor $\hookrightarrow$ are computable relations in general. However, we will be *approximating* properties of the $\hookrightarrow$ relation for the subject program.

Proposition 1. *If program p is not terminating, there exists an infinite state transition sequence: $(f_0, \boldsymbol{v}_0) \hookrightarrow (f_1, \boldsymbol{v}_1) \hookrightarrow (f_2, \boldsymbol{v}_2) \hookrightarrow \ldots$.*

2.3 Abstraction of Program Behaviour

Definition 3. *(Data sizes)*

1. *We will* assume *a* size function $|\bullet| : \mathit{Val} \to I\!N$.
2. *For e a subexpression of e^f, we describe e as* non-increasing *on $f^{(i)}$ if for every input $\boldsymbol{v} = (v_1, \ldots, v_n)$ to f, $\mathcal{E}[\![e]\!]\boldsymbol{v} = \mathit{lift}\ u$ implies $|v_i| \geq |u|$; we describe e as* decreasing *on $f^{(i)}$ if $|v_i| > |u|$.*

Typically, the size of a list is the number of constructors in the list, and the size of a natural number is its value. With this size function, reducing the head of `ls` by computing `hd(hd(ls)):tl(ls)` is size-decreasing on `ls`.

We will assume, for our examples, that $\texttt{hd}(x)$, $\texttt{tl}(x)$ and $x\texttt{-1}$ are all size-decreasing on x. This means that `hd([])` and `tl([])` should evaluate to *Err*, rather than `[]`. A semantic consequence: Attempting to apply an unbounded number of size-decreasing operations to a value results in erroneous termination.

The composition of size-decreasing operations is size-decreasing. For instance, $\texttt{hd(tl(}x\texttt{))}$ is size-decreasing on x. An expression consisting of just the variable reference x is clearly non-increasing on x.

Definition 4. *(Size-change graphs)*

1. *A* size-change graph *Γ is formally a triple $f \xrightarrow{G} g$, where G is a bipartite graph $(\mathit{Param}(f), \mathit{Param}(g), A)$, and A is a set of arcs of the form $x \xrightarrow{r} y$ with $x \in \mathit{Param}(f)$, $y \in \mathit{Param}(g)$ and $r \in \{\downarrow, \overset{=}{\downarrow}\}$. We also loosely write $x \xrightarrow{r} y \in G$. It is assumed that $x \xrightarrow{\downarrow} y$ and $x \xrightarrow{\overset{=}{\downarrow}} y$ are not both in G.*
2. *The finite set of possible size-change graphs for program p is denoted SCG_p.*

A set of size-change graphs for Ex. 4a is depicted below. Each $\texttt{ack} \xrightarrow{G_c} \texttt{ack}$ describes any program state transition due to the function call at c. (Note that for a transition at callsite 2, the nested call at callsite 3 must have terminated successfully.) The graphs $\texttt{ack} \xrightarrow{G_1} \texttt{ack}$ and $\texttt{ack} \xrightarrow{G_2} \texttt{ack}$ (formally the same graph) capture function-call state transitions where the size of `m` is decreased, as indicated by the arc $\texttt{m} \xrightarrow{\downarrow} \texttt{m}$. The graph $\texttt{ack} \xrightarrow{G_3} \texttt{ack}$ captures function-call state transitions where the size of `m` is not increased, as indicated by the arc $\texttt{m} \xrightarrow{\overset{=}{\downarrow}} \texttt{m}$, *and* the size of `n` is decreased, as indicated by the arc $\texttt{n} \xrightarrow{\downarrow} \texttt{n}$.

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

$$\begin{tikzcd}
\texttt{m} \arrow[r, "\downarrow"] & \texttt{m} \\
\texttt{n} & \texttt{n}
\end{tikzcd}$$

$\texttt{ack} \xrightarrow{G_1} \texttt{ack}$

$$\begin{tikzcd}
\texttt{m} \arrow[r, "\downarrow"] & \texttt{m} \\
\texttt{n} & \texttt{n}
\end{tikzcd}$$

$\texttt{ack} \xrightarrow{G_2} \texttt{ack}$

$$\begin{tikzcd}
\texttt{m} \arrow[r, "\overset{=}{\downarrow}"] & \texttt{m} \\
\texttt{n} \arrow[r, "\downarrow"] & \texttt{n}
\end{tikzcd}$$

$\texttt{ack} \xrightarrow{G_3} \texttt{ack}$

Definition 5. *A size-change graph describes the program state transitions for which it is* safe.

1. *Graph $f \xrightarrow{G} g$ is* safe for $(f,(v_1,\dots,v_n)) \hookrightarrow (g,(u_1,\dots,u_m))$ *if for each* $f^{(i)} \xrightarrow{\downarrow} g^{(j)} \in G$, $|v_i| > |u_j|$, *and for each* $f^{(i)} \xrightarrow{\overline{\downarrow}} g^{(j)} \in G$, $|v_i| \geq |u_j|$.
2. *A set of graphs $\mathcal{G}$ is* safe for the program p *if for every program state transition* $(f,\boldsymbol{v}) \hookrightarrow (g,\boldsymbol{u})$, *there is a graph* $f \xrightarrow{G} g$ *in* $\mathcal{G}$ *that is safe for it.*

The size-change graphs seen earlier for `ack` are clearly safe for the program.

Sets and sequences of size-change graphs. We may regard a set of size-change graph as arcs of an *annotated digraph.*

Definition 6. *(Annotated digraph)*

1. *An* annotated digraph $D = (V, A)$ *is a pair, consisting of* vertex set V, *and* annotated-arc set A. *An* annotated arc *has the form* $u \xrightarrow{a} v$ *where* $u, v \in V$ *and* $a \in \Sigma$ *for some set of annotations* Σ. *For a set A of annotated arcs, let* $vert(A) = \{u, v \mid u \xrightarrow{a} v \in A\}$.
2. *Let* $\rightarrow_D$ *(or just $\rightarrow$ if D is clear from context) denote* single-step reachability*:* $u \rightarrow v$ *if some* $u \xrightarrow{a} v$ *is in* A. *The* reachability relation $\rightarrow^*$ *is the reflexive transitive closure of* $\rightarrow$.
3. *A* subgraph *of D is some* $D' = (V', A')$ *such that* $V' \subseteq V$ *and* $A' \subseteq A$.
4. *Annotated digraph D is* strongly-connected *if for all* $u, v \in V : u \rightarrow^* v$. *A set A of annotated arcs is* strongly-connected *if* $(vert(A), A)$ *is strongly-connected.*
5. *For a set A of annotated arcs, define $SCC(A)$ to be the set of maximal strongly-connect subsets, or* strongly-connected components (SCCs)*, of A.*

Definition 7. *An* annotated call graph (ACG) *for program p is an annotated digraph whose vertex set is Fun and whose arc set is some* $\mathcal{G} \subseteq SCG_p$.

Definition 8. *(Multipaths)*

1. *A* multipath $\mathcal{M}$ *is a non-empty, finite or infinite sequence of size-change graphs:* $f_0 \xrightarrow{G_1} f_1, \dots, f_t \xrightarrow{G_{t+1}} f_{t+1}, \dots$. *If the graphs are members of $\mathcal{G}$, then $\mathcal{M}$ is known as a $\mathcal{G}$-multipath. We also write* $f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \dots$.
2. *A* thread *th in $\mathcal{M}$ is any finite or infinite connected sequence of size-change arcs:* $x_0 \xrightarrow{r_1} x_1, \dots, x_t \xrightarrow{r_{t+1}} x_{t+1}, \dots$ *such that there exists* $T \geq 0$, *for each* $d = 1, 2, \dots : x_{d-1} \xrightarrow{r_d} x_d \in G_{T+d}$. *We also write* $th = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} x_2 \dots$.
3. *Thread th has* infinite descent *if* $r_d = \downarrow$ *for infinitely many d.*
4. *Multipath $\mathcal{M}$ has infinite descent if it has a thread with infinite descent.*
5. *A thread is complete if it starts at the beginning of the multipath, and has the same length as the multipath.*

Referring to the picture seen earlier for the size-change graphs of `ack`, consider the multipath $\texttt{ack} \xrightarrow{G_1} \texttt{ack} \xrightarrow{G_2} \texttt{ack} \xrightarrow{G_3} \texttt{ack}$. The entire picture may be regarded as depicting this multipath. Observe the complete thread: $\texttt{m} \xrightarrow{\downarrow} \texttt{m} \xrightarrow{\downarrow} \texttt{m} \xrightarrow{\overline{\downarrow}} \texttt{m}$.

Role of size analysis. To obtain a safe set of size-change graphs, we could include a graph for each callsite c, safe for the possible function-call state transitions originating there. Suppose ${}^{c}g(e_1, \ldots, e_n)$ occurs in e^f. We would include some graph $f \xrightarrow{G_c} g$. For the arcs of G_c, include $f^{(i)} \xrightarrow{\downarrow} g^{(j)}$ if e_j is decreasing on $f^{(i)}$ and $f^{(i)} \xrightarrow{\overline{\downarrow}} g^{(j)}$ if e_j is non-increasing on $f^{(i)}$, as deduced by size analysis. Without global analysis, we could obtain a safe set of graphs as follows.

Definition 9. *The* natural graphs $\mathcal{G}_p$ *for subject program* p *contains a size-change graph* $f \xrightarrow{G_c} g$ *for each call* ${}^{c}g(e_1, \ldots, e_n)$ *in* e^f. *The graph* G_c *contains the arc* $f^{(i)} \xrightarrow{\overline{\downarrow}} g^{(j)}$ *if* e_j *is* $f^{(i)}$ *and* $f^{(i)} \xrightarrow{\downarrow} g^{(j)}$ *if* e_j *is a sequence of destructive operators:* `hd`, `tl`, `• - 1` *applied to* $f^{(i)}$, *e.g.,* `hd(tl(`$f^{(i)}$`))`.

It is also *safe* to incorporate information from the conditions of `if` expressions. The code below has the structure of a program transformed by a binding-time improvement called The Trick [8]. Any transition $(\mathtt{f}, (v_1, v_2)) \hookrightarrow (\mathtt{g}, (u_1))$ is such that $|v_2| > |u_1|$ and $|v_1| = |u_1|$ (since for the transition to exist, $u_1 = \mathcal{E}[\![\mathtt{hd(s)}]\!](v_1, v_2) = \mathcal{E}[\![\mathtt{d}]\!](v_1, v_2) = v_1$). This is a natural way for "fan-ins" (multiple arcs connected to the same parameter) to arise in our framework.

Example 5.

```
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))
```

$$\begin{tikzcd}
x \arrow[r, "\downarrow"] & d \\
 & s
\end{tikzcd}
\qquad
\begin{tikzcd}
d \arrow[r, "\overline{\downarrow}"] & d \\
s \arrow[r, "\downarrow"] & s
\end{tikzcd}
\qquad
\begin{tikzcd}
d \arrow[r, "\overline{\downarrow}"] & x \\
s \arrow[ur, "\downarrow"] &
\end{tikzcd}$$

$$\mathtt{g} \xrightarrow{G_1} \mathtt{f} \qquad \mathtt{f} \xrightarrow{G_2} \mathtt{f} \qquad \mathtt{f} \xrightarrow{G_3} \mathtt{g}$$

3 Size-Change Termination

Definition 10. *A set of size-change graphs* $\mathcal{G}$ *satisfies* size-change termination (SCT) *if every infinite* $\mathcal{G}$*-multipath has infinite descent.*

Theorem 1. *Let* $\mathcal{G}$ *be safe for* p. *If* $\mathcal{G}$ *satisfies SCT, then* p *is terminating.*

Proof. Suppose that $\mathcal{G}$ satisfies SCT, but p is *not* terminating. By Proposition 1, there exists an infinite transition sequence $(f_0, \boldsymbol{v}_0) \hookrightarrow (f_1, \boldsymbol{v}_1) \hookrightarrow (f_2, \boldsymbol{v}_2) \ldots$. By the safety of $\mathcal{G}$, there exists a $\mathcal{G}$-multipath $\mathcal{M}$: $f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \ldots$ such that each $f_t \xrightarrow{G_{t+1}} f_{t+1}$ is safe for $(f_t, \boldsymbol{v}_t) \hookrightarrow (f_{t+1}, \boldsymbol{v}_{t+1})$. By SCT, $\mathcal{M}$ has a thread *th* with infinite descent: $x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} x_2 \ldots$ such that for some T, and $d = 1, 2, \ldots$, it holds that $x_{d-1} \xrightarrow{r_d} x_d \in G_{T+d}$. Define a value sequence $u_0, u_1, \ldots$ corresponding to *th*: Suppose x_i is $f^{(k)}_{T+i}$, and $\boldsymbol{v}_{T+i} = (v_1, \ldots, v_n)$, then let $u_i = v_k$. By safety, the arc from x_{d-1} to x_d implies $|u_{d-1}| > |u_d|$ if $r_d = \downarrow$, and $|u_{d-1}| \geq |u_d|$ if $r_d = \overline{\downarrow}$. Since *th* has infinite descent, infinitely many of the r_d are $\downarrow$. This implies a sequence of data values whose sizes decrease infinitely, which is impossible. □

Definition 11. *The size-change graphs $f_0 \xrightarrow{G_1} f_1$ and $f_1 \xrightarrow{G_2} f_2$ can be composed to summarize the size changes among variable values at the start and end of a state transition sequence modelled by $f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2$. Formally, the composition $f_0 \xrightarrow{G_1} f_1; f_1 \xrightarrow{G_2} f_2 = f_0 \xrightarrow{G} f_2$, where $G = G^{\downarrow} \cup G^{\overline{\downarrow}}$ and*

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

Definition 12. *For $\mathcal{G} \subseteq SCG_p$, the* composition closure *of $\mathcal{G}$, denoted $\overline{\mathcal{G}}$, is the least set satisfying: $\overline{\mathcal{G}} = \mathcal{G} \cup \{f_0 \xrightarrow{G_1} f_1; f_1 \xrightarrow{G_2} f_2 \mid f_0 \xrightarrow{G_1} f_1, f_1 \xrightarrow{G_2} f_2 \in \overline{\mathcal{G}}\}$.*

Theorem 2. *([11]) $\mathcal{G}$ satisfies SCT iff for every size-change graph $f \xrightarrow{G} f$ in $\overline{\mathcal{G}}$ such that $f \xrightarrow{G} f; f \xrightarrow{G} f = f \xrightarrow{G} f$, there exists $x \xrightarrow{\downarrow} x \in G$ for some x.*

It follows that SCT is decidable. Unfortunately, SCT is *intrinsically hard.*

Theorem 3. *([11]) SCT satisfaction is complete for* PSPACE.

4 Polynomial-Time Approximations

The PSPACE-hardness result is motivation for studying viable polynomial-time approximations. The approximations we propose are centered around the ACG.

Definition 13. *Let S be any finite set. The* infinity set *of an infinite sequence $s_1 s_2 \ldots$ of S elements is the set $\{s \mid \{i \mid s_i = s\}$ is infinite$\}$.*

Definition 14. *Let $\mathcal{H}$ be a set of size-change graphs. Call $f \xrightarrow{G} g \in \mathcal{H}$ an* anchor for *$\mathcal{H}$ if every $\mathcal{H}$-multipath whose infinity set contains $f \xrightarrow{G} g$ has infinite descent. $\mathcal{H}$* has an anchor *means it contains some graph that is an anchor for it.*

Lemma 1. *Suppose that every non-empty strongly-connected $\mathcal{H} \subseteq \mathcal{G}$ has an anchor, then $\mathcal{G}$ satisfies SCT.*

Lemma 2. *Suppose that $\mathcal{H}$ is strongly-connected, and $\mathcal{A} \subseteq \mathcal{H}$ is a non-empty set of anchors for $\mathcal{H}$. If some non-empty strongly-connected subset $\mathcal{H}^\star$ of $\mathcal{H}$ is without anchors, then $\mathcal{H}^\star$ is a subset of a member of $SCC(\mathcal{H} \setminus \mathcal{A})$.*

Lemma 2 leads to a procedure to approximate SCT for $\mathcal{G}$, if we have some means to determine anchors easily for a strongly-connected $\mathcal{H}$. The procedure *SCP* (size-change polytime) below is called with each $\mathcal{H} \in SCC(\mathcal{G})$.

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

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

The procedure terminates, as $\mathcal{H}$ is reduced on each recursive invocation. If it does not fail, argue that $\mathcal{G}$ satisfies SCT as follows. If $\mathcal{G}$ does *not* satisfy SCT, Lemma 1 guarantees a non-empty strongly-connected $\mathcal{H}^\star$ without anchors. At the start of the procedure, $\mathcal{H}$ is equal to $\mathcal{G}$, so it includes $\mathcal{H}^\star$. For each invocation of the procedure, if $\mathcal{H}$ includes $\mathcal{H}^\star$, the input to some recursive call of *SCP* is a superset of $\mathcal{H}^\star$ by Lemma 2. Therefore, if the procedure does not fail, $\mathcal{H}^\star$ is eventually discovered, and this would cause it to fail.

4.1 Deducing Anchors

For this, we introduce a few notions centered around the parameter set. Forward (backward) *thread preservers for* $\mathcal{H}$ are parameters that are always forward (backward) connected to other thread preservers when they occur among the source (target) parameters in any graph of $\mathcal{H}$.

Definition 15. *(Thread preservers) Let* $P \subseteq Par$*. For size-change graphs* $\mathcal{H}$*, define* $\overrightarrow{TP}_P(\mathcal{H})$*,* $\overleftarrow{TP}_P(\mathcal{H})$ *to be the largest sets such that:*

$$\begin{aligned}\overrightarrow{TP}_P(\mathcal{H}) = \{x \in P \mid \forall f \xrightarrow{G} g \in \mathcal{H}\cdot \\ x \in Param(f) \Rightarrow \exists\, y \in \overrightarrow{TP}_P(\mathcal{H}) \cdot x \xrightarrow{r} y \in G \textit{ for some } r\}\\ \overleftarrow{TP}_P(\mathcal{H}) = \{y \in P \mid \forall f \xrightarrow{G} g \in \mathcal{H}\cdot \\ y \in Param(g) \Rightarrow \exists\, x \in \overleftarrow{TP}_P(\mathcal{H}) \cdot x \xrightarrow{r} y \in G \textit{ for some } r\}\end{aligned}$$

The subscript P *is omitted if it is the set* Par*.* $\overrightarrow{TP}_P(\mathcal{H})$ *and* $\overleftarrow{TP}_P(\mathcal{H})$ *are called respectively the* forward and backward preservers of $\mathcal{H}$ with respect to P.

The *size-relation graph due to* $\mathcal{H}$ is the digraph with vertex set Par and arcs corresponding to all the arcs in the size-change graphs of $\mathcal{H}$.

Definition 16. *(SRG) The* size-relation graph due to $\mathcal{H}$, *denoted* $SRG_{\mathcal{H}}$*, has the annotated arcs* $\{x \xrightarrow{r}_{\Gamma} y \mid \Gamma = f \xrightarrow{G} g \in \mathcal{H}, x \xrightarrow{r} y \in G\}$.

Deducing anchors for fan-in free graphs.

Definition 17. *(Fan-in free)*

1. *A size-change graph* $f \xrightarrow{G} g$ *is fan-in free if* $x \xrightarrow{r} y, x' \xrightarrow{r'} y \in G \Rightarrow x = x'$.
2. *A set of size-change graphs* $\mathcal{H}$ *is fan-in free if its graphs are fan-in free.*

Experiments using Terminweb's size analysis 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.

Theorem 4. *Let* $\mathcal{H}$ *be fan-in free and strongly-connected. If for* $f \xrightarrow{G} g$ *in* $\mathcal{H}$*, there exists* $x \xrightarrow{\downarrow} y \in G$ *with* $x, y \in \overrightarrow{TP}(\mathcal{H})$*, then* $f \xrightarrow{G} g$ *is an anchor for* $\mathcal{H}$*.*

Proof. As $\mathcal{H}$ is strongly-connected, there is a multipath $f_0 \overset{G_1}{\to} f_1 \dots \overset{G_{n+1}}{\to} f_{n+1}$ with $f_0 = f_{n+1}$, containing *every* graph of $\mathcal{H}$. Let $P_i = Param(f_i) \cap \overrightarrow{TP}(\mathcal{H})$ for each i. By definition of $\overrightarrow{TP}(\mathcal{H})$ and the assumption that $\mathcal{H}$ is fan-in free, $|P_i| \leq |P_{i+1}|$. Therefore $|P_0| \leq \dots \leq |P_{n+1}| = |P_0|$. Let $K = |P_0|$.

Consider any *infinite* $\mathcal{H}$-multipath $\mathcal{M} = f_0 \overset{G_1}{\to} f_1 \overset{G_2}{\to} f_2 \dots$. As before, let $P_t = Param(f_t) \cap \overrightarrow{TP}(\mathcal{H})$ for each t. Then each P_t has cardinality K. It is easy to see that the arcs from each P_t to P_{t+1} form K complete threads over $\mathcal{M}$. Suppose that $f \overset{G}{\to} g$ is in the infinity set of $\mathcal{M}$. By the premise, there exists some $x \overset{\downarrow}{\to} y \in G$ with $x, y \in \overrightarrow{TP}(\mathcal{H})$. This arc appears infinitely often among the K threads. Consequently, some thread has infinite descent. □

Remark: The above proof gives insight into the form of descent inhibiting unbounded repetitions of $f \overset{G}{\to} g$ in any $\mathcal{H}$-multipath representing a state transition sequence. The sum of P_t parameter sizes is never increased, and strictly decreased for a transition described by $f \overset{G}{\to} g$, due to the decreasing arc in G.

Lemma 3. *(Ben-Amram) Let $\mathcal{H} \subseteq \mathcal{G}$ be non-empty, fan-in free and strongly-connected. And let N and M be the size of $\mathcal{G}$ and $\mathcal{H}$ respectively. Then $\overrightarrow{TP}(\mathcal{H})$ can be computed in time $O(M)$, with an $O(N^2)$ initialization performed on $\mathcal{G}$.*

The algorithm for computing $\overrightarrow{TP}(\mathcal{H})$ maintains a set of counts for the source parameters in each size-change graph of $\mathcal{H}$. These counts initially record the source parameters' out-degrees. Start by marking every parameter whose count is 0, and adding these parameters to a worklist. When processing x from the worklist, inspect those graphs with arcs connected to x. For such an arc $x' \overset{r}{\to} x$, if x' is not yet marked, reduce its count. If the count becomes 0, mark x' and insert it into the worklist. Terminate when the worklist is empty. We claim, without proof, that upon termination, the unmarked parameters form $\overrightarrow{TP}(\mathcal{H})$, and that the algorithm has $O(M)$ time complexity, with an $O(N^2)$ initialization performed on $\mathcal{G}$. (Initialization is needed to index the functions and parameters, and set up indexing structures.)

The procedure *SCP* considers each graph of $\mathcal{G}$ in no more than $|\mathcal{G}| \sim O(N)$ invocations. Each invocation of the procedure involves computing some $\overrightarrow{TP}(\mathcal{H})$, working out $\mathcal{A}$ and computing $SCC(\mathcal{H} \setminus \mathcal{A})$. All these steps have linear time complexity. Therefore we have an $O(N^2)$ procedure to approximate SCT for fan-in free $\mathcal{G}$. We refer to the resulting approximation of SCT as SCP1.

An example. Consider the graphs $\mathcal{G}$ for `ack` seen earlier. Now, $\mathcal{G}$ is strongly-connected. We work out that $\overrightarrow{TP}(\mathcal{G}) = \{\mathtt{m}\}$ (`m` is connected to `m` in each graph of $\mathcal{G}$). We deduce that $\mathtt{ack} \overset{G_1}{\to} \mathtt{ack}$ and $\mathtt{ack} \overset{G_2}{\to} \mathtt{ack}$ are anchors for $\mathcal{G}$, as they have the arc $\mathtt{m} \overset{\downarrow}{\to} \mathtt{m}$. Intuitively, this means their occurrence in the infinity set of a $\mathcal{G}$-multipath would give the multipath infinite descent due to the decreasing arc. Removing these graphs from consideration leaves $\mathcal{H} = \{\mathtt{ack} \overset{G_3}{\to} \mathtt{ack}\}$. We work out that $\overrightarrow{TP}(\mathcal{H}) = \{\mathtt{m}, \mathtt{n}\}$, and deduce that $\mathtt{ack} \overset{G_3}{\to} \mathtt{ack}$ is an anchor for $\mathcal{H}$, as it has the arc $\mathtt{n} \overset{\downarrow}{\to} \mathtt{n}$. We conclude that $\mathcal{G}$ satisfies SCT.

In general, if $\mathcal{G}$ satisfies SCP1, then in every function-call state transition sequence of the form $(f, \boldsymbol{v}) \hookrightarrow^* (f, \boldsymbol{u})$, the parameter tuples $\boldsymbol{v}$ and $\boldsymbol{u}$ exhibit lexical descent in various sums of parameter sizes. SCP1 subsumes most practical forms of descent, including those of Ex. 1 to Ex. 4b in Sect. 1.2.

General graphs considered. The above method to deduce anchors suffers from obvious defects. To begin with, there does not appear to be any simple extension to cope with fan-ins. We consider the issue in greater detail.

Example 6.

```
f(x,y) = if y=tl(x) then ¹f(x,y) else 0
```

$$\boxed{\begin{tikzcd} \mathtt{x} \arrow[r, "\bar{\downarrow}"] \arrow[dr, "\downarrow"] & \mathtt{x} \\ \mathtt{y} \arrow[r, "\bar{\downarrow}"'] & \mathtt{y} \end{tikzcd}}$$

$\mathtt{f} \xrightarrow{G_1} \mathtt{f}$

Now $\mathcal{G}$ is strongly-connected, $\overrightarrow{TP}(\mathcal{G}) = \{\mathtt{x}, \mathtt{y}\}$, and in the graph $\mathtt{f} \xrightarrow{G_1} \mathtt{f}$, G_1 contains $\mathtt{x} \xrightarrow{\downarrow} \mathtt{y}$, *but* $\mathcal{M} = \mathtt{f} \xrightarrow{G_1} \mathtt{f} \xrightarrow{G_1} \mathtt{f} \ldots$ has no infinite descent. Indeed, $\mathcal{M}$ models the state transition sequence: $(\mathtt{f}, ([1], [])) \hookrightarrow (\mathtt{f}, ([1], [])) \hookrightarrow \ldots$. Intuitively, when size-change graphs are not fan-in free, a descending arc between $\overrightarrow{TP}$ parameters does not necessarily constitute "progress." In the example, the arc $\mathtt{x} \xrightarrow{\downarrow} \mathtt{y}$ in G_1 indicates a size relationship between the values of $\mathtt{x}$ and $\mathtt{y}$ that persists throughout the state transition sequence.

It seems sensible to consider the thread behaviour in $\{\mathtt{x}\}$ and $\{\mathtt{y}\}$ separately. A little reflection reveals that arcs not in the same SCC of the SRG due to $\mathcal{G}$ cannot give rise to anchor graphs. Therefore, we might define the following *critical parameter sets*, to be inspected independently for descent.

Definition 18. *Define the* critical parameter sets *for size-change graphs $\mathcal{H}$ as: $\mathit{Critical}(\mathcal{H}) = \{P \mid \overleftarrow{TP}_P(\mathcal{H}) = P,\ P$ strongly-connected in $SRG_{\mathcal{H}},\ P$ maximal$\}$.*

If P is a critical set of $\mathcal{H}$, then at the end of a state transition sequence described by any $\mathcal{H}$-multipath, the value of each P parameter has size no greater than the initial value of *some* P parameter. We will look for descent among such parameters. Observe that for fan-in free graphs, $\overleftarrow{TP}(\mathcal{H})$ at least includes $\overrightarrow{TP}(\mathcal{H})$. The use of a larger set of parameters makes it possible to detect more descent, but also means more complicated deductions (see Theorem 5).

Deducing anchors for general graphs.

Theorem 5. *Let $\mathcal{H}$ be strongly-connected. Then $f \xrightarrow{G} g$ in $\mathcal{H}$ is an anchor for $\mathcal{H}$ if there exists $P \in \mathit{Critical}(\mathcal{H})$ such that one of the following holds.*

1. *For $f_0 \xrightarrow{G_1} f_1$ in $\mathcal{H}$, if $x \xrightarrow{r} y, x \xrightarrow{r'} y' \in G_1$ and $x, y, y' \in P$, then $y = y'$ (i.e. no "fan-outs" among P elements)* and *there exists $x \xrightarrow{\downarrow} y \in G$ with $x, y \in P$.*
2. *The digraph with arcs $\{x \xrightarrow{\bar{\downarrow}}_{\Gamma} y \in A \mid x, y \in P\}$, where A are $SRG_{\mathcal{H}}$ arcs, has no strongly-connected subgraph with an arc $x \xrightarrow{\bar{\downarrow}}_{\Gamma} y$ where $\Gamma = f \xrightarrow{G} g$.*

Refer to Appendix B for a proof. The first condition detects the same form of descent as the condition for fan-in free graphs in Theorem 4. When the backward thread-preserving P contains fan-outs, it is difficult to know whether the decreasing arcs among P parameters are significant for descent. The second condition implements a weak principle for identifying anchors: Any infinite $\mathcal{H}$-multipath has an infinite thread *th* remaining in just the P parameters (easily shown), so if there are no *non-descending* threads remaining infinitely among $x \in P$ in any multipath whose infinity set has $f \overset{G}{\rightarrow} g$, then *th* is infinitely descending in any multipath whose infinity set *does* contain $f \overset{G}{\rightarrow} g$, i.e., this graph is an anchor for $\mathcal{H}$. This reasoning is effective when there are sufficiently many decreasing arcs among $x \in P$. For example, the natural graphs of Ex. 4c in Sect. 1.2 are handled. Note the rather subtle descent in this example.

Lemma 4. *Let $\mathcal{H} \subseteq \mathcal{G}$ be non-empty and strongly-connected. Let N and M be the size of $\mathcal{G}$ and $\mathcal{H}$ respectively. Then $Critical(\mathcal{H})$ can be computed in time $O(M)$ for fan-in free $\mathcal{H}$ and $O(M^2)$ in general, with an $O(N^2)$ initialization performed on $\mathcal{G}$.*

The critical sets of $\mathcal{H}$ are computed as follows. Initialize $\wp$ to $\{\overleftarrow{TP}(\mathcal{H})\}$. In the iterative step, replace each $P \in \wp$ with possibly several non-empty $\overleftarrow{TP}_{P'}(\mathcal{H})$, where P' are strongly-connected components in $SRG_{\mathcal{H}}$ restricted to P. This is repeated until no further change occurs. The calculation of $\overleftarrow{TP}_{P'}(\mathcal{H})$ is similar to the calculation of $\overrightarrow{TP}(\mathcal{H})$. The total $\overleftarrow{TP}$ calculations for all P' in the current $\wp$ can be accomplished in time $O(M)$, so the critical sets of $\mathcal{H}$ can be worked out in time $O(M^2)$, as there are at most M iterative steps. The required initialization can be performed in time $O(N^2)$. To see that critical sets can be computed in time $O(M)$ for fan-in free graphs, we require the following.

Lemma 5. *Let $\mathcal{H}$ be non-empty,* fan-in free *and strongly-connected. Suppose that $P = \overleftarrow{TP}(\mathcal{H})$ and P' is a strongly-connected component in $SRG_{\mathcal{H}}$ restricted to P. Then either $\overleftarrow{TP}_{P'}(\mathcal{H}) = \{\}$ or $\overleftarrow{TP}_{P'}(\mathcal{H}) = P'$.*

Refer to Appendix B for a proof. Now it follows from Lemma 5 that no iteration of $\overleftarrow{TP}_{P'}(\mathcal{H})$ calculations is necessary for fan-in free graphs. The computation of critical sets is the dearest part of the procedure to deduce anchors. The conditions of Theorem 5 can be checked in time $O(M)$. It follows that the *SCP* procedure can be performed in time $O(N^2)$ for fan-in free graphs, and $O(N^3)$ in general. We refer to the resulting approximation of SCT as SCP2.

An example with fan-ins. Consider the graphs $\mathcal{G}$ of Ex. 5 seen earlier. The initial step to determine anchors for $\mathcal{G}$ finds the only critical set $\{\mathtt{x}, \mathtt{d}\}$. This blocks out $\mathtt{s}$, which cannot contribute to infinite descent in any multipath whose infinity set contains $\mathtt{g} \overset{G_1}{\rightarrow} \mathtt{f}$ or $\mathtt{f} \overset{G_3}{\rightarrow} \mathtt{g}$ (any thread visiting $\mathtt{s}$ at some point in such a multipath is eventually continued to the $\{\mathtt{x}, \mathtt{d}\}$ component of the SRG, never to return to $\mathtt{s}$). By either condition in Theorem 5, $\mathtt{g} \overset{G_1}{\rightarrow} \mathtt{f}$ is determined to be an anchor for $\mathcal{G}$. Next, consider the strongly-connected $\{\mathtt{f} \overset{G_2}{\rightarrow} \mathtt{f}\}$. The critical parameter sets here are $\{\mathtt{d}\}$ and $\{\mathtt{s}\}$. The second critical set allows us to conclude that $\mathtt{f} \overset{G_2}{\rightarrow} \mathtt{f}$ is an anchor. It follows that $\mathcal{G}$ satisfies SCT.

4.2 Assessment of the SCT Approximations

Lemma 6. *Let $\mathcal{H}$ be non-empty,* fan-in free *and strongly-connected. Then for $x, y \in \overrightarrow{TP}(\mathcal{H})$, if there exists $f \overset{G}{\to} g$ in $\mathcal{H}$ such that $x \overset{r}{\to} y \in G$ for some r, there is a critical set P such that $x, y \in P$ and $\overrightarrow{TP}_P(\mathcal{H}) = P$.*

Refer to Appendix B for a proof.

Lemma 7. *Let SCT, SCP1 and SCP2 denote those $\mathcal{G}$ satisfying SCT, SCP1 and SCP2 respectively. Then SCP1 $\subset$ SCP2 $\subset$ SCT.*

Proof. It follows quite easily from Lemma 6 that any anchor deduced for a strongly-connected, fan-in free $\mathcal{H}$ by the condition of Theorem 4 is also deduced to be an anchor by the first condition of Theorem 5. Thus, *SCP1* $\subseteq$ *SCP2*. The other containment *SCP2* $\subseteq$ *SCT* is clear. The containments are strict: SCP1 does not handle the natural graphs for Ex. 4c of Sect. 1.2, whereas SCP2 does; SCP2 does not handle the natural graphs for Ex. 4d, whereas SCT does. □

Experimental results. Finally, we report some empirical findings. The size-change graphs used for our experiments have been generated using Terminweb's size analysis [3], applied to the test suites in [13, 1, 2, 4, 12]. For each program, a set of size-change graphs $\mathcal{G}$ is compiled using either the *list-length norm (ll)* or the *termsize norm (ts)*. Instead of timing the analysis of each $\mathcal{G}$, we consider their SCCs, since our analyses scale linearly in this number. For each SCC, we time 1000 runs each of the SCT, SCP1 and SCP2 tests. The table in Fig. C.1 shows the benchmark reference, program name, size norm used, total number of arcs in the tested component's graphs (the problem size), results and timings of the various tests, and names of some procedures in the component.

A total of 103 programs with 281 components have been analyzed. For our experiments, the SCP2 test has given the same results as SCT in all but one instance, a contrived example due to Van Gelder. Apart from 19 components where the SCP1 test does not apply due to fan-ins (this situation is indicated by - in the table), it has also given the same results, suggesting that SCT is "overkill" in practice. Timing-wise, SCP1 is consistently and significantly faster than SCT, when it is applicable. For 19 small components, SCP2 is actually slower than SCT (probably due to its more difficult implementation), but it gives up quickly on complicated examples that eventually cause SCT to fail. For our table, we have simply ordered the experiments according to problem size, and listed the results at the top.

5 Concluding Remarks

We have reviewed the size-change condition for program termination. In a previous article, we established that SCT is complete for PSPACE. In this article, we have explored, theoretically and empirically, polynomial-time approximations of SCT. In particular, we have proposed two tests that may establish SCT. The first is simpler but restricted to fan-in free graphs. It has worst-case quadratic time complexity and is very fast in practice. The general test has worst-case cubic time complexity and accurately predicts SCT in practice.

There remain a number of obstacles to the analysis of “real programs.”

– **For functional programs**: We require a convincing extension of the basic approach to higher-order programs.
– **For Prolog programs**: Fast supporting analyses are needed. They have to be combined with SCP2 sensibly to produce a useful tool for Prolog.
– **For imperative programs**: Research is on-going. It is hard to deal with imperative programs, as they do not use well-founded data. Further, sequential programming encourages nested loops that maintain complex invariants among variable values. This makes the job of size analysis difficult.

References

1. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
2. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, 1994.
3. Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97–HOA '97, Southampton, U.K., Sep 3–5, 1997*, volume 1298 of *LNCS*, pages 31–45. Springer, 1997.
4. D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *The Journal of Logic Programming*, 19–20:199–260, 1994.
5. Carl C. Frederiksen. SCT analyzer. At `http://www.diku.dk/~xeno/sct.html`.
6. Carl C. Frederiksen. A simple implementation of the size-change principle. Technical Report D-442, Datalogisk Institut Københavns Universitet, 2001.
7. Neil Jones and Arne Glenstrup. Program generation, termination, and binding-time analysis. In *Principles, Logics, and Implementations of High-Level Programming Languages, PLI 2002 (invited paper)*. Springer, 2002.
8. Chin Soon Lee. Partial evaluation of the euclidean algorithm, revisited. *Higher-Order and Symbolic Computation*, 12(2):203–212, Sep 1999.
9. Chin Soon Lee. *Program Termination Analysis, and Termination of Offline Partial Evaluation*. PhD thesis, UWA, University of Western Australia, Australia, 2001.
10. Chin Soon Lee. Finiteness analysis in polynomial time. In M. Hermenegildo and G. Puebla, editors, *Proceedings of Static Analysis Symposium*, volume 2477 of *LNCS*. Springer, 2002.
11. Chin Soon Lee, Neil D. Jones, and Amir Ben-Amram. The size-change principle for program termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages, Jan 2001*. ACM, 2001.
12. Naomi Lindenstrauss and Yehohua Sagiv. Automatic termination analysis of logic program (with detailed experimental results). Article available at `http://www.cs.huji.ac.il/~naomil/`, 1997.
13. Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *LNAI*. Springer-Verlag, 1990.

14. Wim Vanhoof and Maurice Bruynooghe. Binding-time annotations without binding-time analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), 8th International Conference, Havana, Cuba, Dec 3-7, 2001*, volume 2250 of *LNCS*, pages 707–722, 2001.

A Program Semantics

Programs are interpreted according to the semantic operator $\mathcal{E}$ of Fig. A.1.

$$
\boxed{
\begin{array}{ll}
\text{Domains} & u, v, v_i \in \mathit{Val} \\
 & w, w_i \in \mathit{Val}^\sharp = \mathit{Val} \cup \{\bot, \mathit{Err}\}, \text{ where } \bot \sqsubseteq w \text{ for all } w. \\
 & \\
\text{Types} & \begin{array}{ll}
\mathcal{E} & : \mathit{Expr} \to \mathit{Val}^* \to \mathit{Val}^\sharp \\
\mathcal{O} & : \mathit{Op} \to \mathit{Val}^* \to \mathit{Val}^\sharp \\
\mathit{lift} & : \mathit{Val} \to \mathit{Val}^\sharp \text{ (the natural injection)} \\
\mathit{strictapp} & : (\mathit{Val}^* \to \mathit{Val}^\sharp) \to (\mathit{Val}^\sharp)^* \to \mathit{Val}^\sharp
\end{array} \\
 & \\
\begin{array}{l}\text{Semantic}\\ \text{operator}\end{array} & \begin{array}{ll}
\mathcal{E}[\![f^{(i)}]\!](v_1, \ldots, v_n) & = \mathit{lift}\ v_i \\
\mathcal{E}[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\boldsymbol{v} & = \mathcal{E}[\![e_1]\!]\boldsymbol{v} \to \mathcal{E}[\![e_2]\!]\boldsymbol{v}, \mathcal{E}[\![e_3]\!]\boldsymbol{v} \\
\mathcal{E}[\![op(e_1, \ldots, e_n)]\!]\boldsymbol{v} & = \mathit{strictapp}\ (\mathcal{O}[\![op]\!])\ (\mathcal{E}[\![e_1]\!]\boldsymbol{v}, \ldots, \mathcal{E}[\![e_n]\!]\boldsymbol{v}) \\
\mathcal{E}[\![f(e_1, \ldots, e_n)]\!]\boldsymbol{v} & = \mathit{strictapp}\ (\mathcal{E}[\![e^f]\!])\ (\mathcal{E}[\![e_1]\!]\boldsymbol{v}, \ldots, \mathcal{E}[\![e_n]\!]\boldsymbol{v})
\end{array} \\
 & \\
\text{Auxiliary} & w_1 \to w_2, w_3 = \\
 & \quad \begin{cases} w_1, & \text{if } w_1 \in \{\bot, \mathit{Err}\}, \\ w_2, & \text{if } w_1 = \mathit{lift}\ u \text{ where } u = \mathit{True}, \\ w_3, & \text{if } w_1 = \mathit{lift}\ u \text{ where } u \neq \mathit{True}. \end{cases} \\
 & \\
 & \mathit{strictapp}\ \psi\,(w_1, \ldots, w_n) = \\
 & \quad \begin{cases} \psi(v_1, \ldots, v_n) & \text{if each } w_i = \mathit{lift}\ v_i;\text{ else} \\ w_i & \text{where } i = \text{ least index such that } w_i \in \{\bot, \mathit{Err}\} \end{cases} \\
 & \\
\text{Assume} & \mathcal{O}[\![op]\!]\boldsymbol{v} \neq \bot
\end{array}
}
$$

Figure A.1: Program semantics. True is a distinguished element of Val

B Proofs of Theorems and Lemmas

Theorem 5. *Let $\mathcal{H}$ be a strongly-connected. Then $f \xrightarrow{G} g \in \mathcal{H}$ is an anchor for $\mathcal{H}$ if there exists $P \in \mathit{Critical}(\mathcal{H})$ such that one of the following holds.*

1. *For every $f_0 \xrightarrow{G_1} f_1 \in \mathcal{H}$, if $x \xrightarrow{r} y, x \xrightarrow{r'} y' \in G_1$ and $x, y, y' \in P$, then $y = y'$ (i.e., no fan-outs among P parameters),* and *there exists some $x \xrightarrow{\downarrow} y \in G$ with $x, y \in P$.*

2. *The digraph D with arcs $\{x \xrightarrow{\downarrow}_{\Gamma} y \in A \mid x, y \in P\}$, where A are $SRG_{\mathcal{H}}$ arcs, has no strongly-connected subgraph with an arc $x \xrightarrow{\downarrow}_{\Gamma} y$ where $\Gamma = f \xrightarrow{G} g$.*

Proof. The proof that Condition 1 implies $f \xrightarrow{G} g$ is an anchor for $\mathcal{H}$ is omitted because it is similar to the proof of Theorem 4. As for Condition 2, suppose that D has no strongly-connected subgraph with an arc of the form $x \xrightarrow{\downarrow}_{\Gamma} y$ where $\Gamma = f \xrightarrow{G} g$, *but* $f \xrightarrow{G} g$ is not an anchor of $\mathcal{H}$. We show this leads to a contradiction.

Since $f \xrightarrow{G} g$ is not an anchor of $\mathcal{H}$, some $\mathcal{H}$ multipath whose infinity set contains $f \xrightarrow{G} g$ has no infinite descent. Let $\mathcal{M} = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \ldots$ be such a multipath. Since $P \in \mathit{Critical}(\mathcal{H})$, by definition, $P = \overleftarrow{TP}_P(\mathcal{H})$, so a P target parameter is always connected to a P source parameter in each graph of $\mathcal{H}$. This means that in every finite $\mathcal{H}$-multipath, we can find a complete thread remaining in just P parameters. In particular, every finite initial section of $\mathcal{M}$ has such a thread. Let Th_0 be the set of such threads for all finite initial sections of $\mathcal{M}$. We will deduce the existence of an *infinite* thread in $\mathcal{M}$ remaining in just P parameters.

Since Th_0 is infinite, and there are only finitely many P parameters among parameters of f_0, an infinite subset of Th_0 must start with some $x_0 \in P$. Let Th_1 be the set of threads $p_1 \xrightarrow{r_2} p_2 \xrightarrow{r_3} \ldots$ such that $x_0 \xrightarrow{r_1} p_1 \xrightarrow{r_2} p_2 \xrightarrow{r_3} \ldots$ is a thread in Th_0. Then Th_1 is an infinite set of (finite) threads starting from $f_1 \xrightarrow{G_2} f_2$ in $\mathcal{M}$. Repeating the argument, there exists $x_1 \in P$ such that infinitely many threads of Th_1 start with x_1. Continuing inductively, there exist $x_2, x_3, \ldots$, all in P, connecting up to give an infinite thread of $\mathcal{M}$. (The reader may recognize that this is just König's Lemma.)

Let $th = x_0 \xrightarrow{r_1} x_1 \xrightarrow{r_2} x_2 \ldots$ be an infinite thread of $\mathcal{M}$, where $x_0, x_1, \ldots \in P$. Since $\mathcal{M}$ has no infinite descent, eventually all r_t is $\overline{\downarrow}$. Clearly, there exists a section of th, $x_i \xrightarrow{r_{i+1}} x_{i+1} \ldots x_{j-1} \xrightarrow{r_j} x_j$ with $i < j$, such that $x_i = x_j$, $r_{i+1} = \ldots = r_j = \overline{\downarrow}$, and for some $i < t \leq j$, $f_{t-1} \xrightarrow{G_t} f_t = f \xrightarrow{G} g$. (Recall that, by assumption, $f \xrightarrow{G} g$ is in the infinity set of $\mathcal{M}$.) Now observe that the sequence of size-change arcs induces a cycle in D, containing the arc $x_{t-1} \xrightarrow{\downarrow}_{\Gamma} x_t$ where $\Gamma = f \xrightarrow{G} g$. This contradicts our starting assumption. □

Lemma 5. *Let $\mathcal{H}$ be non-empty,* fan-in free *and strongly-connected. Suppose that $P = \overleftarrow{TP}(\mathcal{H})$ and P' is a strongly-connected component in $SRG_{\mathcal{H}}$ restricted to P. Then either $\overleftarrow{TP}_{P'}(\mathcal{H}) = \{\}$ or $\overleftarrow{TP}_{P'}(\mathcal{H}) = P'$.*

Proof. If in $SRG_{\mathcal{H}}$ restricted to P, there does not exist $x \xrightarrow{r}_{\Gamma} y$ such that $x \notin P'$ and $y \in P'$, then since in any graph of $\mathcal{H}$, a target P parameter is backward connected to a source P parameter (by definition of $\overleftarrow{TP}$), we deduce that any target P' parameter is backward connected to a source P' parameter. In this case, $\overleftarrow{TP}_{P'}(\mathcal{H}) = P'$.

Now suppose in $SRG_{\mathcal{H}}$ restricted to P, there exists $x \xrightarrow{r}_{\Gamma} y$ such that $x \notin P'$ and $y \in P'$. Let $\Gamma = f \xrightarrow{G} g$. Since $\mathcal{H}$ is fan-in free, x is the only parameter

connected to y in G. In this case, y cannot be a backward preserver wrt P', since $x \notin P'$. Furthermore, if y is connected to some z in the restricted $SRG_{\mathcal{H}}$ by an arc $y \xrightarrow{r'}_{\Gamma'} z$, then z too is not a backward preserver wrt P'. By induction on the length of a path from y to any vertex $p \in P'$ within the restricted $SRG_{\mathcal{H}}$, it can be proved that p is not a backward preserver wrt P'. Thus, $\overleftarrow{TP}_{P'}(\mathcal{H}) = \{\}$.
□

Lemma 6. *Let $\mathcal{H}$ be non-empty,* fan-in free *and strongly-connected. Then for $x, y \in \overrightarrow{TP}(\mathcal{H})$, if there exists $f \xrightarrow{G} g \in \mathcal{H}$ such that $x \xrightarrow{r} y \in G$ for some r, there is a critical set P such that $x, y \in P$ and $\overrightarrow{TP}_P(\mathcal{H}) = P$.*

Proof. Some observations:

1. A critical set of $\mathcal{H}$ does not mix $\overrightarrow{TP}(\mathcal{H})$ parameters with non $\overrightarrow{TP}(\mathcal{H})$ parameters: Otherwise, for some $f_0 \xrightarrow{G_1} f_1 \in \mathcal{H}$, there exists an arc in $SRG_{\mathcal{H}}$: $x \xrightarrow{r}_{\Gamma} y$ with $\Gamma = f_0 \xrightarrow{G_1} f_1$, such that $x \notin \overrightarrow{TP}(\mathcal{H})$ and $y \in \overrightarrow{TP}(\mathcal{H})$. This means that $x \xrightarrow{r} y \in G_1$. However, by the argument in the proof of Theorem 4, $\overrightarrow{TP}(\mathcal{H})$ target parameters are connected to $\overrightarrow{TP}(\mathcal{H})$ source parameters in every graph of $\mathcal{H}$, so there is some $x' \in \overrightarrow{TP}(\mathcal{H})$ such that $x' \xrightarrow{r'} y \in G_1$. Since $x \neq x'$, $\mathcal{H}$ could not be fan-in free.
2. For $x, y \in \overrightarrow{TP}(\mathcal{H})$ such that $f_0 \xrightarrow{G_1} f_1 \in \mathcal{H}$ and $x \xrightarrow{r} y \in G_1$, for any multipath $\mathcal{M}$ starting with $f_0 \xrightarrow{G_1} f_1$ and ending at f_0, sufficient repeats of $\mathcal{M}$ has a complete thread starting with $x \xrightarrow{r} y$ and ending at x: Otherwise, there would be threads from x to some z, and from z to z over $\mathcal{M}$, so $\mathcal{H}$ could not be fan-in free.

For a strongly-connected component P of $SRG_{\mathcal{H}}$ restricted to $\overrightarrow{TP}(\mathcal{H})$, we claim that $\overrightarrow{TP}_P(\mathcal{H}) = \overleftarrow{TP}_P(\mathcal{H}) = P$ and there are no inter-component arcs: Observation 2 implies that every arc in $SRG_{\mathcal{H}}$ restricted to $\overrightarrow{TP}(\mathcal{H})$ is in a cycle, so inter-component arcs are impossible. It follows from the fact $\overrightarrow{TP}(\mathcal{H})$ is both a set of forward preservers and a set of backward preservers, that for each component P, $\overrightarrow{TP}_P(\mathcal{H}) = \overleftarrow{TP}_P(\mathcal{H}) = P$.

By the claim and Observation 1, each strongly-connected component P of $SRG_{\mathcal{H}}$ restricted to $\overrightarrow{TP}(\mathcal{H})$ is a critical set. Thus, if for $x, y \in \overrightarrow{TP}(\mathcal{H})$, we have $f \xrightarrow{G} g \in \mathcal{H}$ such that $x \xrightarrow{r} y \in G$, then x, y must be in some critical set P, and $\overrightarrow{TP}_P(\mathcal{H}) = P$.
□

C Results of Experiments

As instantiatedness information is not accounted for, it is *ground termination behaviour* that is analyzed. A positive SCT result for a component does not indicate the termination of procedures in the component. It indicates that no looping occurs among these procedures. The ground termination of particular procedures can be recovered from this information and the call graph. SCT termination deductions are comparable with Termilog and Terminweb's.

Ref	program	$\lvert\cdot\rvert$	size	*SCT* (ms)		*SCP1* (ms)		*SCP2* (ms)		procedure(s)
[2]	`read`	ts	56	N	1103.53	-	-	N	4.68	`read exprtl` ...
[2]	`qplan`	ts	28	N	25.10	-	-	N	2.35	`schedule plan` ...
[2]	`tictactoe`	ts	27	N	12.37	-	-	N	1.99	`alpha_beta` ...
[12]	`vangelder`	ts	24	Y	59.66	-	-	N	2.37	`q t r p`
[2]	`rdtok`	ts	23	N	68.94	N	1.22	N	2.49	`read_tokens` ...
[2]	`warplan`	ts	22	N	13.25	-	-	N	1.62	`plan achieve` ...
[12]	`sicstus3`	ts	15	N	8.89	N	0.51	N	1.20	`put_assoc`
[2]	`rdtok`	ts	15	N	7.49	-	-	N	1.17	`read_string` ...
[2]	`aiakl`	ts	15	Y	7.47	Y	0.04	Y	0.07	`intersect`
[2]	`qplan`	ts	14	Y	1.96	Y	0.17	Y	0.36	`incorporate` ...
[2]	`serialize`	ts	12	Y	1.66	Y	0.02	Y	0.05	`split0`
[2]	`ann`	ts	12	Y	9.67	-	-	Y	0.08	`collect_info`
[12]	`sicstus3`	ts	11	N	7.00	N	0.45	N	0.95	`get_assoc`
[2]	`browse`	ts	11	N	20.19	N	0.23	N	1.10	`get_pats`
[2]	`browse`	ts	10	N	20.94	N	0.34	N	0.83	`init`
[2]	`ann`	ts	9	Y	2.57	Y	0.03	Y	0.09	`intersect`
[2]	`ann`	ts	9	Y	2.51	Y	0.03	Y	0.08	`merge`
[12]	`yale_s_p`	ll	8	Y	1.40	-	-	Y	0.23	`ab holds`
[2]	`warplan`	ts	8	Y	0.96	Y	0.03	Y	0.05	`retrace1`
[2]	`qplan`	ts	8	Y	0.52	-	-	Y	0.08	`best_goal`
[2]	`hanoiapp`	ts	8	Y	19.47	Y	0.03	Y	0.07	`shanoi`
[2]	`ann`	ts	8	N	2.49	N	0.39	N	0.88	`un_number_vars_2`
[12]	`arit_exp`	ts	7	Y	18.52	Y	0.14	Y	0.43	`e g f`
[13]	`pl8.4.2`	ts	7	Y	6.38	Y	0.17	Y	0.50	`e n t`
[2]	`tak`	ts	7	N	4.00	N	0.38	N	0.78	`tak`
[2]	`rdtok`	ts	7	N	7.93	N	0.29	N	0.70	`read_digits`
[2]	`qplan`	ts	7	N	5.86	N	0.38	N	0.88	`variablise`
[2]	`qplan`	ts	7	N	3.85	N	0.38	N	0.82	`quantificate`
[2]	`qplan`	ts	7	N	5.25	-	-	N	0.78	`cost`
[2]	`boyer`	ts	7	N	1.72	N	0.38	N	0.81	`tautology`
[12]	`ack`	ts	6	Y	1.99	Y	0.02	Y	0.06	`ack`
[13]	`mergesort_t`	ll	6	Y	0.56	Y	0.09	Y	0.23	`split2 split`
[2]	`warplan`	ts	6	N	1.93	N	0.28	N	0.62	`mkground` ...
[2]	`tictactoe`	ts	6	N	0.60	N	0.16	N	0.74	`choose legal`
[2]	`qplan`	ts	6	Y	2.74	Y	0.03	Y	0.08	`mark`
[2]	`qplan`	ts	6	N	5.92	N	0.37	N	0.85	`variables`
[2]	`qplan`	ts	6	Y	0.64	Y	0.04	Y	0.10	`instantiate` ...
[2]	`peephole`	ll	6	Y	2.56	Y	0.04	Y	0.20	`popt3 popt31`
[2]	`peephole`	ll	6	Y	1.01	Y	0.09	Y	0.21	`popt1a popt1a1`
[2]	`peephole`	ll	6	Y	1.01	Y	0.04	Y	0.09	`popt2 popt21`
[2]	`hanoiapp`	ts	6	N	16.62	N	0.20	N	0.73	`shanoi`
[2]	`fib_t`	ts	6	Y	1.70	Y	0.03	Y	0.07	`add`
[2]	`ann`	ts	6	N	5.99	N	0.39	N	0.87	`numbervars_2`
[2]	`ann`	ts	6	Y	0.94	Y	0.03	Y	0.09	`subtract`

Figure C.1: Experiments conducted on a P3/800 running Redhat Linux 6.1