The Size-Change Principle for Program Termination

Chin Soon Lee*
Department of Computer Science and Software Engineering
The University of Western Australia
Nedlands 6907
Western Australia
leecs@cs.uwa.edu.au

Neil D. Jones
Datalogisk Institut
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen
Denmark
neil@diku.dk

Amir M. Ben-Amram
Academic College of Tel-Aviv–Yaffo
4 Antokolsky Street
Tel-Aviv 64044
Israel
amirben@mta.ac.il

ABSTRACT

The "size-change termination" principle for a first-order functional language with well-founded data is: a program terminates on all inputs if *every infinite call sequence* (following program control flow) would cause an infinite descent in some data values.

Size-change analysis is based only on local approximations to parameter size changes derivable from program syntax. The set of infinite call sequences that follow program flow and can be recognized as causing infinite descent is an ω-regular set, representable by a Büchi automaton. Algorithms for such automata can be used to decide size-change termination. We also give a direct algorithm operating on "size-change graphs" (without the passage to automata).

Compared to other results in the literature, termination analysis based on the size-change principle is surprisingly simple and general: lexical orders (also called lexicographic orders), indirect function calls and permuted arguments (descent that is not *in-situ*) are all handled *automatically and without special treatment*, with no need for manually supplied argument orders, or theorem-proving methods not certain to terminate at analysis time.

We establish the problem's *intrinsic complexity*. This turns out to be surprisingly high, complete for PSPACE, in spite of the simplicity of the principle. PSPACE hardness is proved by a reduction from Boolean program termination. An interesting consequence: the same hardness result applies to many other analyses found in the termination and quasi-termination literature.

*This research was done while visiting DIKU.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.4 [**Programming Languages**]: Processors; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

Keywords

Termination, program analysis, omega automaton, PSPACE-completeness, partial evaluation.

1. INTRODUCTION

1.1 Motivation

There are many reasons to study automatic methods to prove program termination, including:

- Program verification: typically deductive methods are used to show partial correctness (the input-output specification is satisfied provided the program terminates), followed by a separate proof of termination [11].
- Automatic program manipulation: termination has to be ensurable when dealing with machine-generated programs, or ones imported from a possibly untrustworthy context.
- Broad interest: termination has been studied in fields including functional programming [8], logic programming [7, 15, 17, 19, 14], term rewriting systems [3, 20] and partial evaluation. Discussion of related work appears at the end of this paper.
- Interesting analysis: termination is not just an "abstract interpretation" of program values, but rather more subtle.
- Use in *partial evaluation*: this is a step towards a binding-time analysis that will guarantee termination of program specialization [12, 2, 9, 10] and still allow an acceptably high degree of specialization in an offline partial evaluator such as Similix [5].

We emphasize here a careful and precise formulation of a simple but powerful *principle* to decide termination. It is owing to this clear statement of the termination criterion that the PSPACE hardness result has been achieved. The result should interest researchers working with related analyses of comparable power [12, 2, 14, 7, 9], as our complexity result (PSPACE hardness) also applies to their methods. Further, it suggests striving for a PTIME approximation to the present criterion that is sufficiently strong on practical programs.

1.2 This Analysis

We do termination analysis in two distinct phases. Phase 1 is to extract a set of *size-change graphs* from the program. For each function call that *may* occur during actual execution, there is a size-change graph that safely approximates the size relations between source and destination parameters in this call. We assume that the measure of size gives rise to a well-founded order, so that the following principle applies:

> If every infinite computation would give rise to an infinitely decreasing value sequence (according to the size-change graphs), then no infinite computation is possible.

Phase 2 is to apply this criterion. It can be decided *precisely* given a safe set of size-change graphs.

Definition 1. For any set A, define A^* to be the set of all finite sequences over A; and A^ω to be the set of all infinite sequences over A; and $A^{*\omega} = A^* \cup A^\omega$. We use the same notation: $as = a_1 a_2 a_3 \ldots$ for elements of either A^* or A^ω, and write $as = a_1 a_2 a_3 \ldots a_n$ for elements of A^*.

1.3 Syntax and notations

A first order functional language L with the following syntax is considered.

$$
\begin{array}{lcl}
\mathtt{p} \in \mathit{Prog} & ::= & \mathtt{def}_1 \ldots \mathtt{def}_m \\
\mathtt{def} \in \mathit{Def} & ::= & \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}^{\mathtt{f}} \\
\mathtt{e} \in \mathit{Expr} & ::= & \mathtt{x} \\
 & | & \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \\
 & | & \mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \\
 & | & c\!: \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) \\
\mathtt{x} \in \mathit{Parameter} & ::= & \text{identifier} \\
\mathtt{f} \in \mathit{FcnName} & ::= & \text{identifier not in } \mathit{Parameter} \\
\mathtt{op} \in \mathit{Op} & ::= & \text{primitive operator}
\end{array}
$$

The *definition* of function $\mathtt{f}$ has form $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{e}^{\mathtt{f}}$, where $\mathtt{e}^{\mathtt{f}}$ is called the *body* of $\mathtt{f}$. The number $n \geq 0$ of parameters in the definition of $\mathtt{f}$ is called its *arity*, written $\mathit{arity}(\mathtt{f})$. Notation: $\mathit{Param}(\mathtt{f}) = \{\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(n)}\}$ is the set of $\mathtt{f}$'s parameters. In examples the $\mathtt{f}^{(i)}$'s may be named by identifiers, e.g., $\mathtt{f}^{(i)}$ corresponds to $\mathtt{x}_i$ in the grammar above.

Parameters are assumed to be in scope when they are used. This can be checked syntactically. The *entry function* is the first function in the program's list of definitions, denoted $\mathtt{f}_{initial}$. Call sites are labeled with numbers prepended to the call expression, e.g., $c : \mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$.

Without loss of generality, all function names, parameter names and call site labels are distinct from one another. Constants are regarded as 0-ary operators.

1.4 Programs and their semantics

Programs in L are untyped, and are interpreted according to the (very standard call-by-value) evaluation semantics in Figure 5.1. The semantic operator $\mathcal{E}$ is defined as usual for a functional language: $\mathcal{E}[\![\mathtt{e}]\!]\vec{v}$ is the value of expression $\mathtt{e}$ in environment $\vec{v} = (v_1, \ldots, v_n)$ – a tuple containing values of parameters $\mathtt{f}^{(1)}, \ldots, \mathtt{f}^{(n)}$.

$\mathcal{E}$ has type $\mathit{Expr} \to \mathit{Value}^* \to \mathit{Value}^\sharp$, where Value^* is the flat domain of finite value sequences. Domain $\mathit{Value}^\sharp = \mathit{Value} \cup \{\bot, \mathit{Err}\}$ includes values, plus Err to model runtime errors, and $\bot$ to model non-termination. Function $\mathit{lift} : \mathit{Value} \to \mathit{Value}^\sharp$ is the natural injection.

Program $\mathtt{p}$ is terminating on input $\vec{v}$ iff $\mathcal{E}[\![\mathtt{e}^{\mathtt{f}_{initial}}]\!]\vec{v} \neq \bot$.

Definition 2.

1. We write $c : \mathtt{f} \to \mathtt{g}$, or alternatively, $\mathtt{f} \xrightarrow{c} \mathtt{g}$ for a call c to function $\mathtt{g}$ occurring in $\mathtt{e}^{\mathtt{f}}$. The set of all call sites in $\mathtt{p}$ is C.
2. A *call sequence* is a finite or infinite sequence $cs = c_1 c_2 c_3 \ldots \in C^{*\omega}$. It is *well-formed* (for the current program) if and only if there is a sequence of functions $\mathtt{f}_0, \mathtt{f}_1, \ldots$, such that $\mathtt{f}_0 \xrightarrow{c_1} \mathtt{f}_1 \xrightarrow{c_2} \mathtt{f}_2 \xrightarrow{c_3} \ldots$
3. We write $cs : \mathtt{f} \to \mathtt{g}$, or alternatively, $\mathtt{f} \xrightarrow{cs} \mathtt{g}$ if $cs = c_1 c_2 \ldots c_k$ and $\mathtt{f}_0 \xrightarrow{c_1} \mathtt{f}_1 \xrightarrow{c_2} \ldots \xrightarrow{c_k} \mathtt{f}_k$ where $\mathtt{f} = \mathtt{f}_0$ and $\mathtt{g} = \mathtt{f}_k$.
4. A *state* is a pair in $\mathit{FcnName} \times \mathit{Value}^*$. A *state transition* $(\mathtt{f}, \vec{v}) \xrightarrow{c} (\mathtt{g}, \vec{u})$ is a pair of states connected by a call $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ in $\mathtt{f}$'s body $\mathtt{e}^{\mathtt{f}}$, such that $\vec{u} = (u_1, \ldots, u_n)$ and $\mathcal{E}[\![\mathtt{e}_k]\!]\vec{v} = \mathit{lift}(u_k),\ k = 1, \ldots, n$.
5. A *state transition sequence* is a sequence (finite or infinite) of form:
$$sts = (\mathtt{f}_0, \vec{v}_0) \xrightarrow{c_1} (\mathtt{f}_1, \vec{v}_1) \xrightarrow{c_2} (\mathtt{f}_2, \vec{v}_2) \xrightarrow{c_3} \ldots,$$
where $(\mathtt{f}_t, \vec{v}_t) \xrightarrow{c_{t+1}} (\mathtt{f}_{t+1}, \vec{v}_{t+1})$ is a state transition for each $t = 0, 1, \ldots$.
6. The call sequence of sts is $\mathit{calls}(sts) = c_1 c_2 c_3 \ldots$.

A size ordering on values.

We assume given a fixed well-founded partial ordering $<$ on the *Value* domain. Remark: the partial order $<$ is completely distinct from the "definedness" order $\sqsubseteq$ for the semantic domain $\mathit{Value}^\sharp$, and should not be confused with it.

Base operators are interpreted by the auxiliary function $\mathcal{O} : \mathit{Op} \to \mathit{Value}^* \to \mathit{Value}^\sharp$, which is assumed never to yield $\bot$. Thus base operations always terminate, but may cause runtime errors. Typical examples of base operators are the list operators `hd` and `tl` and the predecessor on $\mathbb{N}$. Since $<$ is a well-founded ordering on *Value*, any sequence of base operations that appear to decrease values infinitely must eventually cause abortion, i.e., failure with *Err*.

A *destructor* is defined to be a base operator `op` such that $\mathcal{O}[\![\mathtt{op}]\!](\vec{v}) < v_i$, for each i, provided $\mathcal{O}[\![\mathtt{op}]\!](\vec{v}) \neq \mathit{Err}$, where $\vec{v} = (v_1, \ldots, v_n)$. We use the element *Err* for the result of operations like `tl []` or `pred 0`; note that the common definition `tl [] = []` contradicts the destructor property.

Some examples of terminating programs.
The following examples, mostly tail-recursive, will serve to illustrate the power of the size-change principle.

1. Reverse function, with accumulating parameter:

```
rev(ls)    = 1:r1(ls,[])
r1(ls,a) = if ls=[] then a
               else 2:r1(tl ls, cons (hd ls) a)
```

2. Program with indirect recursion:

```
f(i,x)   = if i=[] then x else 1:g(tl i,x,i)
g(a,b,c) = 2:f(a, cons b c)
```

3. Function with lexically ordered parameters:

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

4. Program with permuted parameters:

```
p(m,n,r) = if r>0 then 1:p(m, r-1, n) else
           if n>0 then 2:p(r, n-1 ,m)
                  else m
```

5. Program with permuted and possibly discarded parameters:

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

6. Program with late-starting sequence of descending parameter values:

```
f(a,b) = if b=[] then 1:g(a,[])
                 else 2:f(cons (hd b) a, tl b)
g(c,d) = if c=[] then d
                 else 3:g(tl c, cons (hd c) d)
```

Claim: all these programs must terminate, for a common reason: any infinite call sequence (regardless of test outcomes) causes infinite descent in one or more values.

1.5 The remainder of the article

Section 2 describes the use and derivation of size-change graphs to model size changes observed at function calls. Section 3 shows two solutions to the problem of deciding whether every infinite call sequence causes an infinitely decreasing sequence of parameter values. One is based on ω-automata that directly characterize the phenomenon of infinite descent. The other solution (probably more practical) employs only elementary graph manipulation. Section 4 proves that the problem of deciding whether every infinite call sequence has infinite descent is PSPACE hard in the size of the subject program. Section 5 concludes with related work and open problems. The Appendix contains additional proofs and disussions.

2. TRACING SIZE CHANGES

2.1 Size-change graphs

Definition 3. Let f, g be function names in program p. A *size-change graph* from f to g, written $G : \mathtt{f} \to \mathtt{g}$, is a bipartite graph from f parameters to g parameters, with labeled-arc set E:

$$\begin{array}{rcl} G & = & (Param(\mathtt{f}), Param(\mathtt{g}), E), \\ E & \subseteq & Param(\mathtt{f}) \times \{\downarrow, \overset{=}{\downarrow}\} \times Param(\mathtt{g}) \end{array}$$

where E does not contain both $\mathtt{f}^{(i)} \xrightarrow{\downarrow} \mathtt{g}^{(j)}$ and $\mathtt{f}^{(i)} \xrightarrow{\overset{=}{\downarrow}} \mathtt{g}^{(j)}$.

The size-change graph is used to capture "definite" information about a function call. An $\mathtt{f}^{(i)} \xrightarrow{\downarrow} \mathtt{g}^{(j)}$ arc indicates that a data value *must* decrease in this call, with respect to the $<$ ordering, while an $\mathtt{f}^{(i)} \xrightarrow{\overset{=}{\downarrow}} \mathtt{g}^{(j)}$ arc indicates that a value must either decrease or remain the same. The absence of an arc between a pair of parameters means that none of these relations is asserted to be true for them.

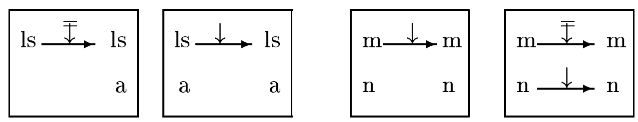
Note: For given f, g in program p there are only finitely many possible size-change graphs $G : \mathtt{f} \to \mathtt{g}$.

Definition 4. Henceforth $\mathcal{G} = \{G_c \mid c \in C\}$ denotes a set of size-change graphs associated with subject program p, one for each of p's calls.

Examples of size-change graphs.
Following are size-change graphs for example programs 1 and 3 seen earlier.

Example 1 **Example 3**

$G_1 : \mathtt{rev} \to \mathtt{r1}$ $G_1, G_2 : \mathtt{a} \to \mathtt{a}$



$G_2 : \mathtt{r1} \to \mathtt{r1}$ $G_3 : \mathtt{a} \to \mathtt{a}$

Remarks: In Example 3, there is no arrow in G_1 to n since its value is constant; and none in G_2 since the second argument of call 2 may exceed m and n.

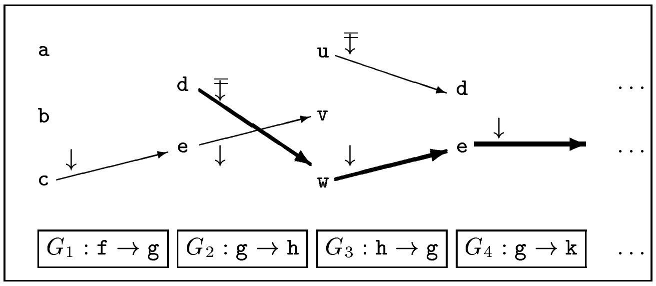
2.2 Multipaths

Definition 5. A *multipath* $\mathcal{M}$ is a finite or infinite sequence $G_{c_1}, G_{c_2}, \ldots$ of size-change graphs. This sequence may be viewed as a concatenated (possibly infinite) graph, as illustrated by:

Program p:

```
f(a,b,c) =    1: g(cons a b, tl c)
g(d,e)   =    ... 2: h([], tl e, d)... 4:k(tl e)
h(u,v,w) =    3: g(u, tl w)
k(x)     =    ...
```

Multipath $\mathcal{M}$ describing the calls in p:



Definition 6.

1. A *thread th* in multipath $\mathcal{M} = G_{c_1}, G_{c_2}, \ldots$ is a connected path of arcs:
$$th = \mathtt{f}_t^{(i_t)} \xrightarrow{r_{t+1}} \mathtt{f}_{t+1}^{(i_{t+1})} \xrightarrow{r_{t+2}} \ldots$$
An example is marked by heavy lines in the example. *Remarks:* a thread need not start at $t = 0$. An instance is the thread starting in d. A thread need not be infinite even if $\mathcal{M}$ is infinite, for instance the thread from c ending in v.

   A thread is *maximal* if the connected path of arcs is maximal in the multipath.

2. Thread *th* is *descending* if the sequence $r_{t+1}, r_{t+2}, \ldots$ has at least one $\downarrow$. The thread is *infinitely descending* if it contains infinitely many occurrences of $\downarrow$.

2.2.1 Multipaths of a state transition sequence and of a call sequence

A size-change graph can be used to describe the parameter size changes in *one concrete state transition sequence*, or it may be used abstractly, to depict size changes following a call sequence *cs*.

Definition 7. Consider state transition sequence
$$sts = (\mathtt{f}_0, \vec{v}_0) \xrightarrow{c_1} (\mathtt{f}_1, \vec{v}_1) \xrightarrow{c_2} (\mathtt{f}_2, \vec{v}_2) \xrightarrow{c_3} \ldots,$$
Define $\mathcal{M}(sts)$ to be the multipath $G_1, G_2, \ldots$, such that for each t, G_{t+1} is a size-change graph from $\mathtt{f}_t$ to $\mathtt{f}_{t+1}$, with arcs $\mathtt{f}_t^{(i)} \xrightarrow{r} \mathtt{f}_{t+1}^{(j)}$ satisfying $r = \downarrow$ if $u_j < v_i$, and $r = \overline{\downarrow}$ if $u_j = v_i$, where $\vec{v}_t = (v_1, \ldots, v_m), \vec{v}_{t+1} = (u_1, \ldots, u_n)$.

Definition 8. Suppose $\mathcal{G} = \{G_c \,|\, c \text{ is a call in } \mathtt{p}\}$ is a set of size-change graphs for p. Given a call sequence $cs = c_1c_2c_3\ldots$, the *$\mathcal{G}$-multipath for cs* is defined by $\mathcal{M}^{\mathcal{G}}(cs) = G_{c_1}, G_{c_2}, G_{c_3}, \ldots$.

Note that $\mathcal{M}(sts)$ displays the *actual size relations* among parameter values along a state transition sequence, while $\mathcal{M}^{\mathcal{G}}(cs)$ displays the information provided by the size-change graphs in $\mathcal{G}$.

2.2.2 Safety of a set $\mathcal{G}$ of size-change graphs

Definition 9. Suppose $\mathcal{G} = \{G_c \,|\, c \text{ is a call in } \mathtt{p}\}$ is a set of size-change graphs for p.

1. Let f's definition contain call $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$. The phrase "arc $\mathtt{f}^{(i)} \xrightarrow{r} \mathtt{g}^{(j)}$ safely describes the $\mathtt{f}^{(i)}$-$\mathtt{g}^{(j)}$ size relation in call c" means: For every $v \in \mathit{Value}$ and $\vec{v} = (v_1, \ldots, v_{arity(\mathtt{f})})$ such that $\mathcal{E}[\![\mathtt{e}_j]\!]\vec{v} = \mathit{lift}\ v$:
$$r = \downarrow \text{ implies } v < v_i \text{ ; and } r = \overline{\downarrow} \text{ implies } v \leq v_i.$$
2. Size-change graph G_c is *safe for call* $c : \mathtt{f} \to \mathtt{g}$ if every arc in G_c is a safe description as just defined.
3. Set $\mathcal{G}$ of size-change graphs is a *safe description of program* p if graph G_c is safe for every call c.

It is easy to see that all the size-change graphs given earlier for examples 1 and 3 are safe for their respective calls. Consider the call `2:a(m-1, 3:a(m,n-1))` in example 3, and the size-change graph $G_2 : \mathtt{a} \to \mathtt{a}$ shown earlier. Call 2 clearly decreases the current value of m, accounting for the arc $\mathtt{m} \xrightarrow{\downarrow} \mathtt{m}$. No size relation can be safely asserted about argument n, since `3:a(m,n-1)` may exceed the current values of m and n. According to Definition 9, G_2 safely models the parameter size-changes caused by call 2.

2.2.3 Choice of $\mathcal{G}$

The analysis is highly dependent on the choice of set $\mathcal{G}$. In general, we cannot insist that each G_c be the most precise set of arcs possible, as this is generally undecidable. However, it is safe to include only relations that must always hold (assuming successful argument evaluation).

In general, it is possible to build $\mathcal{G}$ around any size measure that is well-founded, for instance, the absolute value of an integer, the number of nodes in a tree, or the length of a list. Local properties of base functions `hd`, `tl`, `-1` suffice to yield suitable graph sets for Examples 1–6.

It may be necessary to perform *global size-analysis* [6, 9, 10, 14, 15, 19] to make the best use of size considerations. For instance, global size analysis is needed to handle sorting algorithms automatically.

By the definition of safety of $\mathcal{G}$, it is always valid to omit an arc, but if an arc can be safely included it should be: greater precision may be obtained since more threads may be discovered to have infinite descent. Although a *maximal* safe $\mathcal{G}$ is in general noncomputable, the size-change principle separates the concerns of approximating $\mathcal{G}$ and analyzing it; and in this paper we focus on the analysis phase.

2.3 Termination analysis based on a safe $\mathcal{G}$

2.3.1 Basis of the analysis

If $\mathcal{G}$ is a safe set of size-change graphs and *sts* is a state transition sequence, then $\mathcal{M}(sts)$ is safely described by the $\mathcal{G}$-multipath $\mathcal{M}^{\mathcal{G}}(cs)$ that follows the calls *cs* in *sts*:

LEMMA 1. *Suppose $\mathcal{G}$ is a safe description of program* p, *and state transition sequence* $sts = (\mathtt{f}_0, \vec{v}_0) \xrightarrow{c_1} (\mathtt{f}_1, \vec{v}_1) \xrightarrow{c_2} (\mathtt{f}_2, \vec{v}_2) \xrightarrow{c_3} \ldots$ *has call sequence* $cs = \text{calls}(sts)$. *Consider multipaths* $\mathcal{M}^{\mathcal{G}}(cs) = G_1, G_2, \ldots$ *and* $\mathcal{M}(sts) = G'_1, G'_2, \ldots$. *Then*

1. *if G_{t+1} has arc $\mathtt{f}_t^{(i)} \xrightarrow{\downarrow} \mathtt{f}_{t+1}^{(j)}$, then G'_{t+1} has the same arc; and*
2. *if G_{t+1} has $\mathtt{f}_t^{(i)} \xrightarrow{\overline{\downarrow}} \mathtt{f}_{t+1}^{(j)}$, then G'_{t+1} has an arc $\mathtt{f}_t^{(i)} \xrightarrow{r} \mathtt{f}_{t+1}^{(j)}$ for $r = \overline{\downarrow}$ or $r = \downarrow$.*

PROOF. Immediate by comparing definitions 9 and 7. □

COROLLARY 1. *If $\mathcal{M}^{\mathcal{G}}(cs)$ has an infinite thread th, and $cs = calls(sts)$, then $\mathcal{M}(sts)$ also has an infinite thread th'. Furthermore, thread th' has at least as many $\downarrow$-labeled arcs as th.*

PROOF. Immediate from Lemma 1 and Definition 9. □

2.3.2 The analysis, abstractly

We next define two sets of infinite call sequences: those that are possible according to the program's flow graph, and those that necessarily cause an infinite descent.

Definition 10.

$$FLOW^\omega = \{cs = c_1c_2\ldots \in C^\omega \mid cs \text{ is well-formed and } c_1 : \mathtt{f}_{initial} \to \mathtt{f}_1\}$$

$$DESC^\omega = \{cs \in FLOW^\omega \mid \text{some thread } th \text{ in } \mathcal{M}^{\mathcal{G}}(cs) \text{ has infinitely many } \downarrow\text{-arcs}\}$$

The result $\perp$ can only arise from an infinite state transition sequence. This holds even though calls may be nested (and even in a higher-order extension of the programming language). Two lemmas prove this property:

LEMMA 2. *Assume that $\mathcal{E}[\![\mathtt{e}]\!]\vec{v} = \perp$. Then there exists a call $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ in $\mathtt{e}$ such that $\mathcal{E}[\![\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\vec{v} = \perp$ but $\mathcal{E}[\![\mathtt{e}_i]\!]\vec{v} \neq \perp$ for each i.*

PROOF. Suppose inductively that the result holds for all subexpressions of $\mathtt{e}$ and for every $\vec{v} \in Value^*$. Referring to the semantics of Figure 5.1:

Case $\mathtt{e} = \mathtt{x}$: The result is trivial.

Case $\mathtt{e} = \mathtt{if}\ \mathtt{e}'_1\ \mathtt{then}\ \mathtt{e}'_2\ \mathtt{else}\ \mathtt{e}'_3$: The result holds by induction, since $\mathcal{E}[\![\mathtt{e}]\!]\vec{v} = \mathcal{E}[\![\mathtt{e}'_i]\!]\vec{v}$ where $i = 1$ if $\mathcal{E}[\![\mathtt{e}'_1]\!]\vec{v} \in \{\perp, Err\}$, $i = 2$ if $\mathcal{E}[\![\mathtt{e}'_1]\!]\vec{v} = True$, else $i = 3$.

Case $\mathtt{e} = \mathtt{op}(\mathtt{e}'_1, \ldots, \mathtt{e}'_m)$: Since $\mathcal{O}[\![\mathtt{op}]\!](\vec{u}) \neq \perp$ for all $\vec{u} \in Value^*$, by definition of *strictapply*, $\mathcal{E}[\![\mathtt{e}]\!]\vec{v}$ is equal to the least i for which $\mathcal{E}[\![\mathtt{e}'_i]\!]\vec{v} \in \{\perp, Err\}$. For this value of i, $\mathcal{E}[\![\mathtt{e}'_i]\!]\vec{v} = \perp$, so the result follows from the inductive hypothesis.

Case $\mathtt{e} = c : \mathtt{h}(\mathtt{e}'_1, \ldots, \mathtt{e}'_m)$: If $\mathcal{E}[\![\mathtt{e}'_i]\!]\vec{v} = \perp$ for some i, then the result follows from the inductive hypothesis. Otherwise, $\mathcal{E}[\![\mathtt{e}'_i]\!]\vec{v} \neq \perp$ for $i = 1, \ldots, m$, so the result holds with $\mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n) = \mathtt{h}(\mathtt{e}'_1, \ldots, \mathtt{e}'_m)$. □

LEMMA 3. *Suppose $\mathcal{E}[\![\mathtt{e}^{\mathtt{f}_{initial}}]\!]\vec{v}_0 = \perp$. Then there exists an infinite state transition sequence: $sts = (\mathtt{f}_{initial}, \vec{v}_0) \stackrel{c_1}{\to} (\mathtt{f}_1, \vec{v}_1) \stackrel{c_2}{\to} (\mathtt{f}_2, \vec{v}_2) \stackrel{c_3}{\to} \ldots$.*

PROOF. It follows from the previous result that given any program state $(\mathtt{f}, \vec{v})$ where $\mathcal{E}[\![\mathtt{e}^{\mathtt{f}}]\!]\vec{v} = \perp$, there exists a call $c : \mathtt{g}(\mathtt{e}_1, \ldots, \mathtt{e}_n)$ in $\mathtt{e}^{\mathtt{f}}$ such that $\mathcal{E}[\![\mathtt{e}_i]\!]\vec{v} \neq \perp$ for each i. Let $\mathcal{E}[\![\mathtt{e}_i]\!]\vec{v} = lift(u_i)$ for each i, and $\vec{u} = (u_1, \ldots, u_n)$. By definition, $(\mathtt{f}, \vec{v}) \to (\mathtt{g}, \vec{u})$ is a state transition, such that $\mathcal{E}[\![\mathtt{e}^{\mathtt{g}}]\!]\vec{u} = \perp$.

Starting with the one-state transition-sequence $(\mathtt{f}_{initial}, \vec{v}_0)$, where $\mathcal{E}[\![\mathtt{e}^{\mathtt{f}_{initial}}]\!]\vec{v}_0 = \perp$, and extending inductively, the existence of the infinite state transition sequence is deduced. □

Example 3 revisited: consider the three calls `1:a(m-1,1)`, `2:a(m-1, 3:a(m,n-1))` and `3:a(m,n-1)`. Lemma 2 asserts that for a call to function `a` to be non-terminating, either call 1 is non-terminating; *or* call 3 is non-terminating; *or* call 3 is terminating, but call 2 is non-terminating. By the definition of safety, there is a size-change graph in $\mathcal{G}$ to account for each of these possibilities.

THEOREM 1. *If $FLOW^\omega = DESC^\omega$ then program $\mathtt{p}$ terminates for all inputs.*

PROOF. It will be proved that if $\mathtt{p}$ is *not* terminating, there is a cs in $FLOW^\omega$ but not in $DESC^\omega$. Suppose $\mathtt{p}$ does not terminate on $\vec{v}$. Then by Lemma 3, there exists an infinite state transition sequence $sts = (\mathtt{f}_{initial}, \vec{v}_0) \stackrel{c_1}{\to} (\mathtt{f}_1, \vec{v}_1) \stackrel{c_2}{\to} (\mathtt{f}_2, \vec{v}_2) \stackrel{c_3}{\to} \ldots$.

Call sequence $cs = calls(sts) \in C^\omega$ is clearly in $FLOW^\omega$. Suppose $cs \in DESC^\omega$. Then multipath $\mathcal{M}^{\mathcal{G}}(cs)$ has a thread with infinitely many $\downarrow$-labeled arcs. By Corollary 1, the same is true of $\mathcal{M}(sts)$. By definition of $\mathcal{M}(sts)$, there exists a corresponding sequence of values, infinitely decreasing in a well-founded domain. This is impossible. □

Definition 11. Program $\mathtt{p}$ is *size-change terminating* (for this choice of $\mathcal{G}$) if and only if $FLOW^\omega = DESC^\omega$.

2.3.3 The examples revisited

Theorem 1 can be used, as is, for termination by proving that any $cs \in FLOW^\omega$ must be in $DESC^\omega$. However, the reasoning can be tricky for some programs (for instance, see Examples 4, 5 below, which seem to possess no natural lexical descent). We prove later that the reasoning is necessarily tricky, since the problem is PSPACE-hard. In the next section, we will give two algorithms to perform the test automatically.

In the following, we extend the regular-expression notation to allow a single (final) use of ω, e.g., $12^\omega = 1222\ldots$.

Example 1: $FLOW^\omega$ is the singleton set $\{12^\omega\}$. Call sequence $cs = 12^\omega$ gives infinite descent in parameter `ls`, so $FLOW^\omega = DESC^\omega$.

Example 2: $FLOW^\omega = \{(12)^\omega\}$. Call sequence $cs = (12)^\omega$ gives infinite descent in `i`.

Example 3: $FLOW^\omega = (1+2+3)^\omega$ (the set of all infinite strings composed of 1, 2 and 3). If $cs \in FLOW^\omega$ ends in 3^ω, then `n` descends infinitely. Otherwise $cs \in FLOW^\omega$ contains infinitely many 1's or 2's, so `m` descends infinitely.

Example 4: Consider multipath $\mathcal{M}^{\mathcal{G}}(cs)$ for any $cs \in FLOW^\omega = (1+2)^\omega$. The threads starting at `m,n,r` all continue regardless of call sequence, and at least one $\downarrow$ occurs for each call. Now $\mathcal{M}^{\mathcal{G}}(cs)$ has 3 maximal threads and infinitely many $\downarrow$, so at least one thread must contain infinitely many $\downarrow$. That thread is thus infinitely descending, so $cs \in DESC^\omega$.

Example 5: Any finite sequence in (12^*) has a thread from `y` to `y` containing at least one $\downarrow$. If $cs \in FLOW^\omega$ contains infinitely many 1's, then `y` descends infinitely. Otherwise cs ends in 2^ω, and both `x` and `y` descend infinitely.

Example 6: Infinite call sequences must have form 2^ω or

2^*13^ω. Both cause infinite descent, of parameter `b` in the first case, and `c` in the other.

3. DETECTING THREADS OF INFINITE DESCENT

The first solution to size-change termination analysis is based on the theory of ω-automata. These automata can directly characterize the infinite-descent phenomenon.

3.1 An analysis based on ω-automata

Definition 12. A *Büchi automaton* $\mathcal{A} = (In, S, S_0, \rho, F)$ is a tuple where In is a finite set called *input symbols*, S is a finite set called *states*, $S_0 \subseteq S$ is the set of *initial states*, and $F \subseteq S$ is the set of *accepting states*. The *state transition relation* is a set of transition triples $\rho \subseteq S \times In \times S$.

Definition 13. Behavior of a Büchi automaton $\mathcal{A}$.

1. A *run* of $\mathcal{A}$ on an infinite word $w = a_1a_2a_3\ldots \in In^\omega$ is a sequence $s_0a_1s_1a_2s_2a_3s_3\ldots \in S(InS)^\omega$ such that $s_0 \in S_0$, and $(s_t, a_{t+1}, s_{t+1}) \in \rho$ for $t = 0, 1, 2, 3, \ldots$.
2. The run r is *accepting* if and only if for some $s \in F$, s occurs infinitely often among $s_0s_1s_2s_3\ldots$.
3. $L_\omega(\mathcal{A}) = \{w \in In^\omega \mid \text{some run on } w \text{ is accepting}\}$

A set $A \subseteq In^\omega$ is called *ω-regular* iff it is accepted by some Büchi automaton.

THEOREM 2. *[18] The following problem is complete for* PSPACE*:* Given *Büchi automata $\mathcal{A}$ and $\mathcal{A}'$*, to decide *whether $L_\omega(\mathcal{A}) = L_\omega(\mathcal{A}')$.*

LEMMA 4. *$FLOW^\omega$ is an ω-regular subset of C^ω.*

PROOF. $FLOW^\omega = L_\omega(\mathcal{A})$ for Büchi automaton $\mathcal{A} = (C, FcnName, \{\mathtt{f}_{initial}\}, \rho, FcnName\})$. The transition relation is: $\rho = \{(\mathtt{f}, c, \mathtt{g}) \mid c : \mathtt{f} \to \mathtt{g}\}$.

Explanation: $\mathcal{A}$ is just the program's call graph, with function names as states, the initial function as initial state, and calls as transitions. Any infinite call sequence must enter at least one function infinitely often. Thus, defining all states as accepting puts every well-formed infinite call sequence in $L_\omega(\mathcal{A})$. □

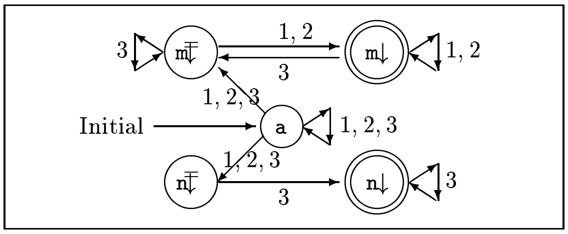
A Büchi automaton to accept $DESC^\omega$

We first describe the construction informally by an example, before stating the formal construction.

STAGE 1: Build a Büchi automaton to accept call sequence cs iff its associated multipath $\mathcal{M}^{\mathcal{G}}(cs)$ has an infinite descending thread *from the start of cs*. The states of this automaton represent function parameters $\mathtt{x}, \mathtt{y}$ and the transitions correspond to calls c whose associated size-change graph G_c includes an arc $\mathtt{x} \xrightarrow{r} \mathtt{y}$.

In order to track size changes that occur in the thread, the states are defined as pairs of function parameters and size changes: $\mathtt{y}\overline{\downarrow}$ or $\mathtt{y}\downarrow$, according to the size-change r on the incoming arc. An infinitely descending thread from the start of the multipath then corresponds to a run of the automaton which enters infinitely many states of form $\mathtt{x}\downarrow$.

For Example 3, the automaton can be seen in the diagram below. (Ignore state `a`, treated in Stage 2.) The states are $\mathtt{m}\overline{\downarrow}, \mathtt{m}\downarrow, \mathtt{n}\overline{\downarrow}, \mathtt{n}\downarrow$. Size-change graphs G_1 and G_2 (shown in Section 2.1) decrease `m`, accounting for the arcs labeled 1,2 in the figure (entering $\mathtt{m}\downarrow$). Size-change graph G_3 decreases `n` and copies `m`, explaining the arcs labeled 3. Accepting states are $\mathtt{m}\downarrow, \mathtt{n}\downarrow$, and initial states are $\mathtt{m}\overline{\downarrow}, \mathtt{n}\overline{\downarrow}$.



STAGE 2: Stage 1 traces size changes only in threads that start at the beginning of execution. To deal with late-starting threads, add to the automaton of Stage 1 a copy of the program's call graph: the automaton of Lemma 4. Further, for every call $c : \mathtt{f} \to \mathtt{g}$, allow a c-transition from function-name state `f` to any parameter-name state $\mathtt{g}^{(i)}\overline{\downarrow}$.

For Example 3 the call graph has only node `a` and calls 1,2,3 from `a` to itself, so the result is as above.

Section 2.3.3 had an argument to justify $FLOW^\omega = DESC^\omega$ for this example. That reasoning can clearly be applied to the automaton's behavior on words cs in $(1+2+3)^\omega$. If cs ends in infinitely many 3's, the automaton can cycle in state `a` until the last symbol in $\{1, 2\}$ is read and then proceed to accepting state $\mathtt{n}\downarrow$ and stay there. If on the other hand cs contains infinitely many symbols in $\{1, 2\}$, a transition to the top half causes the automaton to enter accepting state $\mathtt{m}\downarrow$ infinitely often.

LEMMA 5. *$DESC^\omega$ is an ω-regular subset of C^ω.*

PROOF. Stage 1 of the construction just sketched leads to automaton $\mathcal{A}_1 = (C, S_1, S_0, \rho_1, F)$ where

$$\begin{array}{lcl} S_1 & = & Parameter \times \{\downarrow, \overline{\downarrow}\} \\ S_0 & = & Param(\mathtt{f}_{initial}) \times \{\overline{\downarrow}\} \\ \rho_1 & = & \{\ (\mathtt{x}r, c, \mathtt{x}'r') \mid \mathtt{x} \xrightarrow{r'} \mathtt{x}' \in G_c, r \in \{\downarrow, \overline{\downarrow}\}, c \in C\ \} \\ F & = & \{\mathtt{x}\downarrow \mid \mathtt{x} \in Parameter\} \end{array}$$

The program's call graph in automaton form was seen in Lemma 4 to be $(C, FcnName, \{\mathtt{f}_{initial}\}, \rho, FcnName\})$. Combining this with $\mathcal{A}_1$, we obtain

$$\mathcal{A} = (C, S_1 \cup FcnName, S_0 \cup \{\mathtt{f}_{initial}\}, \rho_1 \cup \rho \cup \rho_2, F)$$

where $\rho_2 = \{(\mathtt{f}, c, \mathtt{x}\overline{\downarrow}) \mid c : \mathtt{f} \to \mathtt{g}, \mathtt{x} \in Param(\mathtt{g})\}$. Correctness of this construction is straightforward. □

THEOREM 3. *Size-change termination can be decided in space polynomial in the size of program* `p`.

PROOF. By definition 11, $\mathtt{p}$ is size-change terminating if and only if $FLOW^\omega = DESC^\omega$. The automata constructed in Lemmas 4 and 5 to accept $FLOW^\omega$ and $DESC^\omega$ have size that is polynomially bounded in the length of the program $\mathtt{p}$ from which they were constructed. By Theorem 2, their equivalence can be tested in PSPACE. $\square$

In algorithmic practice, tests for equivalence of atomata involve determinization of the (nondeterministic) automata. While in principle this can be done in PSPACE, the best known algorithm (due to Safra [16]) seems to give large automata and thus slow computations.

3.2 A graph-based algorithm

An alternative algorithm uses graph manipulation rather than ω-automata.

Definition 14. The *composition* of two size-change graphs $G : \mathtt{f} \to \mathtt{g}$ and $G' : \mathtt{g} \to \mathtt{h}$ is $G; G' : \mathtt{f} \to \mathtt{h}$ with arc set E defined below. Notation: we write $\mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z}$ if $\mathtt{x} \xrightarrow{r} \mathtt{y}$ and $\mathtt{y} \xrightarrow{r'} \mathtt{z}$ are respectively arcs of G and G'.

$$
\begin{array}{rl}
E = & \{\mathtt{x} \xrightarrow{\downarrow} \mathtt{z} \mid \exists \mathtt{y}, r\ .\ \mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \xrightarrow{r} \mathtt{z} \text{ or } \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{\downarrow} \mathtt{z}\} \\
\bigcup & \{\mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z} \mid (\exists \mathtt{y}\ .\ \mathtt{x} \xrightarrow{\overset{=}{\downarrow}} \mathtt{y} \xrightarrow{\overset{=}{\downarrow}} \mathtt{z}) \text{ and} \\
& \quad \forall \mathtt{y}, r, r'\ .\ \mathtt{x} \xrightarrow{r} \mathtt{y} \xrightarrow{r'} \mathtt{z} \text{ implies } r = r' = \overset{=}{\downarrow}\}
\end{array}
$$

LEMMA 6. *Graph composition is associative.*

Definition 15. For a well-formed nonempty call sequence $cs = c_1 \ldots c_n$, define the size-change graph for cs, denoted G_{cs}, as $G_{c_1}; \ldots ; G_{c_n}$.

LEMMA 7. *Multipath $\mathcal{M} = G_1, \ldots , G_n$ has a thread from $\mathtt{x}$ to $\mathtt{y}$ over its entire length, containing at least one $\downarrow$-labeled arc, if and only if $\mathtt{x} \xrightarrow{\downarrow} \mathtt{y} \in G_1; \ldots ; G_n$.*

Definition 16. Define the set $\mathcal{S}$ by

$$\mathcal{S} = \{G_{cs} \mid cs, cs_0 \text{ are well-formed and } \mathtt{f}_{initial} \xrightarrow{cs_0} \mathtt{f} \xrightarrow{cs} \mathtt{g}\}$$

The set $\mathcal{S}$ is finite since there are finitely many possible graphs. However, its size may be exponential in the program's size (in fact, the construction in the following section can be used to create such examples).

The central idea in the graph-based algorithm:

THEOREM 4. *Program $\mathtt{p}$ is* not *size-change terminating iff $\mathcal{S}$ contains $G : \mathtt{f} \to \mathtt{f}$ such that $G = G; G$ and G has no arc of form $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$.*

PROOF. For the forward implication, suppose $\mathtt{p}$ is not size-change terminating. Then there is an infinite call sequence $cs = c_1 c_2 \ldots$ such that $\mathcal{M}^{\mathcal{G}}(cs)$ has no infinitely descending thread.

Define a *2-set* to be a 2-element set $\{t, t'\}$ of positive integers. Without loss of generality, $t < t'$. Now for each $G \in \mathcal{S}$, define the *class P_G of 2-sets yielding G* by:

$$P_G = \{(t, t') \mid G = G_{c_t}; G_{c_{t+1}}; \ldots , G_{c_{t'-1}}\}$$

This set $\{P_G \mid G \in \mathcal{S}\}$ of classes is mutually disjoint, every 2-set belongs to exactly one of them, and it is finite since $\mathcal{S}$ is finite. By Ramsey's theorem, there is an infinite set of positive integers, T, such that all 2-sets $\{t, t'\}$ with $t, t' \in T$ are in the same class. Call this class P_{G°.

Thus for any $t, t' \in T$ with $t < t'$, $G_{c_t}; \ldots ; G_{c_{t'-1}}$ is equal to the same G°. This implies that $G^\circ : \mathtt{f} \to \mathtt{f}$ for some $\mathtt{f}$, and for $t, t', t'' \in T$, with $t < t' < t''$,

$$
\begin{array}{rcl}
G^\circ & = & G_{c_t}; \ldots ; G_{c_{t''-1}} \\
& = & (G_{c_t}; \ldots ; G_{c_{t'-1}}); (G_{c_{t'}}; \ldots ; G_{c_{t''-1}}) \\
& = & G^\circ; G^\circ.
\end{array}
$$

If G° has an arc $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$, then by Lemma 7, each multipath section $G_{c_t}, \ldots , G_{c_{t'-1}}$, where $t \in T$, and t' is the next bigger integer after t in T, would have a descending thread from $\mathtt{x}$ to $\mathtt{x}$, and $\mathcal{M}^{\mathcal{G}}(cs)$ would have an infinitely descending thread, violating the assumption about cs. Therefore, G° has no arc of form $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$. This establishes the forward implication.

For the reverse implication, let $G^\circ \in \mathcal{S}$ be such that $G^\circ = G^\circ; G^\circ$ and suppose G° has no arc of form $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$. By definition of $\mathcal{S}$, there exist cs_0 and cs_1 such that $cs = cs_0(cs_1)^\omega \in FLOW^\omega$, and $G_{cs_1} = G^\circ$. Suppose, for a contradiction, that $\mathtt{p}$ is size-change terminating. Then $(cs_1)^\omega$ has an infinitely descending thread. Consider the position of this thread at the start of each cs_1-section. Some parameter $\mathtt{x}$ must be visited by the thread at these points infinitely often, since *Parameter* is finite. Given sufficiently many repeats of cs_1, we can find a $\downarrow$-labeled arc in a thread from $\mathtt{x}$ to $\mathtt{x}$. In other words, there is a number n such that $\mathcal{M}^{\mathcal{G}}((cs_1)^n)$ has a descending thread from $\mathtt{x}$ to $\mathtt{x}$. By Lemma 7, arc $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x}$ is in $G_{(cs_1)^n} = (G_{cs_1})^n = (G^\circ)^n = G^\circ$, which gives the required contradiction. $\square$

An algorithmic realization of Theorem 4.

1. Build the set $\mathcal{S}$ by a transitive closure procedure:
   - Include every $G_c : \mathtt{f} \to \mathtt{g}$ where $c : \mathtt{f} \to \mathtt{g}$ is a call in program $\mathtt{p}$, and $\mathtt{f}$ is reachable by some well-formed $cs_0 : \mathtt{f}_{initial} \to \mathtt{f}$.
   - For any $G : \mathtt{f} \to \mathtt{g}$ and $H : \mathtt{g} \to \mathtt{h}$ in $\mathcal{S}$, include also $G; H$ in $\mathcal{S}$.
2. For each $G : \mathtt{f} \to \mathtt{f}$ in $\mathcal{S}$, test whether $G = G; G$ and $\mathtt{x} \xrightarrow{\downarrow} \mathtt{x} \notin G$ for each $\mathtt{x} \in Param(\mathtt{f})$.

The test in step 2 takes low-order polynomial time; so the bottleneck in this algorithm is the cardinality of $\mathcal{S}$, i.e., the number of different compositions of reachable size-change graphs. This number can be exponential in the input program's size, hence our algorithm has exponential time and space complexity.

However, space usage can be reduced to polynomial by not creating all of the set $\mathcal{S}$ at any time, but generating it "on the fly" as demanded by step 2. We omit the details, which are standard. For practical usage, the increase in time required to make the space polynomial is probably a waste, and the simple algorithm seems more promising than the PSPACE version, or the solution based on ω-automata.

4. COMPLEXITY OF SIZE-CHANGE TERMINATION

As the previous section shows, given the set $\mathcal{G}$, size-change termination can be decided in polynomial space (and exponential time) using either of the given approaches. It may surprise the reader, as it did the authors, to find that size-change termination, in spite of its simplicity, is *a complete problem for* PSPACE, hence intractable in general, unless PTIME = PSPACE. The proof is, as usual, by reduction from a known PSPACE-complete problem. Note that equivalence of Büchi automata is known to be PSPACE-complete, as mentioned earlier, but our problem is a *special case*, hence a specific hardness proof is necessary.

Definition 17. A *Boolean program* is an instruction sequence $b = \texttt{1:I}_1\ \texttt{2:I}_2\ \ldots\ \texttt{m:I}_m$ specifying a computation on variables $\texttt{X}_1, \ldots, \texttt{X}_k$, ranging over truth values *true*, *false*.

Instructions $\texttt{I}_\ell$ have two formats: $\texttt{X}_i\ \texttt{:= not X}_i$, and $\texttt{if X}_i\ \texttt{then goto}\ \ell'\ \texttt{else}\ \ell''$. Here $1 \leq i \leq k$ and $\ell, \ell', \ell'' \in \{0, 1, 2, \ldots, m\}$.

Semantics: the *computation* by b is a finite or infinite state sequence $b \vdash (\ell_1, \sigma_1) \to (\ell_2, \sigma_2) \to \ldots$, where each store σ assigns a truth value in $\{true, false\}$ to each of b's variables, and ℓ_t is the control point at time t.

Initially $\ell_1 = 1$ and σ_1 assigns *false* to every variable. Inductively, given state (ℓ_t, σ_t), if $\ell_t = 0$ then the computation has terminated, else the following rules apply.

If instruction $\texttt{I}_{\ell_t}$ is $\texttt{X}_i\ \texttt{:= not X}_i$, then σ_{t+1} is identical to σ_t except that $\sigma_{t+1}(\texttt{X}_i) = \neg\sigma_t(\texttt{X}_i)$. Further, $\ell_{t+1} = (\ell_t + 1) \bmod (m+1)$.

If instruction $\texttt{I}_{\ell_t}$ is $\texttt{if X}_i\ \texttt{then goto}\ \ell'\ \texttt{else}\ \ell''$, then σ_{t+1} is identical to σ_t. Further, $\ell_{t+1} = \ell'$ if $\sigma_t(\texttt{X}_i) = true$, and $\ell_{t+1} = \ell''$ if $\sigma_t(\texttt{X}_i) = false$.

Finally, program b *terminates*, written $[\![b]\!]\!\downarrow$, iff for some t: $b \vdash (\ell_1, \sigma_1) \to \ldots \to (\ell_t, \sigma_t) = (0, \sigma_t)$.

LEMMA 8. *The following set is complete for* PSPACE:

$$\mathcal{B} = \{b \mid b \textit{ is a Boolean program and } [\![\mathrm{b}]\!]\!\downarrow\}$$

PROOF. $\mathcal{B}$ is in PSPACE by a simple simulation, using a counter to declare nontermination if the computation has taken more that $(m+1) \cdot 2^k$ steps. For PSPACE-hardness, see [13]; or reduce QBF (truth of quantified Boolean formulas) to membership in $\mathcal{B}$. $\square$

THEOREM 5. *Size-change termination is* PSPACE*-hard.*

Let SCT stand for the set of all $\mathcal{G}$'s that satisfy the size-change termination criterion $FLOW^\omega = DESC^\omega$, and $\overline{\text{SCT}}$ be its complement. The theorem will be proved by reduction from $\mathcal{B}$ to $\overline{\text{SCT}}$. Concretely, given a Boolean program b, we will construct a program p of size polynomial in the size of b, with associated set of size-change graphs $\mathcal{G}$, and prove that $b \in \mathcal{B}$ if and only if $\mathcal{G} \notin$ SCT.

Construction.

Suppose program $b = \texttt{1:I}_1\ \texttt{2:I}_2\ \ldots\ \texttt{m:I}_m$ has k variables $\texttt{X}_1, \ldots, \texttt{X}_k$. Without loss of generality, each variable has value *false* after execution, if b terminates (just add at the end of b one test and one assignment for each variable.)

Program p will have functions $\{F_0, F_1, \ldots, F_m\}$, each one of $2k+1$ parameters named $\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{Z}$. It will use a single operator tl, assumed to be a unary destructor.

Definition of initial function F_0:

$$\begin{aligned} &F_0(\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{Z}) = \\ &\qquad 0 : F_1(\texttt{tl}\,\overline{\texttt{X}}_1, \overline{\texttt{X}}_1, \ldots, \texttt{tl}\,\overline{\texttt{X}}_k, \overline{\texttt{X}}_k, \overline{\texttt{X}}_k) \end{aligned}$$

Definition of F_ℓ, for instruction $\ell : \texttt{X}_i\ \texttt{:= not X}_i$:

$$\begin{aligned} &F_\ell(\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \texttt{X}_i, \overline{\texttt{X}}_i, \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{Z}) = \\ &\qquad \ell : F_{(\ell+1) \bmod (m+1)}(\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \overline{\texttt{X}}_i, \texttt{X}_i \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{tl}\,\texttt{Z}) \end{aligned}$$

Definition of F_ℓ, for instruction
$\ell : \texttt{if X}_i\ \texttt{then goto}\ \ell'\ \texttt{else}\ \ell''$:

$$\begin{aligned} &F_\ell(\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{Z}) = \ \texttt{if}\ \textit{some-test} \\ &\qquad \texttt{then}\ \ \ell^+ : F_{\ell'}(\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \texttt{X}_i, \texttt{tl}\overline{\texttt{X}}_i, \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{tl}\,\texttt{Z}) \\ &\qquad \texttt{else}\ \ \ell^- : F_{\ell''}(\texttt{X}_1, \overline{\texttt{X}}_1, \ldots, \texttt{tl}\,\texttt{X}_i, \overline{\texttt{X}}_i, \ldots, \texttt{X}_k, \overline{\texttt{X}}_k, \texttt{tl}\,\texttt{Z}) \end{aligned}$$

For *some-test* we use whatever the language permits; our analysis considers every possible flow sequence anyway. The program's set of calls is

$$\begin{aligned} C = \{0\}\ \cup\ &\{\ell \mid \texttt{I}_\ell = \text{“}\texttt{X := not X}\text{”}\} \cup \\ &\{\ell^+, \ell^- \mid \texttt{I}_\ell = \text{“}\texttt{if X goto}\ \ell'\ \texttt{else}\ \ell''\text{”}\} \end{aligned}$$

Example of the construction.

Suppose b is the Boolean program:

```
1: X := not X
2: if Y then goto 5 else 3
3: Y := not Y
4: if X then goto 2 else 3
5: X := not X
6: Y := not Y
```

We construct the following program p:

```
F0(X, X̅, Y, Y̅, Z) = 0: F1(tl X̅, X̅, tl Y̅, Y̅, Y̅)
F1(X, X̅, Y, Y̅, Z) = 1: F2(X̅, X, Y, Y̅, tl Z)
F2(X, X̅, Y, Y̅, Z) = if...
                    then 2⁺: F5(X, X̅, Y, tl Y̅, tl Z)
                    else 2⁻: F3(X, X̅, tl Y, Y̅, tl Z)
F3(X, X̅, Y, Y̅, Z) = 3: F4(X, X̅, Y̅, Y, tl Z)
F4(X, X̅, Y, Y̅, Z) = if...
                    then 4⁺: F2(X, tl X̅, Y, Y̅, tl Z)
                    else 4⁻: F3(tl X, X̅, Y, Y̅, tl Z)
F5(X, X̅, Y, Y̅, Z) = 5: F6(X̅, X, Y, Y̅, tl Z)
F6(X, X̅, Y, Y̅, Z) = 6: F0(X, X̅, Y̅, Y, tl Z)
```

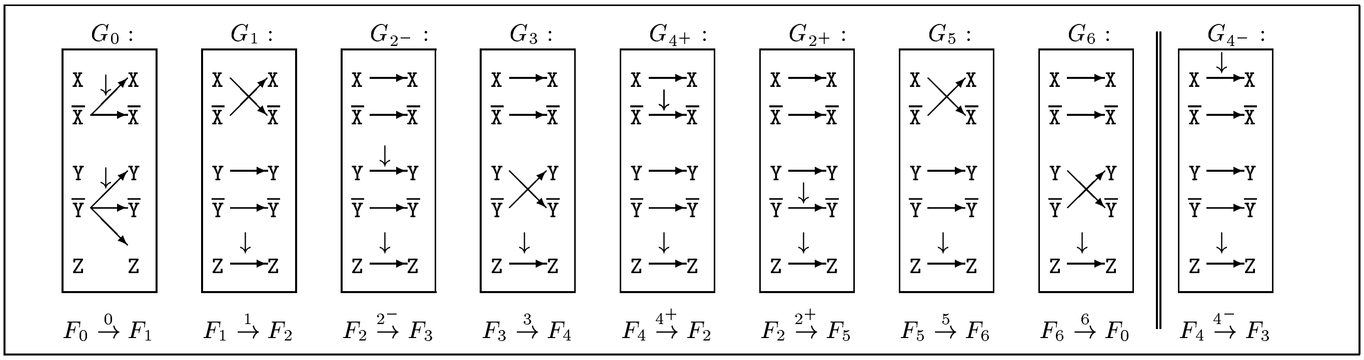


Figure 4.1: Size-change graphs for program p.

Figure 4.1 contains the size-change graphs for all of the calls in p, starting with those that appear within b's computation and ending with call G_{4^-} (this is "dead code"). To avoid cluttering the diagram, $\overset{=}{\downarrow}$ labels have been omitted.

Boolean program b's computation is (writing stores compactly by abbreviating *true*, *false* to T, F):

$$b \vdash \quad 1FF \to 2TF \to 3TF \to 4TT \to 2TT \to 5TT \to 6FT \to 0FF$$

Sequence $cs = 1\,2^-\,3\,4^+\,2^+\,5\,6$ is a "trace" of b's computation, where $+, -$ indicate whether respectively the positive or negative branch of a conditional was taken.

Now consider the sequence $0cs = 0\,1\,2^-\,3\,4^+\,2^+\,5\,6$, obtained by prepending 0 to b's computation. Within p, call sequence $0cs$ is a "loop" from function F_0 back to itself, so $(0cs)^\omega \in FLOW^\omega$. As the reader may verify, there are no infinite descending threads in the corresponding multipath. Thus $DESC^\omega \neq FLOW^\omega$, so p is not size-change terminating, as desired (indeed, for appropriate choices of *some-test*, p really is a non-terminating program).

Appendix A completes the correctness proof.

COROLLARY 2. *The termination and quasi-termination criteria of [2, 7, 9, 10, 12, 14, 17] all are* PSPACE*-hard.*

PROOF. Point: These analyses all give correct results when applied to programs whose data flow is similar to that of p above. The proof is essentially the same, with the construction modified as necessary to make the program fail the condition tested by the respective method, just when the Boolean program terminates. □

Termination analysis in polynomial time?

The reduction shows that the PSPACE hardness of termination analysis holds for very simple programs. In particular, the only expressions we have used are a variable and unary operators applied to a variable. This indicates that the root of the complexity is in the ways values move around among parameter positions. Conclusion: the problem will be less difficult if these ways are restricted. We have an algorithm (as yet unpublished) that decides SCT in worst-case cubic time for restricted programs including the following two cases:

1. Programs whose size-change graphs have in- and out-degrees bounded by 1 (Examples 1, 3, 4 and 6).
2. Programs whose function parameters can be "stratified," i.e., dependencies among call parameters can be partially ordered by position (for example in Example 2, parameter x indirectly depends on i, while i only depends on itself).

5. CONCLUDING REMARKS

5.1 Related work

Our PSPACE lower bound is the first such result of which we are aware. The algorithms to detect termination, though, have some counterparts in other programming language areas.

- Typed functional programs: Abel and Altenkirch [1] have developed a system called **foetus** that accepts as input mutual recursive function definitions over strict positive datatypes. It returns a lexical ordering on the arguments of the program's functions, if one exists. As we have seen (program examples 3, 4, 5), our handles programs with or without such a lexical ordering.
- Logic programs: the *Termilog* [14] approach is as powerful as ours, if applied to the result of converting a functional program into Horn clause form. Graphs analogous to ours are used in that method, but the overall development is considerably more complex.
- Term rewriting: Arts and Giesl [4, 3] translate a subject program into a TRS whose termination implies termination of the program. This approach requires extending existing techniques for TRS termination. Automatic TRS termination also involves expensive searches for suitable orderings.
- Quasi-termination: It can be seen, by adapting the PSPACE hardness construction, that the in-situ descent criterion used to decide quasi-termination in [12, 2, 10] is PSPACE hard.

  Glenstrup [9] shows one way that quasitermination analysis techniques can be used for termination. Simplified version: add a "recursion depth" parameter to

Domains	$v \in \mathit{Value}$ (a flat domain). $u, w \in \mathit{Value}^\sharp = \mathit{Value} \cup \{\bot, \mathit{Err}\}$, where $\bot \sqsubseteq w$ for all w.
Types	$\mathcal{E} : \mathit{Expr} \to \mathit{Value}^* \to \mathit{Value}^\sharp$ $\mathcal{O} : \mathit{Op} \to \mathit{Value}^* \to \mathit{Value}^\sharp$ $\mathit{lift} : \mathit{Value} \to \mathit{Value}^\sharp$ (the natural injection) $\mathit{strictapply} : (\mathit{Value}^* \to \mathit{Value}^\sharp) \to (\mathit{Value}^\sharp)^* \to \mathit{Value}^\sharp$
Semantic operator	$\mathcal{E}[\![\mathtt{f}^{(i)}]\!](v_1, \ldots, v_n) = \mathit{lift}\ v_i$ $\mathcal{E}[\![\mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3]\!]\vec{v} = \mathcal{E}[\![\mathtt{e}_1]\!]\vec{v} \to \mathcal{E}[\![\mathtt{e}_2]\!]\vec{v}, \mathcal{E}[\![\mathtt{e}_3]\!]\vec{v}$ $\mathcal{E}[\![\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\vec{v} = \mathit{strictapply}\ (\mathcal{O}[\![\mathtt{op}]\!])\ (\mathcal{E}[\![\mathtt{e}_1]\!]\vec{v}, \ldots \mathcal{E}[\![\mathtt{e}_n]\!]\vec{v})$ $\mathcal{E}[\![\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n)]\!]\vec{v} = \mathit{strictapply}\ (\mathcal{E}[\![\mathtt{e}^{\mathtt{f}}]\!])\ (\mathcal{E}[\![\mathtt{e}_1]\!]\vec{v}, \ldots \mathcal{E}[\![\mathtt{e}_n]\!]\vec{v})$
Auxiliary operations	$u \to w, w' = \begin{cases} u, & \text{if } u = \bot \text{ or } u = \mathit{Err}, \\ w, & \text{if } u = \mathit{True}, \\ w', & \text{otherwise.} \end{cases}$ $\mathit{strictapply}\ \psi\,(w_1, \ldots, w_n) = \begin{cases} \psi(v_1, \ldots, v_n) & \text{if } w_i \notin \{\bot, \mathit{Err}\} \text{ for } i = 1, \ldots, n, \\ & \text{and } w_i = \mathit{lift}\, v_i \text{ for each } i; \text{ else} \\ w_i & \text{where } i = \text{ least index such that } w_i \in \{\bot, \mathit{Err}\} \end{cases}$
Assumption	$\mathcal{O}[\![\mathtt{op}]\!]\vec{v} \neq \bot$

Figure 5.1: Semantics of L programs. True is a distinguished element of Value.

each function, incremented at every call. If depth parameters are bounded in every execution, then the program is terminating. This approach is weaker than size-change termination. And it appears that quasi-termination is in general a more difficult problem than termination.

For further discussion of related work see Appendix B.

5.2 Future work

A number of interesting problems to be investigated:

1. The PSPACE hardness result suggests trying to derive good approximations. A current goal is a PTIME **approximation of SCT**, which subsumes lexical orders, and handles permuted arguments and indirect recursion.
2. **An analogue of SCT for binding-time analysis** (to guarantee termination of partial evaluation) is being developed, generalizing the "in-situ" criterion used in [2, 9].
3. **Size analysis** should be incorporated for a practical analyzer, to construct more precise size-change graphs. Something based on the integer programming methods of [6, 19] may be appropriate.
4. An extension of the present approach for **high-order programs** seems possible, although the semantic analyses can become quite complicated if we wish to trace size relations involving arguments captured in closures.

6. ACKNOWLEDGEMENTS

The first author would like to express his gratitude to Professor Neil Jones for his generous and patient guidance, and to TOPPS for a rewarding academic visit in 1999.

APPENDIX

A. PROOF OF PSPACE-HARDNESS OF SCT

Here we prove that the reduction in Section 4 is correct, implying Theorem 5. First, some observations about a multipath of form $\mathcal{M}^{\mathcal{G}}(0\,cs)$:

1. Call 0 begins exactly two threads (to $\mathtt{X}$ and $\overline{\mathtt{X}}$) at every $\overline{\mathtt{X}}$ other than $\overline{\mathtt{X}}_k$. Parameter $\overline{\mathtt{X}}_k$ begins two such threads, and as well as one with initial arc directed to $\mathtt{Z}$.

   For each parameter $\overline{\mathtt{X}}$ of F_0, let us say that arc $\overline{\mathtt{X}} \xrightarrow{\downarrow} \mathtt{X}$ begins its "high thread", and that arc $\overline{\mathtt{X}} \to \overline{\mathtt{X}}$ begins its "low thread".

2. Both high thread and low thread continue the length of $0cs$, and end in either $\mathtt{X}$ or $\overline{\mathtt{X}}$. The (unique) $\mathtt{Z}$-thread remains in $\mathtt{Z}$ after the initial arc for the whole of $0cs$. By the way $\mathtt{p}$ is constructed, no thread is ever lost or duplicated, except at call 0.

3. Either the high or the low thread from $\overline{\mathtt{X}}$ must end at $\overline{\mathtt{X}}$, so $0cs$ has a thread from every $\overline{\mathtt{X}}$ to itself.

LEMMA 9. *Suppose $b \vdash (\ell_1, \sigma_1) \to \ldots \to (\ell_t, \sigma_t)$ with trace $cs = c_1 c_2 \ldots c_{t-1}$. Then for any variable $\mathtt{X}$, multipath $\mathcal{M}^{\mathcal{G}}(0cs)$ has a nondescending thread from parameter $\overline{\mathtt{X}}$ to parameter $\overline{\mathtt{X}}$ if $\sigma_t(\mathtt{X}) = \mathit{false}$, and from $\overline{\mathtt{X}}$ to $\mathtt{X}$ if $\sigma_t(\mathtt{X}) = \mathit{true}$.*

PROOF. Immediate if $t = 1$, since $\sigma_1(\mathtt{X}) = \mathit{false}$ for all $\mathtt{X}$. Inductively, if true for $t-1$, inspection of the cases in construction of $\mathtt{p}$ based on the form of instruction $\mathtt{I}_{\ell_{t-1}}$ shows that the property is preserved. $\square$

To prove that the reduction is correct, we must show that Boolean program b terminates if and only if program $\mathtt{p}$ is *not* size-change terminating.

Only if: Suppose b terminates with (finite) trace cs. Then $0cs$ is a valid call sequence for p, from F_0 back to F_0. By Lemma 9 the "low thread" starting at any F_0 parameter $\overline{\mathtt{X}}_i$ is nondescending. By the assumption that all variables are *false* when execution ends, the low thread returns to $\overline{\mathtt{X}}_i$. Thus the high thread ends in $\mathtt{X}_i$, and fails to be continued if call 0 is repeated following $0cs$. The Z-thread will also be discontinued. Consequently $(0cs)^\omega \notin DESC^\omega$, which implies $DESC^\omega \neq FLOW^\omega$.

If: we show that if b fails to terminate, any $cs \in FLOW^\omega$ must have infinite descent. Now cs must begin with call $0 : F_0 \to F_1$ from p's initial function F_0. If cs contains only a finite number of calls to F_0, then after the last one, parameter Z will decrease in every call, so $cs \in DESC^\omega$.

Suppose cs contains an infinite number of calls to F_0. This means we can decompose it into $cs = 0cs_1 0cs_2 0cs_3 \ldots$ where each cs_i is free of 0's. We claim that each $0cs_i$ has a descending thread from at least one parameter $\overline{\mathtt{X}}$ to $\overline{\mathtt{X}}$ (the claim is proved below). By Observation 3, every cs_i has continuous threads beginning and ending in every parameter $\overline{\mathtt{Y}}$, so call sequence $cs = 0cs_1 0cs_2 0cs_3 \ldots$ has exactly k infinite maximal threads. Infinitely many $\downarrow$ must occur in this set of threads. Consequently at least one thread within cs contains infinitely many $\downarrow$, so $cs \in DESC^\omega$.

Proof of claim: Let $cs_i = c_1 c_2 \ldots c_n$; since $0cs_i0$ is part of a valid call sequence from $FLOW^\omega$, but is not a correct trace of the program (which does not terminate), we conclude that it includes a call that represents the incorrect branch of an `if` statement. Consider the first such call. Being incorrect means that the else-branch is followed when the tested variable X has value *true*, or the then-branch is followed while the value is *false*. By Lemma 9 the low thread starting at $\overline{\mathtt{X}}$ reaches $\overline{\mathtt{X}}$ if X has value *false* and reaches X if X has value *true*. In each case, the size-change graph created by our construction for the *wrong* branch extends this thread with a $\downarrow$-labeled arc.

Since the high thread from $\overline{\mathtt{X}}$ always contains a $\downarrow$-labeled arc, we conclude that the thread from $\overline{\mathtt{X}}$ that happens to return to $\overline{\mathtt{X}}$ at the end of $0cs_i$ (being one of these two) must be descending, as desired.

B. MORE ON RELATED WORK

Logic programs

There has been extensive research on automatic termination analysis for logic programs. As explained in [17], it is not always obvious that a predicate will terminate when executed with unusual instantiation patterns, or that a predicate always terminates on backtracking. For interpreters that have a choice of evaluation orders, termination analysis is especially important.

Some analyses that have been described for logic programs (e.g., in [15, 19]) use a simple criterion: for every recursive invocation of a predicate, determine that the sum over a subset of input fields (fixed for each predicate) is strictly decreased. This does not allow handling of lexical descent.

The strength of these methods derives from aggressive size analysis, which enables, in particular, sorting routines (quicksort and insertion sort) to be handled automatically. It is also possible to incorporate size analysis into the present approach, but the aim of this article has been to investigate the size-change termination principle by itself.

There are also logic program termination analyzers using a termination criterion compatible with size-change termination [14, 7]. The analysis in [17] has been extended to a termination analyzer for Prolog programs called *Termilog* [14]. It turns out that Termilog can be used to solve size-change termination problems precisely via a suitable encoding. In fact, our graph-based algorithm, although devised independently, is in essence a functional programming counterpart of the Termilog algorithm. This means that the PSPACE hardness result of Section 4 applies to Termilog's Analysis.

All the works on Prolog termination that we are aware of devote much attention to orthogonal issues such as instantiation and size analysis. While these are no doubt important considerations in practice, an impression is created that the complexity of the termination problem for Prolog stems from these concerns. The significance of the complexity result in this article is pointing out that the core *size-change termination principle* is intrinsically hard.

Term rewriting systems

One application for term rewriting systems is to model the semantics of functional programs. A functional program is easily translated into a TRS such that termination of the TRS implies termination of the subject program. Unfortunately, the resulting TRS is often *non-simply-terminating*, which means the usual approach to find an ordering for which the LHS of each rewrite rule is strictly greater than the RHS, does not work. To treat such TRS, Arts [3] applied programming intuition to develop methods sufficiently strong for them. For a term-rewriting perspective, these methods are able to a handle a larger class of TRS. From the point of view of analyzing functional programs, a dataflow approach may be less circuitous.

For TRS termination, it is common to perform expensive searches for a suitable ordering to solve a set of inequalities. For instance, in [20], a heuristic is given for automatically generating a general class of orderings known as *transformation orderings*, which includes the lexical order. In the present work, it has not been the aim to *look for* orderings. Size-change termination naturally subsumes an interesting class of orderings, including the lexical ordering, and the ordering for the example with permuted and discarded parameters, which is not obvious.

Finally, for TRS corresponding to programs, the polynomial interpretation method for discovering orderings [8] obviates the need for size analysis by appropriately interpreting function symbols in the subject program. The approach in this article has been to factor out size analysis as an orthogonal concern, and focus on the size-change termination principle and its application. This appears to be a natural factoring of concerns when analyzing termination of programs.

C. REFERENCES

[1] Andreas Abel and Thorsten Altenkirch. A semantical analysis of structural recursion. In *Abstracts of the Fourth International Workshop on Termination WST'99*, pages 24–25. unpublished, May 1999.

[2] Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher order functional language. In *Static Analysis, Proceedings of the Third International Symposium, SAS '96, Aachen, Germany, Sep 24–26, 1996*, volume 1145 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 1996.

[3] Thomas Arts. *Automatically Proving Termination and Innermost Normalisation of Term Rewriting Systems.* PhD thesis, Universiteit Utrecht, 1997.

[4] Thomas Arts and Jürgen Giesl. Proving innermost termination automatically. In *Proceedings Rewriting Techniques and Applications RTA'97*, volume 1232 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 1997.

[5] Anders Bondorf. Similix manual. Technical Report 91/9, DIKU, University of Copenhagen, Denmark, 1991.

[6] Wei Ngan Chin and Siau Cheng Khoo. Calculating sized types. In Julia Lawall, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, Boston, Mass., USA*. ACM, 2000.

[7] 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., September 3–5, 1997*, volume 1298 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 1997.

[8] Jürgen Giesl. Termination analysis for functional programs using term orderings. In Alan Mycroft, editor, *Proc. 2nd Int'l Static Analysis Symposium (SAS), Glasgow, Scotland*, volume 983 of *Lecture Notes in Computer Science*, pages 154–171. Springer-Verlag, September 1995.

[9] Arne J. Glenstrup. Terminator II: Stopping partial evaluation of fully recursive programs. Master's thesis, DIKU, University of Copenhagen, Denmark, 1999.

[10] Arne J. Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics, Proceedings of the Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, Jun 25–28, 1996*, volume 1181 of *Lecture Notes in Computer Science*, pages 273–284. Springer, 1996.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM (CACM)*, 12(10):576–580, October 1969.

[12] Carsten Kehler Holst. Finiteness analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 473–495. Springer, 1991.

[13] Neil D. Jones. *Computability and Complexity From a Programming Perspective.* Foundations of Computing Series. MIT Press, 1997.

[14] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic termination analysis of Prolog programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 64–77, Leuven, Belgium, Jul 1997. MIT Press.

[15] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1990.

[16] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science,* pages 319–327, IEEE, 1988.

[17] Yehoshua Sagiv. A termination test for logic programs. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct 28–Nov 1, 1991*, pages 518–532. MIT Press, 1991.

[18] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[19] Chris Speirs, Zoltan Somogyi, and Harald Søndergaard. Termination analysis for Mercury. In Pascal Van Hentenryck, editor, *Static Analysis, Proceedings of the 4th International Symposium, SAS '97, Paris, France, Sep 8–19, 1997*, volume 1302 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1997.

[20] Joachim Steinbach. Automatic termination proofs with transformation orderings. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, Proceedings of the 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995*, volume 914 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 1995.