# All-Termination($SCP$)

Panagiotis Manolios and Aaron Turon

Northeastern University {pete,turon}@ccs.neu.edu

**Abstract.** We recently introduced the All-Termination($T$) problem: given a termination solver $T$ and a function $F$, find every subset of the formal parameters to $F$ whose consideration is sufficient to show, using $T$, that $F$ terminates. These subsets can be harnessed by a theorem prover to locate and justify induction schemes, and are also useful for guiding rewriting heuristics and ensuring their termination. In this paper, we study the All-Termination problem for *SCP* (polynomial size-change analysis), a powerful, cubic-time termination analysis. *SCP* is the first *nonmonotonic* termination analysis studied in the context of All-Termination, making its analysis both challenging and informative. We develop an algorithm for solving the All-Termination(*SCP*) problem, and briefly report on initial experimental results obtained on the ACL2 regression suite.

## 1 Introduction

Termination analysis is a crucial tool for program verification. Usually, the question is simply whether a given program terminates. But casting termination as a decision problem means that facts discovered during analysis are discarded once the program is known to terminate. For example, consider the function:

```
define insert(i, item, list) =
   if i = 0 then cons(item, list)
            else cons(car(list), insert(i - 1, item, cdr(list)))
```

We can prove that `insert` terminates by showing that any of `length(list)`, `i`, or `length(list) + i` decrease each time `insert` recurs. From the standpoint of termination, one decreasing measure is as good as another. From a broader standpoint, however, distinctions between measures have important implications. Data about the available measures can be harnessed after termination analysis to, for example, derive induction schemes and guide rewriting heuristics.

With each recursive function $F$ we can associate a collection of *measurable sets*. A measurable set is a subset of the formal parameters of $F$ for which there exists a measure function; the value of the measure function on the measured parameters must decrease over each recursive call in $F$. Since measure values are well-founded, infinite descent—and hence nontermination of $F$—is impossible.

In a previous paper, we introduced the All-Termination problem [1]: given a function $F$, enumerate all its measurable sets. All-Termination is a generalization of the classic termination decision problem, since a program is terminating iff it has at least one measurable set. Hence the problem is undecidable

in general. However, decades of work on termination have yielded powerful, but decidable, termination analyses. For any such termination analysis, $T$, we can pose the ALL-TERMINATION($T$) problem: given a function $F$ and a termination solver, $T$, find as many measurable sets for $F$ as possible, using $T$.

The usefulness of measurable sets has been recognized for at least 30 years: they are a central component of the Boyer-Moore family of theorem provers [2], which includes ACL2 [3]. To motivate ALL-TERMINATION analysis, we briefly illustrate two ways Boyer and Moore use measurable sets. First, note that

$$\text{ALL-TERMINATION}(\texttt{insert}) = \left\{ \begin{array}{c} \{\texttt{i}\},\ \{\texttt{list}\},\ \{\texttt{i},\texttt{list}\}, \\ \{\texttt{item},\texttt{i}\},\ \{\texttt{item},\texttt{list}\},\ \{\texttt{item},\texttt{i},\texttt{list}\} \end{array} \right\}.$$

Suppose that we ask an automated theorem prover to prove the conjecture

$$\langle \forall \mathit{list}, \mathit{item} :: \texttt{insert(0, } \mathit{item}\texttt{, } \mathit{list}\texttt{)} = \texttt{cons(}\mathit{item}\texttt{, } \mathit{list}\texttt{)} \rangle$$

A basic technique the theorem prover might apply is *simplification*, replacing subexpressions of the theorem with equivalent, but "simpler," subexpressions. Intuitively, the subexpression `insert(0,` *item*`,` *list*`)` can be simplified by expanding the definition of `insert`, while the subexpression `cons(`*item*`,` *list*`)` cannot. How do we turn this intuition into a formal heuristic?

In Boyer-Moore provers, all functions must terminate on all arguments. It is therefore tempting to consider an aggressive heuristic in which *any* application of a function is replaced by the function's body, substituting actual arguments for the formal parameters. The problem is that this heuristic does *not* guarantee that the simplification process itself terminates! For example, applying the heuristic to the expression `insert(`$i$`, "foo",` *list*`)` results in an infinite series of such "simplifications." On the other hand, the expression `insert(0,` *item*`,` *list*`)` is successfully simplified, using the heuristic, to `cons(`*item*`,` *list*`)`.

Boyer and Moore suggest capturing our intuition with a heuristic based on measurable sets [2, §8.2]. They first distinguish "explicit values," such as `true` or `cons("foo", nil)`, from other expressions, such as variables or function applications. Suppose a function $F(x_1, \ldots, x_n)$ is defined, and has a measurable set $P$ of formal parameters. The Boyer-Moore heuristic expands any application $F(e_1, \ldots, e_n)$ of the function where, for each $x_i \in P$, the expression $e_i$ is an explicit value. According to this heuristic, `insert(0,` *item*`,` *list*`)` can be simplified, because $\{i\}$ is a measurable set of `insert`, but `insert(`$i$`, "foo",` *list*`)` cannot be simplified, because $\{\texttt{item}\}$ is not a measurable set of `insert`. The heuristic also allows iterated simplifications, for example discovering that

```
insert(2, item, list) = cons(car(list),
                             cons(car(cdr(list)), cons(item, cdr(cdr(list)))))
```

Each time a function $F$ is recursively expanded, a measurable set of its arguments have been simplified to explicit values. Because the arguments are measurable, there is some measure function whose value on those arguments decreases for each expansion. In effect, the measure function used to show $F$'s termination is *lifted* to guarantee the termination of simplification involving $F$. Remarkably,

the particular measure function is not important; the only data needed to guide simplification is $F$'s collection of measurable sets, *i.e.*, All-Termination($F$).

Another application of measurable sets is in the derivation and justification of induction schemes. Consider the function `less` on natural numbers [2, §14.2]:

```
define less(i, j) = if j = 0 then false
                    else if i = 0 then true
                                  else less(i-1, j-1)
```

In order to prove $\langle \forall x, y :: \texttt{less}(x\texttt{+}y, \ y) = \texttt{false} \rangle$, we would probably use induction on $y$, which requires proving, for all $x, y$, that

$$\texttt{less}(x\texttt{+0, 0}) = \texttt{false}, \text{ and}$$
$$y > 0, \texttt{less}(x\texttt{+}y\texttt{-1, }y\texttt{-1}) = \texttt{false} \implies \texttt{less}(x\texttt{+}y\texttt{, }y) = \texttt{false}$$

Is it possible to mechanically discover such an induction scheme, given the function definition and theorem we wish to prove? Boyer and Moore develop a heuristic for doing so, again based on measurable sets. The criterion is in a sense the opposite of that for simplification: a conjectured theorem about a function $F$ "suggests" an induction scheme when, for some measurable set of $F$, all measured arguments in the theorem are *variables*. Roughly speaking, the conjectured theorem serves as an induction hypothesis, and the measured arguments serve as the variables on which we are inducting. The induction scheme itself is derived from the bodies of functions appearing in the theorem—a process too complicated to describe here. The key point is that any use of the induction hypothesis is guided by the recursion present in a function's body. Because the measurable set of arguments in the induction hypothesis appear as variables, the hypothesis can always be instantiated with the appropriate values of those arguments for a recursive call. The fact that a measure exists for those arguments ensures the soundness of the induction scheme.

The measurable sets of `less` are $\{\texttt{i}\}$, $\{\texttt{j}\}$, and $\{\texttt{i}, \texttt{j}\}$. In the conjectured theorem, any induction involving the first parameter of `less` is ruled out, because the argument given is not a variable, but rather the expression $x\texttt{+}y$. On the other hand, the second argument is indeed a variable, and because $\{\texttt{j}\}$ is a measurable set, we can soundly derive the induction scheme given above.

**Related work.** The termination problem dates back to Turing, who called it the "Printing Problem" [7], and there has been steady interest in it ever since. Here we can only briefly touch upon the work most directly related to ours; a lengthier discussion of the literature can be found in [1]. The idea of All-Termination can be traced back to Boyer and Moore's work [2], which also provided the impetus for our work. However, the approach they used to find measurable subsets just iterates over their termination analysis in the naive way: it has exponential complexity and little in common with the work presented here, beyond the initial motivation. Besides our initial paper [1], we know of no other work studying All-Termination.

This paper focuses on size-change termination analysis [5], which was introduced in the setting of an applicative language and has since served as a

framework for several other analyses. This includes work on termination in term-rewrite systems that combines size-change analysis with the dependency pair method and recursive path orderings [8]. Tools based on these ideas include AProVE [9]. Another example is work on calling context graphs and measures, which is used to prove termination of functional programs [10], and has been implemented in ACL2s [11] and Isabelle [12].

**Contributions and outline.** We introduce All-Termination($SCP$), where $SCP$ is polynomial-time size-change analysis [4]. That is, we adapt the $SCP$ termination analysis to yield a collection of measurable sets.

We start in section 2, by formalizing our model of programs and defining All-Termination($T$). Section 3 gives the necessary background on size-change analysis [5]. We previously studied its All-Termination problem [1]. However, size-change termination is PSpace-hard, and exponential behavior for the standard algorithms can easily be triggered. This fact led Ben-Amram and Lee to develop a cubic-time approximation to size-change, called $SCP$ [4]. Surprisingly, $SCP$ is as powerful as full size-change analysis in practice, even though it is on average an order of magnitude faster (see Section 5). This fact motivates the study of All-Termination($SCP$). As it turns out, All-Termination($SCP$) is a significantly more complicated problem, and the algorithm we develop has little in common with our earlier work.

Section 4 is the main contribution of the paper. We carefully develop an appropriate treatment of $SCP$, via a formal system, that allows us to develop an All-Termination algorithm for it. We then show how to transform this formal system into a collection of boolean constraints. The minimal models of the resulting constraints are exactly the results of All-Termination($SCP$). We give an algorithm, by reduction to *dual-horn minimization* [6], for enumerating these models. The algorithm is *output-sensitive*, meaning that its runtime is bounded by the size of its output. This property is desirable because the size of the output is usually quite small (see Section 5).

Section 5 reports initial experimental results on the ACL2 regression suite, consisting of over 11,000 functions. We have implemented full size-change analysis and our earlier All-Termination($SCT$) analysis [1], finding that 90% of multiargument functions have at least one nontrivial measurable set, and 7% of them have multiple, incomparable sets. We have also implemented $SCP$ analysis and compared its performance to full size-change. Implementation of the All-Termination($SCP$) algorithm in this paper is under way.

## 2 All-Termination($T$)

We informally think of a program $F$ as a mutually-recursive nest of functions, but we model programs by their *semantic call graphs*. Given a universe of function names $\mathcal{F}$, parameter names $\mathcal{P}$, and values $\mathcal{V}$, we say:

**Definition 1** *A **semantic call graph** $\mathcal{C}$ is a pair $(S, \rightarrow)$ with $S \subseteq \mathcal{F} \times (\mathcal{P} \rightharpoonup \mathcal{V})$ the set of **states** and $\rightarrow \subseteq S \times S$ the **transition relation**.*

The elements of $\mathcal{P} \rightharpoonup \mathcal{V}$ are the partial functions from $\mathcal{P}$ to $\mathcal{V}$. A semantic call graph records the sequence of function calls while computing a given function application. A state $(f, \{(x, 3), (y, 1)\})$ in a semantic call graph represents an invocation of the function $f(x, y)$ with arguments 3 and 1. If a call $f(3, 1)$ results in a call $g(1)$, there is an edge between the respective states. With every program $F$, we associate a semantic call graph $\mathcal{C}_F$.

**Definition 2** *A semantic call graph $\mathcal{C}$ is **terminating** if it contains no infinite sequence of transitions $s_1 \rightarrow s_2 \rightarrow \cdots$.*

The graph $\mathcal{C}_F$ is terminating iff every function in $F$ terminates on every possible input. We can express termination of semantic call graphs in terms of measure functions, the standard tool for proving termination:

**Proposition 1** *$(S, \rightarrow)$ is terminating iff there exists a well-ordered set $(W, >)$ and a **measure** $\mu$, i.e., a map $\mu : S \rightarrow W$ such that if $s \rightarrow t$ then $\mu(s) > \mu(t)$.*

If $(f, V)$ is a state in a semantic call graph $\mathcal{C}$, the values of the formal arguments in $\mathrm{dom}(V)$ are the observations available to a measure on $\mathcal{C}$. Thus, we can force a measure to ignore certain arguments by restricting the domain of $V$:

**Definition 3** *Given $V : \mathcal{P} \rightharpoonup \mathcal{V}$, $f \in \mathcal{F}$ and $P \subseteq \mathcal{P}$, we define the **restrictions***

$$(V \restriction P)(x) = \begin{cases} V(x) & x \in \mathit{dom}(V) \cap P, \\ \text{undefined} & \mathit{otherwise} \end{cases} \qquad (f, V) \restriction P = (f, V \restriction P)$$

Informally, a set of formal parameter names $P$ is measurable if there is a measure that "uses" only those arguments. We can formalize this idea using restriction.

**Definition 4** *$P \subseteq \mathcal{P}$ is a **measurable set** for $\mathcal{C} = (S, \rightarrow)$ if there exists a measure $\mu : S \rightarrow W$ such that $\mu(s) = \mu(t)$ whenever $s \restriction P = t \restriction P$.*

Note that $\mathcal{C}$ is terminating iff it has a measurable set. Termination analyses are usually formulated so that they imply the termination of a program, but not the existence of any particular measurable set. To define All-Termination$(T)$, we limit the analysis $T$ to prove termination using only the set of parameters $P$:

**Definition 5** *A **restricted termination analysis** $T_{\mathsf{R}}$ is a predicate such that, if $T_{\mathsf{R}}(F, P)$, then $P$ is a measurable set for $\mathcal{C}_F$.*

It is not possible to say, in general, how to take a termination analysis $T$ and derive a restricted termination analysis $T_{\mathsf{R}}$, but a reasonable constraint is that $T(F) \iff \langle \exists P :: T_{\mathsf{R}}(F, P) \rangle$. In Section 4, we will see that even this simple constraint can be subtle in practice.

Given a restricted termination analysis $T_{\mathsf{R}}$ and a program $F$, we define

$$\textsc{All-Termination}(T_{\mathsf{R}})(F) = \mathrm{minimal}\{P \subseteq \mathcal{P} \ : \ T_{\mathsf{R}}(F, P)\}.$$

The analysis yields only the minimal sets—the *termination cores*—because the collection of measurable sets for a function is upward-closed under set inclusion.

## 3 The size-change framework

Working directly with measure functions is difficult, because measures are *global*: they must decrease in value over every recursive call in the function they measure. The *size-change framework* of Lee, Jones, and Ben-Amram [5] finesses this issue by constructing a graph that describes *local* changes in size, and then analyzing the global behavior of the graph. We briefly review this framework.

An *annotated call graph* (ACG) is a directed graph with function names as nodes, and an edge from $f$ to $g$ for each call to $g$ that occurs in the body of $f$. The edges of an ACG are annotated with *size-change graphs*, which record the size relationship between the arguments of $f$ and $g$. More formally, we have

| | | |
|---|---|---|
| $p, q, r \in \text{LAB}$ | $= \{>, \geq\}$ | size-change labels |
| $G, H \in \text{SCG}$ | $= 2^{\mathcal{P} \times \text{LAB} \times \mathcal{P}}$ | size-change graphs |
| $\mathcal{G}, \mathcal{H} \in \text{ACG}$ | $= 2^{\mathcal{F} \times \text{SCG} \times \mathcal{F}}$ | annotated call graphs |

We write $x \xrightarrow{r} y$ for $(x, r, y) \in G$ and $f \xrightarrow{G} g$ for $(f, G, g) \in \mathcal{G}$. We also sometimes write $G \in \mathcal{G}$ for $f \xrightarrow{G} g$ if the function names $f$ and $g$ are unimportant.

For simplicity, we postulate a single well-ordering $>$ on all values in $\mathcal{V}$. ACGs are related to semantic call graphs in an obvious way:

**Definition 6** *An ACG $\mathcal{G}$ is **safe** for $\mathcal{C}$ if, whenever $(f, V) \rightarrow (g, U) \in \mathcal{C}$, there is an edge $f \xrightarrow{G} g \in \mathcal{G}$ such that $x \xrightarrow{r} y \in G$ implies $V(x)\ r\ U(y)$.*

Thus, $\mathcal{G}$ is safe for $\mathcal{C}$ just when it accurately describes every possible change in argument size in $\mathcal{C}$. For example, the ACGs for both `insert` and `less` (Section 1) have a single node (labeled `insert` and `less` respectively) and a single self-edge for that node. The size-change graphs labeling the self-edges are:

`insert` $\xrightarrow{G_1}$ `insert`: $\boxed{\begin{array}{c} i \xrightarrow{>} i \\ item \xrightarrow{\geq} item \\ list \xrightarrow{>} list \end{array}}$ `less` $\xrightarrow{G_2}$ `less`: $\boxed{\begin{array}{c} i \xrightarrow{>} i \\ j \xrightarrow{>} j \end{array}}$

Constructing an ACG for a program is a challenging problem that the size-change framework does not address (but see [10]). We will simply assume the existence of a function *analyze* such that $\mathit{analyze}(F)$ is safe for $\mathcal{C}_F$. The safety condition means that any infinite path through $\mathcal{C}_F$ would entail the existence of an infinite "multipath" through $\mathit{analyze}(F)$.

**Definition 7** *A **multipath** $\pi$ through an ACG $\mathcal{G}$ is a (potentially infinite) sequence of edges from $\mathcal{G}$, connected at nodes: $\pi = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \xrightarrow{G_3} \cdots$.*

We write $\mathcal{G}^\omega$ for the set of nonempty multipaths over $\mathcal{G}$ and $\mathcal{G}^+$ for the set of finite, nonempty ones. We sometimes write $G_1, G_2, \ldots$ or $\langle G_i \rangle$ to describe a multipath when the function names are irrelevant.

The reason $\pi = \langle G_i \rangle$ is a *multipath* and not just a path is that the elements $G_i$ of the sequence are themselves graph structures. In particular, a multipath may contain many *threads* through its size-change graphs.

**Definition 8** *A **thread** in a multipath $\pi = \langle G_i \rangle$ is a sequence of size-change edges $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ such that $x_{i-1} \xrightarrow{r_i} x_i \in G_i$ for all $i > 0$.*

A thread abstractly tracks a given value as it flows through the arguments of successive function calls. A value being tracked by a thread can never increase, but it must decrease any time it passes through a $>$-labeled size-change edge. Infinite recursions in $\mathcal{C}_F$ are ruled out by analyzing the infinite multipaths of $\mathit{analyze}(F)$. If every such multipath contains an infinite thread marked infinitely-often with $>$, then any infinite path in $\mathcal{C}_F$ would involve an actual value decreasing infinitely. By well-foundedness this situation is impossible, so $\mathcal{C}_F$ must terminate.

**Definition 9**

*(1) A thread $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ has **infinite descent** if $r_i = >$ for infinitely-many $i$.*
*(2) A multipath $\pi$ has **infinite descent** if it has a thread with infinite descent.*
*(3) $\mathcal{G}$ is **size-change terminating** if every infinite multipath $\pi \in \mathcal{G}^\omega$ has a suffix with infinite descent.*

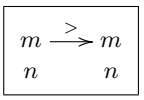
It should be clear that the example ACGs above are size-change terminating.

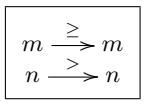
Deciding whether a given $\mathcal{G}$ is size-change terminating is a PSPACE-complete problem [5], and unfortunately exponential-time behavior is easy to trigger. However, Ben-Amram and Lee developed a cubic-time algorithm approximating size-change termination, known as *SCP* [4]. In practice, *SCP* is almost always as powerful as the PSPACE algorithm (see Section 5). *SCP* is based on the following notion of loop anchors.

**Definition 10** *$G \in \mathcal{G}$ is an **anchor** (for $\mathcal{G}$) if every $\pi \in \mathcal{G}^\omega$ in which $G$ appears infinitely often has a suffix with infinite descent.*

To illustrate the idea, we consider Ackermann's function:

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

$G_1$: 


$G_2$: 


The function is abstracted as an ACG $\mathcal{G}_{\texttt{ack}}$ with one node, labeled `ack`, and two self-edges, labeled with the size-change graphs $G_1$ and $G_2$. Notice that there are three recursive calls in the body of `ack`. The call in the second line and the outer call in the third line are both safely abstracted by the single size-change graph $G_1$; the remaining call is abstracted by $G_2$. Thus, any infinite recursion of `ack` would correspond to an infinite multipath $\pi \in \mathcal{G}^\omega_{\texttt{ack}}$, and we can show that any such $\pi$ has a suffix with infinite descent:

– Suppose $G_1$ appears infinitely often in $\pi$. We can track the size of $m$ through $\pi$. Every time $\pi$ goes through $G_2$ the value of $m$ does not increase, and infinitely often $\pi$ goes through $G_1$, where the value of $m$ must decrease. Thus $\pi$ has infinite descent, and thus $G_1$ is an anchor for $\mathcal{G}_{\texttt{ack}}$.

– Otherwise, $G_1$ appears only finitely-many times in $\pi$, which means that $\pi$ has an infinite suffix $\pi'$ in which $G_1$ never appears. Note that $\pi' \in (\mathcal{G}_{\texttt{ack}} \setminus \{G_1\})^\omega$. Tracking the size of $n$ through $\pi'$ is easy: since $\pi'$ just goes through $G_2$ infinitely, the value of $n$ decreases infinitely. Thus $\pi'$ has infinite descent, and thus $G_2$ is an anchor for $\mathcal{G}_{\texttt{ack}} \setminus \{G_1\}$.

The `ack` example gives the general flavor of *SCP*, which rules out infinite loops by locating and removing anchors for those loops. Let $\mathrm{SCC}(\mathcal{G})$ denote the set of nontrivial, strongly-connected components of $\mathcal{G}$, and note that each element of $\mathrm{SCC}(\mathcal{G})$ is another annotated call graph. *SCP* is defined as follows.

**Algorithm 1 (Ben-Amram, Lee [4])**

$SCP(\mathcal{G})$ : *for* $\mathcal{H} \in \mathrm{SCC}(\mathcal{G})$ *do*
$\mathcal{A} := \textsc{FindAnchors}(\mathcal{H})$
*if* $\mathcal{A} = \emptyset$ *or* $SCP(\mathcal{H} \setminus \mathcal{A}) =$ *False then return False*
*return True*

Note that a size-change graph $G$ might fail to be an anchor in one iteration, but become an anchor in a later iteration, after other size-change graphs have been removed from $\mathcal{G}$. The key element of the algorithm, of course, is the implementation of FindAnchors.

**Theorem 1 (Ben-Amram, Lee [4])** *If* $\textsc{FindAnchors}(\mathcal{G})$ *returns a set of anchors of* $\mathcal{G}$*, then SCP soundly approximates size-change termination. If it returns* all *the anchors of* $\mathcal{G}$*, then SCP decides size-change termination.*

Ben-Amram and Lee found two conditions on a size-change graph $G \in \mathcal{G}$, either of which is sufficient to show $G$ to be an anchor for $\mathcal{G}$, but neither of which is necessary. The basis for these two conditions is the notion of a *thread preserver*. Letting $\mathrm{src}(G) = \{x \ : \ \langle \exists r, y :: x \xrightarrow{r} y \in G \rangle\}$, we define:

**Definition 11** *A set* $P \subseteq \mathcal{P}$ *is a* ***thread preserver*** *for* $\mathcal{G}$ *if for any* $G \in \mathcal{P}$ *and* $x \in \mathrm{src}(G) \cap P$ *there is some edge* $x \xrightarrow{r} y \in G$ *with* $y \in P$*. We write* $\mathrm{TP}(\mathcal{G})$ *for the set of thread preservers for* $\mathcal{G}$*.*

The usefulness of thread preservers is illustrated by the following:

**Proposition 2** *If* $P \in \mathrm{TP}(\mathcal{G})$ *and* $\langle G_i \rangle \in \mathcal{G}^\omega$ *is a multipath with* $\mathrm{src}(G_0) \cap P \neq \emptyset$*, then there is a thread in* $\langle G_i \rangle$ *staying within* $P$*.*

This proposition is particularly relevant when applied to infinite multipaths, since we can then apply it to find infinite suffixes of the multipath in which some value never increases. We cannot use thread preservers alone to find an infinite decrease, however, since a thread resulting from a thread preserver might be labeled with only $\geq$ edges. The purpose of the two anchor conditions below is to ensure that an infinite decrease occurs. Before we can define them, we need one additional definition, giving the *restriction* of an ACG to a set of parameters.

**Definition 12** *Given $G$, $\mathcal{G}$, and $P$, we define the* ***restrictions***

$$G \restriction P = \{x \xrightarrow{r} y \in G \ : \ x, y \in P\} \qquad \mathcal{G} \restriction P = \{f \xrightarrow{G \restriction P} g \ : \ f \xrightarrow{G} g \in \mathcal{G}\}$$

The first approach to proving that $G$ is an anchor is to ensure that threads passing through $G$ under a thread preserver $P$ must always go through a *strict* edge (one labeled by $>$) within $G$. This can be accomplished as follows.

**Definition 13**

*(1) A size-change graph $G$ has* ***strict fan-in*** *if whenever two edges $x \xrightarrow{p} z$ and $y \xrightarrow{q} z$ with $x \neq y$ are in $G$, then $p = q = >$.*
*(2) An ACG $\mathcal{G}$ has* ***strict fan-in*** *if each $G \in \mathcal{G}$ has strict fan-in.*
*(3) A size-change graph $G \in \mathcal{G}$ is a* ***type-1 anchor*** *for $\mathcal{G}$ with respect to $P \in \mathrm{TP}(\mathcal{G})$ if $\mathcal{G} \restriction P$ has strict fan-in and there is some strict edge in $G$.*

The second approach is to rule out edges $x \xrightarrow{\geq} y \in G$ such that there is a thread taking $y$ back to $x$ without passing through a strict edge. These edges $x \xrightarrow{\geq} y$ represent the first step of a possible infinite thread that loops through $x$ without ever decreasing. We write $y \xrightarrow{\geq} x \in \pi$ if there is a thread in $\pi$ from $y$ to $x$ passing only through $\geq$-labeled edges.

**Definition 14**

*(1) The* ***no-descent set*** *is* $\mathrm{ND}(\mathcal{G}) = \left\{x \xrightarrow{\geq} y \in G \in \mathcal{G} \ : \ \left\langle \exists \pi \in \mathcal{G}^+ :: y \xrightarrow{\geq} x \in \pi \right\rangle\right\}$.
*(2) Let $\mathcal{G} \rhd G = (\mathcal{G} \setminus G) \cup \{G \setminus \mathrm{ND}(\mathcal{G})\}$, which removes $G$'s problematic edges.*
*(3) A size-change graph $G \in \mathcal{G}$ is a* ***type-2 anchor*** *for $\mathcal{G}$ if there exists a $P \in \mathrm{TP}(\mathcal{G} \rhd G)$ with $P \cap \mathrm{src}(G) \neq \emptyset$.*

It is also useful to consider anchors for the *transposition* of an ACG.

**Definition 15** *We define* ***transpositions***

$$G^t = \{y \xrightarrow{r} x \ : \ x \xrightarrow{r} y \in G\} \qquad \mathcal{G}^t = \{g \xrightarrow{G^t} f \ : \ f \xrightarrow{G} g \in \mathcal{G}\}$$

**Proposition 3** *$G$ is an anchor for $\mathcal{G}$ iff $G^t$ is an anchor for $\mathcal{G}^t$.*

Despite the fact that *in general* anchors for $\mathcal{G}$ and $\mathcal{G}^t$ are in one-to-one correspondence, it is possible for $G^t$ to be, *e.g.*, a type-1 anchor for $\mathcal{G}^t$ even though $G$ is not a type-1 anchor for $\mathcal{G}$. Hence, we look for anchors in both $\mathcal{G}$ and $\mathcal{G}^t$.

Ben-Amram and Lee show that deciding whether $G \in \mathcal{G}$ is a type-1 anchor is an NP-complete problem. The reason for this high complexity is that finding a thread preserver $P \in \mathrm{TP}(\mathcal{G})$ that has strict fan-in is NP-hard. However, thread preservers are closed under union; hence, there is a maximum thread preserver.

**Definition 16** *The* ***maximum thread preserver*** *for $\mathcal{G}$ is* $\mathrm{MTP}(\mathcal{G}) = \bigcup \mathrm{TP}(\mathcal{G})$.

Checking whether $G \in \mathcal{G}$ is a type-1 anchor *with respect to* $\mathrm{MTP}(\mathcal{G})$ can be done in linear time, and checking whether $G \in \mathcal{G}$ is a type-2 anchor *with respect to* $\mathrm{MTP}(\mathcal{G})$ can be done in quadratic time. These observations lead to the following anchor-finding procedure, which takes overall quadratic time.

$$\begin{aligned}&\textsc{FindAnchors}(\mathcal{G}) \;=\\ &\quad \{G \in \mathcal{G} \;:\; G \text{ type-1 or type-2 anchor for } \mathcal{G} \text{ wrt } \mathrm{MTP}(\mathcal{G})\}\\ &\cup \{G \in \mathcal{G} \;:\; G^t \text{ type-1 or type-2 anchor for } \mathcal{G}^t \text{ wrt } \mathrm{MTP}(\mathcal{G}^t)\}\end{aligned}$$

Because the algorithm uses $\mathrm{MTP}(\mathcal{G})$, it does not find all possible type-1 anchors. As we will see shortly, this has interesting implications for All-Termination.

## 4 All-Termination($SCP$)

In order to state the All-Termination($SCP$) problem, we first need to define a *restricted* termination analysis corresponding to $SCP$. Recall that a restricted termination analysis takes a program $F$ and a set of parameters $P$, and tries to determine if $P$ is a measurable set for $\mathcal{C}_F$. For size-change analysis, there is a fairly obvious approach: define the predicate $T(F, P)$ iff $SCP(\mathit{analyze}(F) \restriction P)$. We do not prove it here (see [1]), but $T$ is a valid termination analysis. An interesting further observation is that $T$ is *nonmonotonic*: if $P \subseteq Q$ and $T(F, P)$, it does *not* follow that $T(F, Q)$. This is in part because of the restriction that type-1 anchors use only the maximum thread preserver: it is possible for $\mathrm{MTP}(\mathcal{G} \restriction P)$ to have strict fan-in while $\mathrm{MTP}(\mathcal{G} \restriction Q)$ does not. But nonmonotonicity is a symptom of a deeper problem:

**Theorem 2** *Deciding* $\langle \exists P :: T(F, P) \rangle$ *is NP-hard.*

As a result, $T$ fails to satisfy our basic criterion: we want $SCP(\mathit{analyze}(F))$ to hold iff $\langle \exists P :: T(F, P) \rangle$ does, but $SCP$ is a polynomial-time decision procedure. We will therefore have to be more careful and creative to find an appropriate restricted termination analysis.

As it turns out, the root of the problem for type-1 anchors is the strict fan-in check, and for type-2 anchors is the no-descent set. In both cases, $SCP$ uses checks that are nonmonotonic, for efficiency sake. We can finesse the problem by performing these checks *without regard* to the restricted set of parameters, just as $SCP$ does, and only *after* the checks succeed, consider a restricted set of parameters. Luckily, both checks happen to be *reverse*-monotonic. For example, if $P \subseteq Q$ and $\mathcal{G} \restriction Q$ has strict fan-in, so does $\mathcal{G} \restriction P$. Thus, once we have found that $\mathcal{G} \restriction \mathrm{MTP}(\mathcal{G})$ has strict fan-in, we are free to consider any smaller thread-preserver, without rechecking the condition.

To clarify these issues, we have constructed a small formal system corresponding to our proposal for a restricted termination analysis. The analysis works by first executing $SCP$ normally, but recording both the anchors and SCCs in a structure we call an *anchor tree*:

| | |
|---|---|
| $\tau ::= \langle \mathcal{G}_1 \mathcal{A}_1 \tau_1,\ \ldots,\ \mathcal{G}_n \mathcal{A}_n \tau_n \rangle$ | anchor tree |
| $\mathcal{A} ::= \{G_1,\ \ldots,\ G_n\}$ | anchor set |

$$\frac{\langle \forall i :: P \vdash \tau_i \rangle \quad \langle \forall i :: \langle \forall G \in \mathcal{A}_i :: \mathcal{G}_i \vdash_P G \rangle \rangle}{P \vdash \langle \mathcal{G}_1 \mathcal{A}_1 \tau_1,\ \ldots,\ \mathcal{G}_n \mathcal{A}_n \tau_n \rangle}\ (\textsc{Tree}) \qquad \frac{\mathcal{G}^t \vdash_P G^t}{\mathcal{G} \vdash_P G}\ (\textsc{Tr})$$

$$\frac{\begin{array}{c} P \subseteq Q \quad P \in \mathrm{TP}(\mathcal{G}) \\ \mathcal{G} \restriction \mathrm{MTP}(\mathcal{G}) \text{ strict fan-in} \quad \langle \exists x \xrightarrow{\geq} y \in G \restriction P \rangle \end{array}}{\mathcal{G} \vdash_Q G}\ (\mathrm{T1}) \qquad \frac{\begin{array}{c} P \subseteq Q \quad P \in \mathrm{TP}(\mathcal{G} \rhd G) \\ P \cap \mathrm{src}(G) \neq \emptyset \end{array}}{\mathcal{G} \vdash_Q G}\ (\mathrm{T2})$$

**Fig. 1.** Formal system for $SCP_{\mathsf{R}}$.

*Anchor tree constraints* $\quad \Phi\langle \mathcal{G}_i \mathcal{A}_i \tau_i \rangle = \bigwedge_i \left( \Phi(\tau_i) \ \wedge\ \bigwedge_{G \in \mathcal{A}_i} \Phi(\mathcal{G}_i, G) \right)$

*Generic anchor constraints* $\quad \Phi(\mathcal{G}, G) = \bigvee_{i \in \{1,2\}} \left( \Psi_i(\mathcal{G}, G) \ \vee\ \Psi_i(\mathcal{G}^t, G^t) \right)$

*Thread preserver constraints* $\quad \Theta_i^H(\mathcal{G}) = \bigwedge_{G \in \mathcal{G}} \Theta_i^H(G)$

$\qquad\qquad\qquad\qquad\quad \Theta_i^H(G) = \bigwedge_{x \in \mathrm{src}(G)} \left( x_i^H \Rightarrow \left( \bigvee_{x \xrightarrow{\tau} y \in G} y_i^H \right) \right)$

*Type-specific anchor constraints*

$$\Psi_1(\mathcal{G}, G) = \begin{cases} \Psi_1'(\mathcal{G}, G) & \text{if } \mathcal{G} \restriction \mathrm{MTP}(\mathcal{G}) \text{ has strict fan-in} \\ \mathrm{false} & \text{otherwise} \end{cases}$$

$$\Psi_1'(\mathcal{G}, G) = \Theta_1^G(\mathcal{G}) \quad \wedge \quad \bigvee_{x \xrightarrow{\geq} y \in G} \left( x_1^G \wedge y_1^G \right) \quad \wedge \quad \bigwedge_{x \in \mathcal{G}} (x_1^G \Rightarrow x)$$

$$\Psi_2(\mathcal{G}, G) = \Theta_2^G(\mathcal{G} \rhd G) \quad \wedge \quad \bigvee_{x \in \mathrm{src}(G)} x_2^G \quad \wedge \quad \bigwedge_{x \in \mathcal{G}} (x_2^G \Rightarrow x)$$

**Fig. 2.** Propositional constraints for $SCP_{\mathsf{R}}$.

An anchor tree which is just a single node without children, written $\langle\rangle$, represents an execution on an ACG $\mathcal{G}$ without any nontrivial SCCs, *i.e.*, without any loops. On the other hand, if an anchor tree node does have children, the edges to its children are labeled with an SCC and a set of anchors. Thus, for example, the tree $\langle \mathcal{G}_1\{G_1, G_2\}\, \langle \mathcal{G}_1'\{G_3\}\, \langle\rangle \rangle,\ \mathcal{G}_2\{G_4\}\, \langle\rangle \rangle$ represents an execution where $\mathcal{G}_1$ and $\mathcal{G}_2$ were the initial SCCs, where $G_1$ and $G_2$ were found as anchors for $\mathcal{G}_1$ and $G_4$ was an anchor for $\mathcal{G}_2$, and where a recursion was required on the $\mathcal{G}_1$ SCC to find the anchor $G_3$.

We let *ISCP* designate an instrumented version of *SCP* that returns an anchor tree when it succeeds, and the symbol $\perp$ when it fails.

An anchor tree is a kind of certificate for polynomial size-change. Once we have an anchor tree in hand, we can analyze it to determine whether it works as a certificate even when certain formal parameters of the program are not allowed to be used. In Figure 1, we give a formal system that makes this determination. The system consists of two judgments:

| | |
|---|---|
| $\mathcal{G} \vdash_P G$ | $G$ is an anchor for $\mathcal{G}$ considering only formal parameters $P$ |
| $P \vdash \tau$ | $\tau$ is a valid certificate considering only formal parameters $P$ |

Both types of anchors require the existence of a thread preserver with certain properties. Intuitively, the formal parameters of the thread preserver may be

required to justify the anchor, since the thread preserver circumscribes the infinitely-descending threads whose existence an anchor proves. Thus if $P$ is the thread preserver used to show that $G$ is an anchor, and $\mathcal{G} \vdash_Q G$, we expect that $P \subseteq Q$. What is surprising is that this is the *only* constraint needed on $Q$, at least for that anchor. The `Tree` rule requires that, in proving $P \vdash \tau$, every anchor and subtree be justifiable within $P$.

We return to the ACG for the `insert` function, given at the beginning of Section 3, to illustrate the formal system. First, we observe that $ISCP(\mathcal{G}_{\texttt{insert}}) = \langle \mathcal{G}_{\texttt{insert}}\{G_1\}\langle\rangle\rangle$. It is not hard to see that $\mathrm{MTP}(\mathcal{G}_{\texttt{insert}}) = \{\texttt{i}, \texttt{list}, \texttt{item}\}$, and that $\mathcal{G}_{\texttt{insert}} \restriction \{\texttt{i}, \texttt{list}, \texttt{item}\} = \mathcal{G}_{\texttt{insert}}$ has strict fan-in. In addition, $\{\texttt{i}\}$ and $\{\texttt{list}\}$ are thread-preservers for $\mathcal{G}_{\texttt{insert}}$. We can apply rule T1 to make use of these thread preservers, deriving $\mathcal{G}_{\texttt{insert}} \vdash_{\{\texttt{i}\}} G_1$ and $\mathcal{G}_{\texttt{insert}} \vdash_{\{\texttt{list}\}} G_1$ respectively. Thus $G_1$ is an anchor in *two different ways*. Notably, the respective parameters sets are the minimal measurable sets for `insert`.

We can now define: $SCP_{\mathsf{R}}(F, P) \iff P \vdash ISCP(analyze(F))$.

**Theorem 3** *If $SCP_{\mathsf{R}}(F, P)$ then $P$ is a measurable set for $F$. In addition, $\langle \exists P :: SCP_{\mathsf{R}}(F, P)\rangle$ iff $SCP(analyze(F))$.*

With an appropriate restricted termination analysis in hand, we now ask: is there an efficient algorithm for All-Termination($SCP_{\mathsf{R}}$)?

It is first important to get clear on what efficiency means in this setting. Because All-Termination($SCP_{\mathsf{R}}$) is an enumeration problem, and in particular because its output is (in general) exponential in the size of its input, no polynomial-time algorithm can be given for it. However, in practice the output of the algorithm is very small (see Section 5). We therefore seek an *output-sensitive* algorithm, whose running time depends on the size of its output.

The formal system for $SCP_{\mathsf{R}}$ can be reformulated as a propositional constraint system, which we give in Figure 2; the constraints for an anchor tree $\tau$ are generated by $\Phi(\tau)$. The idea is that models of the constraint system, which will be sets of atomic propositions, correspond to sets $P$ such that $P \vdash \tau$. Atomic propositions for the constraint system come in two flavors. First, there are propositions like $x$ which represent individual formal parameters in a program. Second, there are propositions like $x_i^G$, which also represent formal parameters, but *localized* to a particular size-change graph $G$ and anchor type $i$. The connection between the constraint and formal systems is given by the following theorem.

**Theorem 4** *For all $P \subseteq \mathcal{P}$, $\tau$, we have $P \vdash \tau$ iff $\langle \exists A :: A \models \Phi(\tau) \wedge P = A \cap \mathcal{P}\rangle$.*

The constraint system works because, for any $\tau$, there are essentially[1] finitely-many possible derivations of $P \vdash \tau$. In fact, if we ignore the choices of thread-preservers in a derivation, the number of possible derivations is linear in the size of the tree: there are four possible ways to derive $\mathcal{G} \vdash G$ for each anchor $G$ of the tree. The constraint system enumerates the possible derivations, and for each derivation gives the needed constraints on the choice of $P$ to make that

[1] "Essentially" because the Tr rule can be applied in an arbitrarily-high stack. However, the rule is *involutive*: applying it twice is the same as never applying it.

derivation hold. The main subtlety is that the constraints for each anchor rule are given in terms of super- and sub-scripted propositions. We need to do this because, for example, the requirements for being a thread-preserver will change as we walk down the anchor tree. In the formal system, the rules T1 and T2 both allow the local choice of thread preserver $P$ to be *smaller* than the global set of parameters $Q$. In the constraint system, we thus have separate (localized) copies of the formal parameters so that the local constraints for one rule do not interfere with another. To globally collect the formal parameters used in a derivation, we include constraints like $x_i^G \Rightarrow x$.

We now make a key observation:

**Proposition 4** *For all $\tau$, $\Phi(\tau)$ can be written as a dual-horn formula.*

Syntactically, a dual-horn formula is a formula in conjunctive normal form, where each clause contains at most one negated variable. In other words, a dual-horn formula is a collection of constraints $a \Rightarrow (b_1 \vee \cdots \vee b_n)$. Using the constraint system, we can thus reduce All-Termination($SCP_{\mathsf{R}}$) to the problem of enumerating minimal solutions to dual-horn formulas. This is a useful observation, because there is an output-sensitive algorithm for the problem [6]. Unfortunately, the algorithm takes time *exponential* in the size of its output, and, under standard complexity assumptions, this is the best that can be done. However, as we discuss next, the size of the output in practice is bounded by an extremely small constant: 3. In this case, the reduction to dual-horn minimization means that we can solve All-Termination($SCP_{\mathsf{R}}$) just as quickly we can solve *SCP*: in time cubic in the size of the input.

## 5 Experimental results

ACL2 is an industrial-strength theorem proving system with a large regression suite with over 11,000 function definitions, each of which must be proved terminating in order to be admitted into its logic. The regression suite arises from research contributions from around the world, with examples ranging from bit-vector libraries used by AMD, to set theory libraries, graph algorithms, dag rewriting, and model checkers. In short, the regression suite provides a large, realistic sample of ACL2 programs.

We implemented an All-Termination algorithm for full size-change analysis, using *calling context graphs* (CCGs) to implement the *analyze* function [1, 10, 11]. We collected data on recursive, multiargument functions in the suite, of which there were 1,728. More than 90% had at least one termination core that did not include all the function arguments, and about 7% of the functions had more than one termination core. However, no function had more than three cores. These results show that measurable sets provide the theorem prover with nontrivial information in a vast majority of cases.

We also implemented (with Daron Vroon) the *SCP* analysis itself [10]. On the regression suite, *SCP* is on average an order of magnitude faster than full size-change, and it was able to prove terminating *every function* that full size-change did. On examples where *SCP* was significantly faster, we found that

the exponential behavior of full size-change analysis was triggered by the use of CCGs, which tend to produce ACGs with a large number of derived function parameters. These derived function parameters are crucial to the success of CCG analysis, as they enable us to find complicated relationships that are required to prove termination and that cannot be inferred directly from the function parameters [10]. There are many ways we can imagine enhancing CCG analysis (*e.g.*, by reaching into parameters to extract information relevant to termination). While space limitations do not permit a fuller description, we are certain that many of these enhancements to CCG analysis will have scalability problems due to the exponential behavior of full size change. Thus, *SCP* is clearly a more scalable *and* equally powerful (in practice), choice for All-Termination.

## 6 Conclusion

We carefully studied the All-Termination problem as applied to polynomial-time size-change analysis. By reformulating *SCP* in an appropriate way, we were able to build a boolean constraint system whose solutions are measurable sets. This allows us to *extract* as many termination cores as possible from an execution of *SCP*. In theory, the extraction step only increases the complexity of *SCP* when there are many termination cores; in practice, our experimental results show that functions with many cores are quite rare. Our primary focus for future work is analyzing the All-Termination($T$) problem for other termination analyses.

## References


1. Manolios, P., Turon, A.: All-Termination(T). In: TACAS. LNCS (March 2009)
2. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press (1979)
3. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers (July 2000)
4. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. TOPLAS **29**(1) (2007) 5
5. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, ACM Press (2001) 81–92
6. Ben-Eliyahu, R., Dechter, R.: On computing minimal models. Annals of Mathematics and Artificial Intelligence **18** (1996) 3–27
7. Turing, A.: On computable numbers, with an application to the entscheidungsproblem. In: Proceedings of the London Mathematical Society. (1936)
8. Thiemann, R., Giesl, J.: Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen (January 2003)
9. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: RTA. Volume 3091 of LNCS., Springer (2004) 210–220
10. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: CAV. Volume 4144 of LNCS., Springer (2006) 401–414
11. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: The ACL2 Sedan. ENTCS **174**(2) (2007) 3–18
12. Krauss, A.: Certified size-change termination. In Pfenning, F., ed.: CADE. Volume 4603 of LNCS., Springer (2007) 460–475