All-Termination(T)*

Panagiotis Manolios and Aaron Turon

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

Abstract. We introduce the All-Termination(T) problem: given a termination solver T and a collection of functions F, find every subset of the formal parameters to F whose consideration is sufficient to show, using T, that F terminates. An important and motivating application is enhancing theorem proving systems by constructing the set of strongest induction schemes for F, modulo T. These schemes can be derived from the set of *termination cores*, the minimal sets returned by All-Termination(T), without any reference to an explicit measure function. We study the All-Termination(T) problem as applied to the size-change termination analysis (SCT), a PSpace-complete problem that underlies many termination solvers. Surprisingly, we show that All-Termination(SCT) is also PSpace-complete, even though it substantially generalizes SCT. We develop a practical algorithm for All-Termination(SCT), and show experimentally that on the ACL2 regression suite (whose size is over 100MB) our algorithm generates stronger induction schemes on 90% of multiargument functions.

1 Introduction

Reasoning about recursion requires induction. But there may be several induction schemes that apply to a given recursive function, and different theorems may require the use of different induction schemes. Finding induction schemes for a given function is particularly important for automated theorem provers that perform induction heuristically. In this context, Boyer and Moore explored the strong relationship between termination and both recursion and induction [1]. They showed that proving termination is the key to justifying function definitions and induction schemes, and developed methods for doing so mechanically. This was one of the major insights that led to the success of the Boyer-Moore family of theorem provers, which includes ACL2 [2].

In this paper, we introduce a generalization of the classic termination problem: All-Termination. The motivating application for this problem is its use in mechanically deriving and justifying as many induction schemes for a function as possible, using methods like Boyer and Moore's. Each induction scheme is closely tied to the pattern of recursion in the function, so the schemes are likely to be useful in automated reasoning about the function.

* This research was funded in part by NASA Cooperative Agreement NNX08AE37A and NSF grants CCF-0429924, IIS-0417413, and CCF-0438871.

S. Kowalewski and A. Philippou (Eds.): TACAS 2009, LNCS 5505, pp. 398–412, 2009.

We begin with a few examples, first using traditional induction schemes, and then describing the schemes we can derive through ALL-TERMINATION analysis. Consider the function `even`, which determines whether a natural number is even:

```
even n = if n = 0 then T else if n = 1 then F else even (n - 2)
```

To show the correctness of `even`, there are a few induction principles we might apply. An obvious first choice is standard induction over the naturals. However, this principle does not suffice for proving the theorem, because it is not possible to prove that `even`(n) is correct assuming only that `even`$(n-1)$ is correct; we need instead that `even`$(n-2)$ is correct. On the other hand, we could employ strong induction on the naturals. We would then have to show

$$\langle \forall m < n :: \texttt{even}(m) \text{ iff } \lfloor m/2 \rfloor = m/2 \rangle \implies (\texttt{even}(n) \text{ iff } \lfloor n/2 \rfloor = n/2)$$

where n is implicitly universally quantified, a shorthand we will use throughout this section. While this is a reasonable choice of induction scheme, it is a bit *ad hoc*. In particular, it is hard to see how to derive such an induction scheme from the definition of `even` in an automated way.

What Boyer and Moore propose instead is to derive an induction scheme from the pattern of *recursion* in the function body. For the `even` function, we can derive following induction scheme for proving $\langle \forall n :: \varphi(n) \rangle$:

$$\varphi(0), \qquad \varphi(1), \qquad n \neq 0, n \neq 1, \varphi(n-2) \implies \varphi(n)$$

How do we know that the induction scheme is sound? By proving that `even` *terminates* on all inputs. Since the induction scheme corresponds directly to the recursion of `even`, knowing that the recursion terminates allows us to apply well-founded induction and soundly derive the induction scheme above. Notice that the induction scheme can be applied to *any* φ, not just φ involving `even`. But the derivation of the scheme was mechanically guided by the definition of `even`.

Now we turn to a more interesting example: the function `zip`, which takes a pair of lists and produces a list of pairs:

```
zip xs ys = if nil?(xs) or nil?(ys) then nil
            else cons (head x, head y) (zip (tail xs) (tail ys))
```

Recall that a *measure* μ on a function f is another function, on the same domain, that maps into a well-ordered structure[1] such that whenever $f(a)$ calls $f(b)$, we have $\mu(a) > \mu(b)$. Because the successive values of μ cannot decrease infinitely, f cannot recur infinitely. We say a set P of formal parameter names for a function is *measurable* if there exists a measure on that function that uses only those arguments. Suppose `length` measures the length of a list. For `zip`, the sets {`xs`, `ys`}, {`xs`}, and {`ys`} are measurable, because `length(xs) + length(ys)`, `length(xs)`, and `length(ys)`, respectively, are measures. Ignoring the measures themselves, we can use measurable sets of a function to derive induction schemes. For instance, here are two induction schemes for `zip`:

[1] A set with a total order that has no infinite descending chains $x_1 > x_2 > \cdots$.

Measurable set: $P = \{\texttt{xs}, \texttt{ys}\}$	Measurable set: $P = \{\texttt{xs}\}$
1. $\varphi(\texttt{[]}, \texttt{ys})$	1. $\varphi(\texttt{[]}, \texttt{ys})$
2. $\varphi(\texttt{xs}, \texttt{[]})$	2. $\varphi(\texttt{xs}, \texttt{[]})$
3. $\varphi(\texttt{xs}, \texttt{ys}) \implies \varphi(\texttt{x:xs}, \texttt{y:ys})$	3′. $\langle \forall \texttt{zs} :: \varphi(\texttt{xs}, \texttt{zs}) \rangle \implies \varphi(\texttt{x:xs}, \texttt{y:ys})$

The intuition is that, if a parameter like ys does not appear in a measurable set, then that parameter can vary freely without affecting termination; hence, the parameter can be *instantiated* freely during induction without invalidating the induction scheme. The induction scheme based on {xs} is *stronger* than the one for {xs, ys}: any theorem proved using the latter can be proved using the former, but not *vice versa*.

In practice, Boyer-Moore theorem provers use complex heuristics to propose an induction scheme that is likely needed to prove a given theorem. The proposed scheme must then be *justified*. The key point is that, just as with the simple examples above, the justification is based on measurable sets. Users can introduce new measurable sets, but only through a manual process involving a termination proof. Our goal is to automate the process of justifying induction schemes by computing *all* the measurable sets for a given function.

We thus define ALL-TERMINATION as follows: given a recursive function f, find its measurable sets. The ALL-TERMINATION problem is a generalization of the classic termination decision problem: a program is terminating iff it has at least one measurable set. Therefore, ALL-TERMINATION is undecidable. 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 as possible using T.

In this paper, we focus on the size-change termination analysis (*SCT* [3]) because several powerful termination analyses depend on it (see Section 5 for examples). An introduction to the size-change framework is given in Section 3. Then, in Section 4, we study ALL-TERMINATION(*SCT*) in detail and show that its complexity is the *same* as the complexity of *SCT*: they are both PSPACE-complete problems. We also develop an algorithm, using *dual-horn minimization*. We have implemented this algorithm on a prototype basis, and executed it on the ACL2 regression suite, consisting of over 11,000 functions. We found that over 90% of multiargument functions have at least one measurable set that was smaller than the full set of arguments to the function, and 7% of the multi-argument functions had multiple, incomparable measurable sets. These results suggest that ALL-TERMINATION can increase automation in theorem provers.

An important practical consideration is the tension between termination analysis and ALL-TERMINATION analysis. Since termination analysis tends to be expensive, and theorem provers often require functions to be shown terminating before they can be admitted, the goal is to decide termination as quickly as possible, using the simplest analysis [4]. On the other hand, we can get better ALL-TERMINATION results by employing heavyweight methods, even when simpler methods suffice to show termination—but this involves more work. The algorithm we develop takes this tension into account and is *responsive*: it

answers the basic termination question first, without incurring any additional overhead. Only after termination is settled does it proceed with the full All-Termination(T) analysis. This approach allows a theorem prover to use spare CPU cycles or cores to detect new induction schemes in the background, after the function is determined to terminate. It also allows the theorem prover to use All-Termination(T) analysis in a demand-driven way, asking for induction schemes when the need arises.

A version of this paper with more detail and full proofs is available online [5].

2 All-Termination(T)

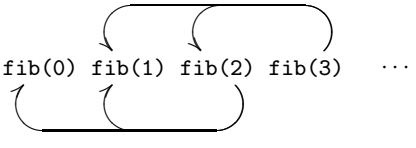
We postulate a universe of programs Prog, but do not specify a particular syntax or semantics. Intuitively, a program $F \in \text{Prog}$ is a mutually-recursive nest of functions; F terminates iff each function in F terminates on every input. Formally, we require that for every program $F \in \text{Prog}$, there is a corresponding transition system $\mathcal{C}_F$, called the *semantic call graph* of F, which terminates iff F does and whose states are function names with actual arguments. Given universes 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 $\mathcal{V}$.*

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 Fibonacci function and its semantic call graph are:

```
fib 0 = 1
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```



A semantic call graph records the actual function calls made in order to compute a given function application. In general, $\mathcal{C}_F$ is an infinite, undecidable structure: even determining whether there is a transition between two states is undecidable. We can express termination of semantic call graphs in terms of measure functions, the standard tool for proving termination, as follows.

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

This proposition follows from basic results in set theory, showing that every terminating relation can be extended to a well-order and that every well-order is order-isomorphic to a unique ordinal number.

If (f, V) is a state in a semantic call graph $\mathcal{C}$, the values of the formal arguments in $\text{dom}(V)$ are the observations available to a measure on $\mathcal{C}$. Thus, to restrict the arguments a measure can observe, and thereby force it to ignore certain arguments, we restrict 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***

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

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

Note that if $\mathcal{C}$ has any measurable set, then in particular $\mathcal{C}$ is terminating. 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 need to limit the analysis T to "use" only a certain set of formal parameters, just as for measures.

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

Finally, given a termination analysis T and a program F, the termination cores of F modulo T are the minimal P such that $T(F, P)$. That is,

$$\text{All-Termination}(T)(F) = \min\{P \subseteq \mathcal{P} \ : \ T(F, P)\}$$

3 The Size-Change Framework

The semantic call graph $\mathcal{C}_F$ precisely captures the recursive behavior of F, at the cost of undecidability. A fruitful approach to termination analysis is to consider safe approximations of $\mathcal{C}_F$. This section describes the *size-change framework* of Lee, Jones, and Ben-Amram [3]. The main result, Theorem 1, is not new. However, our presentation of the framework includes some innovations that are needed for studying All-Termination(SCT): we give a fuller account of the connection between SCT and semantic call graphs (including a notion of simulation), and we make explicit the notion of evaluation.

Example. Consider the well-known total function `ack`:

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Traditionally, to prove that `ack` terminates, a measure μ is introduced corresponding to a lexicographic order on the arguments. The size-change framework takes an alternative perspective, focusing on the change in size of each argument independently, without having to concoct a single measure on the tuple of arguments. We first observe that in every recursive call to `ack`, either the first

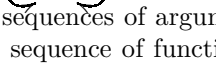
argument decreases, or the first argument does not increase while the second decreases. We display this size-change data as follows:

It follows that any putative infinite recursion would involve an infinite sequence of argument size changes of the form above—but we can show that this is not possible. If size change G_1 occurs infinitely often, then m decreases infinitely, which is impossible under a well-order. Otherwise, since we are considering an *infinite* sequence of size changes, it must be that size change G_2 occurs uninterrupted as an infinite suffix of the sequence. But then n decreases infinitely, which is again impossible. Hence, `ack` terminates. The size-change framework reformulates such reasoning into a decidable analysis.

For simplicity, we postulate a single well-ordering $>$ on all values in $\mathcal{V}$.[2] The notion of size-change "data" above is formalized into a structure called a *size-change graph*. 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 labeled by size-change graphs, which record the size relationship between the arguments of f and g. More formally, we write

$$\begin{array}{rll} p, q, r \in \textsc{Lab} & = \{>, \geq\} & \text{size-change label} \\ G, H \in \textsc{SCG} & = 2^{\mathcal{P} \times \textsc{Lab} \times \mathcal{P}} & \text{size-change graph} \\ \mathcal{G}, \mathcal{H} \in \textsc{ACG} & = 2^{\mathcal{F} \times \textsc{SCG} \times \mathcal{F}} & \text{annotated call graph} \end{array}$$

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.

The annotated call graph for `ack` is: 

. The intuitive demonstration that `ack` terminates was based on sequences of argument size changes during recursive function calls. A potential sequence of function calls is just a path through an ACG.

Definition 6. *A **multipath** π 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 7. *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 $i > 0$.*

[2] Multiple orders can also be handled [4].

For example, consider the multipath $\texttt{ack} \xrightarrow{G_1} \texttt{ack} \xrightarrow{G_2} \texttt{ack} \xrightarrow{G_1} \texttt{ack}$ in $\mathcal{G}_{\texttt{ack}}$. Its only thread is $m \xrightarrow{>} m \xrightarrow{\geq} m \xrightarrow{>} m$. On the other hand, the multipath $\texttt{ack} \xrightarrow{G_2} \texttt{ack} \xrightarrow{G_2} \texttt{ack}$ has two threads: $m \xrightarrow{\geq} m \xrightarrow{\geq} m$ and $n \xrightarrow{>} n \xrightarrow{>} n$.

Threads track 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. Size-change termination analysis works by considering all potential infinite multipaths through an ACG, and demonstrating that each of them contains an infinite thread that forces its value to decrease infinitely often. By well-foundedness, such infinite decreases cannot occur, and so all infinite multipaths are ruled out.

Definition 8

(1) An infinite thread $\langle x_{i-1} \xrightarrow{r_i} x_i \rangle$ *has* ***infinite descent*** *if* $r_i = >$ *for infinitely-many* i.
(2) A multipath π *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.*

We have motivated ACGs in terms of function calls, but it remains to formally connect ACGs to semantic call graphs. An ACG $\mathcal{G}$ can be seen as a finite description of a semantic call graph $\mathcal{C}_\mathcal{G}$ that relates states according to the possible size changes given in $\mathcal{G}$:

Definition 9. *The* ***semantic call graph determined by*** $\mathcal{G}$ *is* $\mathcal{C}_\mathcal{G} = (S, \rightarrow)$, *where* $S = \mathcal{F} \times (\mathcal{P} \rightharpoonup \mathcal{V})$ *and*

$$(f, V) \rightarrow (g, U) \quad \textit{iff} \quad \langle \exists f \xrightarrow{G} g \in \mathcal{G} :: \langle \forall x \xrightarrow{r} y \in G :: V(x)\ r\ U(y) \rangle \rangle$$

In order to use the size-change termination of $\mathcal{G}$ to show the termination of F, we must relate $\mathcal{C}_\mathcal{G}$ and $\mathcal{C}_F$. The relation we use is a form of *simulation*.

Definition 10. *Given two semantic call graphs* $\mathcal{C}_1 = (S_1, \rightarrow_1)$ *and* $\mathcal{C}_2 = (S_2, \rightarrow_2)$, *a* ***simulation*** *between* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *is a relation* $R \subseteq S_1 \times S_2$ *such that*

- *for each* s_1 *there is some* s_2 *with* $s_1\ R\ s_2$
- *if* $s_1\ R\ s_2$ *then* $s_1 = (f, V)$ *and* $s_2 = (f, U)$ *with* $f \in \mathcal{F}$ *and* $U = V \restriction dom(U)$
- *if* $s_1\ R\ s_2$ *and* $s_1 \rightarrow_1 s_1'$ *then there exists an* s_2' *such that* $s_2 \rightarrow_2 s_2'$ *and* $s_1'\ R\ s_2'$

We say $\mathcal{C}'$ **simulates** $\mathcal{C}$, written $\mathcal{C} \sqsubseteq \mathcal{C}'$, if there exists a simulation R between $\mathcal{C}$ and $\mathcal{C}'$. Intuitively, if $\mathcal{C}'$ simulates $\mathcal{C}$, then $\mathcal{C}'$ admits at least as many behaviors as $\mathcal{C}$. We say $\mathcal{G}$ is **safe for** F if $\mathcal{C}_F \sqsubseteq \mathcal{C}_\mathcal{G}$. In general, finding a safe ACG $\mathcal{G}$ for a program F is difficult, and is a problem that the size-change framework does not address (but see [4]). For our purposes, it is sufficient to postulate a function *analyze* : PROG $\rightarrow$ ACG such that *analyze*(F) is safe for F.

The next two propositions allow us to conclude that if $\mathcal{G}$ is safe for F and $\mathcal{G}$ is size-change terminating then F terminates.

Proposition 2. *$\mathcal{G}$ is size-change terminating iff $\mathcal{C}_\mathcal{G}$ is terminating.*

Proposition 3. *If $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C}'$ is terminating then $\mathcal{C}$ is terminating.*

Deciding size-change termination for an ACG $\mathcal{G}$ is a PSPACE-complete problem, but the standard algorithm used in practice needs exponential space in the worst case [3]; we present this algorithm next.

Suppose $\mathcal{G}$ is an ACG. If $f_0 \xrightarrow{G_1} \cdots \xrightarrow{G_n} f_n$ is a multipath in $\mathcal{G}^+$, we know that according to $\mathcal{G}$, a call to f_0 could result in a call to f_n. But what can we say about the size of the arguments to f_n in terms of the arguments to f_0? What we want is a way to compose size-change graphs along a multipath.

Definition 11. *We define composition of size-change labels and graphs by*

$$p \cdot q = \begin{cases} \geq & p = \geq \text{ and } q = \geq \\ > & \text{otherwise.} \end{cases} \qquad G \cdot H = \{x \xrightarrow{p \cdot q} z \ : \ x \xrightarrow{p} y \in G, y \xrightarrow{q} z \in H\}$$

Definition 12. *The **evaluation** of $\pi = \langle G_1, \ldots, G_n \rangle \in \mathcal{G}^+$ is $[\![\pi]\!] = G_1 \cdot \ \cdots \ \cdot G_n$.*

Note that composition is associative, so evaluation is well-defined. The evaluation of a multipath π is useful because it compactly summarizes the threads in π:

Proposition 4. *$x \xrightarrow{r} y \in [\![\pi]\!]$ iff there exists a thread $\langle x \xrightarrow{r_1} z_1 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} z_{n-1} \xrightarrow{r_n} y \rangle$ in π, with $r = r_1 \cdot \ \cdots \ \cdot r_n$.*

The key step for the size-change termination algorithm is to compute the *closure* of an annotated call graph $\mathcal{G}$ under composition: the set $\{[\![\pi]\!] \mid \pi \in \mathcal{G}^+\}$. The closure is formally defined as a least fixpoint. The algorithm looks for certain "maximal" size-change graphs in the closure, called *idempotents*.

Definition 13.

*(1) The **closure** of $\mathcal{G}$ under $\cdot$ is the smallest set satisfying*

$$cl(\mathcal{G}) = \mathcal{G} \cup \{f \xrightarrow{G \cdot H} h \ : \ f \xrightarrow{G} g, g \xrightarrow{H} h \in cl(\mathcal{G})\}$$

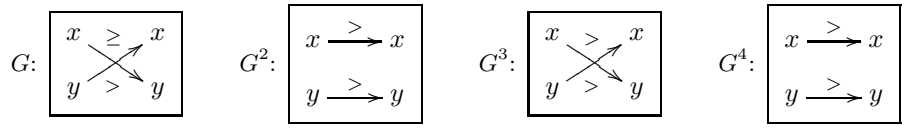
*(2) A size-change graph G is **idempotent** if $G \cdot G = G$.*

Theorem 1 (Lee *et al.* [3]). *$\mathcal{G}$ is size-change terminating iff for every $f \xrightarrow{G} g \in cl(\mathcal{G})$ such that G is idempotent, there is an edge $x \xrightarrow{>} x \in G$.*

Example. Consider the following function `perm`, which permutes its two arguments, decreasing one of them, until one of them is zero.

```
perm 0 y = y
perm x 0 = x
perm x y = perm (y - 1) x
```

How can we use the theorem above to show that `perm` terminates? First, we need to construct $\mathcal{G}_{\texttt{perm}}$. Since there is only one recursive call in `perm`, $\mathcal{G}_{\texttt{perm}}$ has only one node and one edge. The size-change graph G for `perm` and its powers are:



Note that G^2 is idempotent, so $G^4 = G^2$. Consequently, the only distinct size-change graphs in $cl(\mathcal{G}_{\texttt{perm}})$ are G, G^2, and G^3. Since G^2 has an edge $x \xrightarrow{>} x$, and G^2 is the only idempotent graph in $cl(\mathcal{G}_{\texttt{perm}})$, $\mathcal{G}_{\texttt{perm}}$ is size-change terminating.

The standard algorithm for deciding size-change termination is based on Theorem 1: compute $cl(\mathit{analyze}(F))$ as a least fixpoint, and check the strict self-edge condition on the idempotent elements. To adapt this algorithm for all-termination, we will record some additional information as size-change graphs are composed, and build a constraint system from this information after the algorithm finishes. The minimal solutions to these constraints will be exactly the termination cores of F.

4 All-Termination(SCT)

Recall that a termination analysis is a predicate $T(F, P)$ that holds only if P is a measurable set for F. Thus, the first step in studying All-Termination(SCT) is to formulate such a predicate $SCT(F, P)$, with the property that $\langle \exists P :: SCT(F, P) \rangle$ holds exactly when F is size-change terminating. Size-change analysis, like many termination analyses, does not explicitly construct a measure witnessing termination; it only implies the existence of one.[3] We need a way to restrict size-change analysis to a set of parameters P, such that this implied measure only uses parameters from P. To do this, we derive an ACG $\mathit{analyze}(F) \restriction P$ whose size-change termination implies that P is a measurable set of F. This restricted ACG simply drops any size-change information not related to parameters in P.

Definition 14. *Given G, $\mathcal{G}$, and P, we define the* ***restrictions***

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

Similarly, we introduce a notion of restriction on semantic calls graphs, which will allow us to derive a useful new characterization of measurable sets.

Definition 15. *Given $\mathcal{C} = (S, \rightarrow)$ and P, we define the* ***restriction***

$$\mathcal{C} \restriction P = \big(\{(f, V \restriction P) \ : \ (f, V) \in S\}, \rightsquigarrow\big)$$

where $s \rightsquigarrow t$ iff there exist $s', t' \in \mathcal{C}$ such that $s = s' \restriction P$, $t = t' \restriction P$, and $s' \rightarrow t'$.

[3] It is possible to effectively construct a measure from size-change analysis, and thereby extract a *single* measurable set, but the size of the measure is exponential [6].

Proposition 5. *For all $\mathcal{C}$, $\mathcal{G}$ and $P \subseteq \mathcal{P}$ we have*

(1) P is a measurable set for $\mathcal{C}$ iff $\mathcal{C} \restriction P$ is terminating,
(2) $\mathcal{C} \sqsubseteq \mathcal{C} \restriction P$,
(3) if $\mathcal{C} \sqsubseteq \mathcal{C}'$ then $\mathcal{C} \restriction P \sqsubseteq \mathcal{C}' \restriction P$, and
(4) $\mathcal{C}_{\mathcal{G}} \restriction P \approx \mathcal{C}_{\mathcal{G}\restriction P}$, where $(\approx) = (\sqsubseteq) \cap (\sqsupseteq)$.

We now have all the pieces needed to define the SCT predicate:

$$SCT(F, P) \iff \mathit{analyze}(F) \restriction P \text{ is size-change terminating}$$

Theorem 2. *SCT is a termination analysis.*

Proof. Using Proposition 5, $\mathcal{C}_F \restriction P \sqsubseteq \mathcal{C}_{\mathit{analyze}(F)} \restriction P \approx \mathcal{C}_{\mathit{analyze}(F)\restriction P}$. If $SCT(F, P)$ holds then $\mathit{analyze}(F) \restriction P$ is size-change terminating, so $\mathcal{C}_{\mathit{analyze}(F)\restriction P}$ is terminating, and hence so is $\mathcal{C}_F \restriction P$. By Proposition 5, P is a measurable set of F.

Deciding $SCT(F, P)$ is PSPACE-complete, since the restriction of an ACG to P can be computed in polynomial time, after which the problem reduces to size-change termination. Somewhat surprisingly, ALL-TERMINATION(SCT) has the *same* complexity.

Theorem 3. ALL-TERMINATION(SCT) *is* PSPACE*-complete.*

Proof. ALL-TERMINATION(SCT) is PSPACE-hard because $\langle \exists P :: SCT(F, P) \rangle$ can be reduced to ALL-TERMINATION(SCT)(F) in constant space by executing ALL-TERMINATION(SCT)(F) until it either halts with no output or produces its first output, and $\langle \exists P :: SCT(F, P) \rangle$ is PSPACE-hard. On the other hand, the following algorithm solves ALL-TERMINATION(SCT) in polynomial space:

```
ALL-TERMINATION(SCT)(F)
    for P ⊆ 𝒫 do
        if SCT(F, P) and ⟨∀Q ⊂ P :: SCT(F, Q) = False⟩ then output P
```

The algorithm uses polynomial space because $SCT(F, P)$ is in PSPACE, and all of the loops can be implemented using counters whose size is logarithmic in the size of $2^{\mathcal{P}}$, hence linear in the size of $\mathcal{P}$, where $\mathcal{P}$ is the set of parameters to functions in F.

This theorem generalizes to any PSPACE-complete termination analysis. Having settled the basic complexity question, we now turn to practical considerations. The algorithm above executes SCT at least $2^{|\mathcal{P}|}$ times. The algorithm we introduce below runs SCT once, gathering information from which it extracts the termination cores. The key to the algorithm is understanding the threads and multipaths in $(\mathcal{G} \restriction P)^+$ in terms of those in $\mathcal{G}^+$. We first observe that each multipath in $(\mathcal{G} \restriction P)^+$ is the *restriction* of a multipath in $\mathcal{G}^+$, as follows.

Definition 16. *Given a multipath $\pi = f_0 \xrightarrow{G_1} \cdots \xrightarrow{G_n} f_n$ in $\mathcal{G}^+$, the **restriction** of π to $P \subseteq \mathcal{P}$ is $\pi \restriction P = f_0 \xrightarrow{G_1\restriction P} \cdots \xrightarrow{G_n\restriction P} f_n$, which is a multipath in $(\mathcal{G} \restriction P)^+$.*

Proposition 6. *Let $P \subseteq \mathcal{P}$.*

(1) If $\pi \in (\mathcal{G} \restriction P)^+$ then there exists a $\pi' \in \mathcal{G}^+$ such that $\pi = \pi' \restriction P$.
(2) If $\pi \in \mathcal{G}^+$ then the threads of $\pi \restriction P$ are exactly the threads $\langle x_0 \xrightarrow{r_1} \cdots \xrightarrow{r_n} x_n \rangle$ of π such that each x_i is in P.

Notice that as size-change graphs are composed, some information about the possible threads within them is lost. For example, if $x \xrightarrow{\geq} z \in G \cdot H$, we know that there is *some* y for which $x \xrightarrow{p} y \in G$ and $y \xrightarrow{q} z \in H$ with $pq = \geq$, but given only the composed graph $G \cdot H$ it is not possible to determine which choices of y would suffice. More generally, given a multipath $\pi \in \mathcal{G}^+$, we can determine all of its threads; but, given only $x \xrightarrow{r} y \in [\![\pi]\!]$, the most we can say is that there is *some* thread in π from x to y (by Proposition 6). Thus, if we want to reason about the threads of $\pi \restriction P$ (and hence the edges in $[\![\pi \restriction P]\!]$) in terms of $[\![\pi]\!]$, we need to keep track of which variables contribute to each edge $x \xrightarrow{r} y \in [\![\pi]\!]$. We do this using *annotated size-change graphs*:

$$\mathbb{G}, \mathbb{H} \in \mathrm{ASCG} = 2^{\mathcal{P} \times (\mathrm{LAB} \times 2^{\mathcal{P}}) \times \mathcal{P}} \quad \text{annotated size-change graphs}$$

Intuitively, if an edge $x \xrightarrow[Q]{r} y$ is in an ASCG $\mathbb{G}$, then there is some thread relating x to y with size-change r, involving at most the parameters in Q.

If G is a size-change graph in $\mathcal{G}$, and $x \xrightarrow{r} y \in G$, the only parameters needed to show that there is a thread from x to y are x and y themselves. Thus we have a simple way of producing initial ASCGs from SCGs:

Definition 17. *The ASCG for G is $\lfloor G \rfloor = \{x \xrightarrow[\{x,y\}]{r} y \ : \ x \xrightarrow{r} y \in G\}$.*

Just as with SCGs, we have composition, evaluation, and closure for ASCGs.

Definition 18

(1) Annotated composition and evaluation are defined as follows:

$$\mathbb{G} \odot \mathbb{H} = \{x \xrightarrow[P \cup Q]{pq} z \ : \ x \xrightarrow[P]{p} y \in \mathbb{G}, \ y \xrightarrow[Q]{q} z \in \mathbb{H}\}$$
$$\lfloor G_1, \ldots, G_n \rfloor = \lfloor G_1 \rfloor \odot \cdots \odot \lfloor G_n \rfloor$$

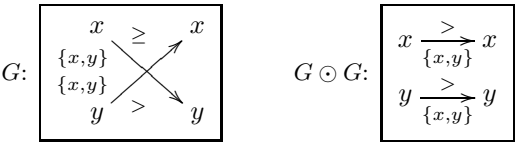
*(2) The **annotated closure** of $\mathcal{G}$ under $\odot$ is the least set satisfying*

$$acl(\mathcal{G}) = \{f \xrightarrow{\lfloor G \rfloor} g \ : \ f \xrightarrow{G} g \in \mathcal{G}\} \cup \{f \xrightarrow{\mathbb{G} \odot \mathbb{H}} h \ : \ f \xrightarrow{\mathbb{G}} g, g \xrightarrow{\mathbb{H}} h \in acl(\mathcal{G})\}$$

We can now reason about the multipaths of $(\mathcal{G} \restriction P)^+$ in terms of $\mathcal{G}^+$:

Proposition 7. *Let $\pi \in \mathcal{G}^+$. Then $x \xrightarrow{r} y \in [\![\pi \restriction P]\!]$ iff there exists a $Q \subseteq P$ such that $x \xrightarrow[Q]{r} y \in \lfloor \pi \rfloor$.*

Example. Returning to the `perm` example, we ask: is {`x`} a measurable set for `perm`? No: a function taking only the `x` parameter for `perm` cannot possibly be a measure. To see why, consider that `perm 1 2` calls `perm 1 1`. Thus, a measure μ for `perm` using only `x` would have to have the property that $\mu(1) > \mu(1)$ which is clearly impossible. A similar argument shows that {`y`} is not a measurable set. We can now reanalyze the `perm` function in using annotated size-change graphs, to see how they are used to discover that {`x`} and {`y`} are not measurable sets. We begin with the same graph G we had before, but with annotated edges.



As before, $G \odot G$ is idempotent. The annotations on the edges of $G \odot G$, however, tell us that to justify a decrease from, *e.g.*, x to x in $G \odot G$, we *must* consider the formal argument y as well.

The next result shows that we have completely characterized size-change termination for any $\mathcal{G} \upharpoonright P$ in terms of the annotated closure $acl(\mathcal{G})$.

Theorem 4. *$\mathcal{G} \upharpoonright P$ is size-change terminating iff for every $f \xrightarrow{\mathbb{G}} g \in acl(\mathcal{G})$ such that $\mathbb{G}$ is idempotent, there is an edge $x \underset{Q}{\overset{\geq}{\longrightarrow}} x \in \mathbb{G}$ with $Q \subseteq P$.*

Corollary 1. *Let $\mathcal{I} = \{\mathbb{G} \in acl(analyze(F)) : \mathbb{G}$ idempotent$\}$. We have* All-Termination$(SCT)(F) = \{P \subseteq \mathcal{P} : \langle \forall \mathbb{G} \in \mathcal{I} :: \langle \exists x \underset{Q}{\overset{\geq}{\longrightarrow}} x \in \mathbb{G} :: Q \subseteq P \rangle \rangle\}$.

We can use Corollary 1 as the basis for an algorithm as follows. First, compute $acl(analyze(F))$ as a least fixpoint, and extract the set of idempotent ASCGs as $\mathcal{I}$. Then, for each $\mathbb{G} \in \mathcal{I}$, construct the constraint $\bigvee_{x \underset{Q}{\overset{\geq}{\longrightarrow}} x \in \mathbb{G}} \bigwedge_{y \in Q} y$. The collection (conjunction) of these constraints is a constraint system Φ_F whose solutions are the elements of All-Termination(SCT). It turns out that by introducing variables, this constraint system can be expressed in *dual-horn* form.

This is a useful observation because there is an *output-sensitive* algorithm for enumerating the minimal solutions to dual-horn formulas [7]. Output sensitivity means that the running time of the algorithm is bounded by the number of *outputs* it produces, and more-over provides "pay-as-you-go" enumeration. For dual-horn minimization, the time is *exponential* in the number of outputs; moreover, the constraint system Φ_F may have an exponential number of minimal solutions. In practice, however, functions have very few termination cores (no more than 3 in our experiment), whereas they often have many arguments (sometimes more than 20 in our experiment). Hence we much prefer the annotation-based algorithm (exponential in the former) than the naive algorithm (exponential in the latter). Finally, we have developed a version of dual-horn minimization based on incremental SAT-solving, which we hope will perform well when faced with a larger number of cores.

The algorithm described above has another appealing property: it can be made *responsive*, by which we mean it can answer the basic termination problem as quickly as the standard size-change algorithm. If the program cannot be shown to terminate, there is no need to continue. If termination is established, then responsiveness can be exploited by the theorem prover in various ways, including the following two. First, the theorem prover can run the ALL-TERMINATION(T) algorithm to completion. For interactive theorem proving applications, this can be done using spare CPU cycles (*e.g.*, by using an underutilized CPU core), because the user is free to continue as soon as termination has been established, and any new induction schemes found can be quietly recorded by the theorem prover. Second, the theorem prover can suspend the ALL-TERMINATION(T) algorithm, coming back to it only when it needs new induction schemes, thereby using the analysis in a demand-driven way. Responsiveness is obtained by controlling the least fixpoint computation of $acl(analyze(F))$ so that the size change graphs needed to compute $cl(analyze(F))$ are generated first. This process differs from the basic size-change algorithm only in that we record the size-change graph annotations required for the annotated closure. Once termination is established, the fixpoint computation for the annotated closure proceeds.

Experimental Results

We have implemented our ALL-TERMINATION(SCT) algorithm in ACL2, an industrial-strength theorem proving system. Our implementation served as a new back-end for the *calling context graph* (CCG) termination analysis, which is implemented in the ACL2 Sedan [4,8]. Normally, CCG analysis uses SCT as a back-end; by using ALL-TERMINATION(SCT) instead, we are able to determine the measurable sets for a function. ACL2 has a large regression suite, with over 11,000 function definitions, each of which must be proved terminating in order to be admitted into the logic. The regression suite is particularly appealing because it arises from the work of researchers around the world, with examples ranging from bit-vector libraries used by AMD, to set theory libraries, graph algorithms and model checkers. In short, the code in the regression suite provides a large, realistic sample of ACL2 programs.

We executed our analysis on the entire regression suite The time running ALL-TERMINATION(SCT) was negligible compared to the time spent within CCG's static analysis, which involves theorem proving. We collected data on the 1,728 recursive, multiargument functions in the suite. More than 90% of the functions had at least one termination core that did not include all the arguments to the function, and about 7% of the functions had more than one termination core. These findings attest to the utility of ALL-TERMINATION: the 90% of functions with nontrivial termination cores can be given a stronger induction scheme, using ALL-TERMINATION, than would otherwise be possible. Thus, by generating stronger induction schemes, our analysis has the potential to extend the automation provided by theorem provers [1,2].

5 Related Work

The termination problem dates back to Turing, who called it the "Printing Problem" [9], and there has been steady interest in termination ever since. Here we can only briefly touch upon the work most directly related to ours.

Boyer and Moore's work [1], developing the strong relationship between termination and both recursion and induction the context of automated theorem proving, provided the impetus for the work we have presented. The idea of All-Termination, too, can be traced back to Boyer and Moore [1]. 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. We know of no other work studying All-Termination.

Termination analysis is currently an active area of research. There is much interest in termination in the context of term-rewrite systems and logic programs [10,11,12,13]. There is also interest in proving termination of programs written in imperative languages, such as C. This work tends to focus on semi-algebraic functions, whose termination behavior is governed by integer arithmetic. Most of it has been even more narrowly defined, dealing only with systems whose behavior is linear [14], though there are extensions to programs with polynomial behavior [15]. Also, abstraction-refinement has been applied to termination analysis, and subsequently used to find bugs in device drivers [16].

This paper has focused on size-change termination analysis [3], 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 [12]. Tools based on these ideas include AProVE [11]. Another example is work on calling context graphs and measures, which is used to prove termination of functional programs [4], and has been implemented in ACL2s [8] and Isabelle [17].

Recently, the problem of conditional termination has been studied [18]. While we are interested in how we can add behaviors to programs while maintaining termination, conditional termination asks how to remove behaviors to ensure termination. This leads to the obvious question: what about All-Conditional-Termination(T)? Similarly, the non-termination problem [19] gives rise to the All-Non-Termination(T) problem.

6 Conclusions and Future Work

We introduced the All-Termination(T) problem and analyzed the complexity when T is size-change (SCT) analysis. We showed that All-Termination(SCT) is a PSpace-complete problem, and introduced an algorithm for solving it. The algorithm imposes no overhead on solving the basic termination problem, and it can be used to generate a subset of all possible termination cores (up to a user provided bound). We implemented our algorithm in ACL2 and ran it on the ACL2 regression suite, consisting of over 100MB of code developed over several decades by a

worldwide user base. Our experiments showed that on over 90% of multiargument functions in the regression suite, we were able to provide stronger induction schemes than those obtained with size change analysis. Our primary focus for future work is analyzing the ALL-TERMINATION(T) problem for other termination analyses.

Acknowledgements. Alec Heller gave helpful feedback on several drafts.

References

1. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, London (1979)
2. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Dordrecht (2000)
3. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92. ACM Press, New York (2001)
4. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 401–414. Springer, Heidelberg (2006)
5. Manolios, P., Turon, A.: All-Termination(T). Technical Report NU-CCIS-09-01, Northeastern University (2009)
6. Ben-amram, A.M., Lee, C.S.: Ranking functions for size-change termination II. In: RDP-WST (2007)
7. Ben-Eliyahu, R., Dechter, R.: On computing minimal models. Annals of Mathematics and Artificial Intelligence 18, 3–27 (1996)
8. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: The ACL2 Sedan. ENTCS 174(2), 3–18 (2007)
9. Turing, A.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society 42(2), 230–265 (1936)
10. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 210–220. Springer, Heidelberg (2004)
12. Thiemann, R., Giesl, J.: Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen (January 2003)
13. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. The Journal of Logic Programming 41(1), 103–123 (1999)
14. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004)
15. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)
16. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426. ACM Press, New York (2006)
17. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
18. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
19. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL, pp. 147–158. ACM, New York (2008)