All-Termination(T)

Panagiotis Manolios
Northeastern University
pete@ccs.neu.edu

Aaron Turon
Northeastern University
turon@ccs.neu.edu

September 6, 2008

Abstract

We introduce the All-Termination(T) problem: given a termination solver, T, and a program (a set of functions), find every set of formal arguments whose consideration is sufficient to show the program terminating using T. One important and motivating application is enhancing theorem proving systems by constructing the set of strongest induction schemes for a program, modulo T. These schemes can be derived from the set of *termination cores*, the minimal sets returned by All-Termination(T). We study the All-Termination(T) problem as applied to two existing termination analyses: general size-change (SCT) and polynomial size-change (SCP). We analyze the intrinsic complexity of the problems and develop algorithms that we expect to perform well in practice. We show that All-Termination(SCT) is a PSpace-complete problem. We also show that no output-polynomial algorithm exists for All-Termination(SCP) (unless P = NP). Finally, we develop practical algorithms, some based on SAT-solving, for both problems.

1 Introduction

Termination analysis has a well-established motivation. It is required for establishing total correctness of transformational systems and for proving liveness properties of reactive systems. It is used in deductive verification to simplify mechanical reasoning and to show that adding axioms to a logic does not render it inconsistent. Termination plays a key role in the analysis of hybrid systems and in control theory, where questions of stability reduce to termination analysis. It is important in planning, logic programming, and the theory of rewrite systems, to mention a few application areas.

In this paper, we initiate the study of the All-Termination problem: given a program, find every "reason" that the program terminates. There are many possible definitions of what constitutes a reason for termination, but in this paper, a reason is a set of formal arguments whose consideration is sufficient to show the program terminating. Our focus is on developing the All-Termination framework and answering basic complexity and algorithmic questions. Implementation considerations will have to be addressed in future work.

Why is solving the All-Termination problem useful? Consider the use of termination analysis in a theorem proving system such as ACL2 [14]. In order to maintain soundness, ACL2 requires that function definitions are shown terminating, but the termination proofs have another, very important, role. They give rise to induction schemes, and different termination proofs for the same function may give rise to different induction schemes.

Subsequent proofs involving an admitted function may require several, distinct induction schemes [4]. Currently, only one induction scheme is introduced per function definition, so users are forced to manually add other induction schemes (a process that requires a termination proof). A solution to the All-Termination problem enables us to automate this process. There are other applications as well. For example, in control theory, there are many conflicting design considerations and designers often want to explore the design space of their control laws, under the constraint that stability is not violated. Determining what these constraints are amounts to solving the All-Termination problem.

The All-Termination problem is a generalization of the classic termination decision problem: a program is terminating iff there is at least one reason for termination. Therefore, the All-Termination problem is undecidable. However, decades of work on termination has yielded powerful, but decidable, termination analyses. For any such termination analysis, T, we can solve the All-Termination(T) problem: given a program and a termination solver, T, find every set of formal arguments sufficient to show the program terminating using T. A formalization can be found in the next section.

We consider the complexity of the All-Termination(T) problem and develop algorithms for the case where T is size-change analysis [17] and polynomial size-change analysis [2]. We focus on the size change framework, presented in Section 3, because several powerful termination analyses depend on it. This includes work on termination in term-rewrite systems that combines size-change analysis with the dependency pair method and recursive path orderings [20]. Tools based on these ideas include AProVE [12]. Another example is work on calling context graphs and measures, which is used to prove termination of functional programs [18], and has been implemented in ACL2s [10] and Isabelle [15].

An introduction to the size-change framework is given in Section 3. In Section 4, we consider the general size-change problem (SCT) and show that the complexity of All-Termination(SCT) is the *same* as the complexity of SCT: they are both PSpace-complete problems. In Section 5, we consider polynomial size-change analysis (SCP), whose complexity is $O(n^3)$, and present some interesting complexity results. First, notice that the number of reasons for termination can be exponential in the number of formals, even if we restrict our attention to termination cores (minimal sets of formals). So, at best we might hope for an output-polynomial algorithm, *i.e.*, an algorithm whose running time is polynomial in the size of its output. Unfortunately, we show that, under standard complexity assumptions, no output-polynomial algorithm exists for All-Termination(SCP). One of the technical tools we make essential use of is dual-horn minimization, and we present algorithms for solving this problem in Section 6. One of the algorithms we develop is based on SAT-solving, and thus can make use of modern, incremental SAT-solvers.

We also develop algorithms for All-Termination(SCT) and All-Termination(SCP), using dual-horn minimization as a back-end. We note that there is a tension between termination analysis and All-Termination analysis. Since termination analysis tends to be expensive, the goal is to decide termination as quickly as possible, using the least amount of analysis. For example, the calling context graph termination analysis algorithm, as implemented in the ACL2s system, is hierarchical: it starts with relatively inexpensive and coarse termination analyses and gradually works its way up to heavyweight methods, if needed. On the other hand, we can get better All-Termination(T) results by employing the heavyweight methods to extract all the reasons for termination; but this involves more work. The algorithms we develop take this tension into account and are *responsive* in that they answer the basic termination question first, without incurring any additional overhead. Only once termination is settled do they proceed with the full All-Termination(T) analysis. This is something that can be done in the background or off-loaded to an unused CPU core. This approach allows a theorem prover to use spare CPU cycles to detect new induction schemes.

We end with conclusions and future work in Section 8, after briefly discussing related work in Section 7.

2 All-Termination(T)

Recall that a well-order $>$ is an order with no infinite descending chains $x_1 > x_2 > \cdots$. Traditionally, programs are proved terminating via *measures*, which are functions mapping program states to some well-ordered set so that every state transition leads to a decrease in measure. In general, a terminating program may have many distinct measures. While each measure alone is sufficient to show that the program terminates, different measures may correspond to different *reasons* that the program terminates.

Example 1. Consider the following function `zip` which takes a pair of lists and produces a list of pairs:

```
zip [] ys = []
zip xs [] = []
zip xs ys = (hd x, hd y):(zip (tl xs) (tl ys))
```

If we consider a program state for this function to be a pair of actual arguments for `xs` and `ys`, there are three natural measures: `length(xs)`, `length(ys)`, and `length(xs) + length(ys)`, where `length` measures the length of a list. While there are many ways to distinguish measures, a particularly simple property of a measure is the set of formal arguments it depends on. A subset of the formals for which a measure exists is called a *measurable subset* [4]. This turns out to be a very useful notion. First, we can construct an induction scheme that depends only on the program and the measurable set, but not on the measure, so the theorem prover does not have to reason about the measure. Second, we can extract induction schemes even if our termination analysis does not construct an explicit measure (a property of many current termination analyses). Third, we can statically determine the instantiations required in the induction step of the induction schemes; this makes automation possible, since finding appropriate instantiations is hard.

Boyer and Moore show how to construct such induction schemes [4]. Here, we just give some examples for `zip`. The first two measures for `zip` are particularly interesting, because they tell us that `zip`'s termination really depends on only one of its arguments. The measure `length(xs)` gives rise to the measurable subset {`xs`}, which leads to the following induction scheme.

$$\left[\begin{array}{l} (\forall \texttt{ys} \,.\, \varphi(\texttt{[], ys})) \wedge \\ (\forall \texttt{xs}.\, \varphi(\texttt{xs, []})) \wedge \\ \left(\begin{array}{l} \forall \texttt{x}, \texttt{y}, \texttt{xs}, \texttt{ys} \,.\, (\forall \texttt{zs} \,.\, \varphi(\texttt{xs, zs})) \\ \qquad \rightarrow \varphi(\texttt{x:xs, y:ys}) \end{array}\right) \end{array}\right] \rightarrow \forall \texttt{xs}, \texttt{ys} \,.\, \varphi(\texttt{xs, ys})$$

The induction scheme for the measure `length(ys)` is similar. On the other hand, the measure `length(xs)+length(ys)` gives rise to the measurable subset {`xs`, `ys`}, which leads to the following induction scheme.

$$\left[\begin{array}{l} (\forall \texttt{ys}.\, \varphi(\texttt{[], ys})) \wedge \\ (\forall \texttt{xs}.\, \varphi(\texttt{xs, []})) \wedge \\ (\forall \texttt{x}, \texttt{y}, \texttt{xs}, \texttt{ys} \,.\, \varphi(\texttt{xs, ys}) \rightarrow \varphi(\texttt{x:xs, y:ys})) \end{array}\right] \rightarrow \forall \texttt{xs}, \texttt{ys} \,.\, \varphi(\texttt{xs, ys})$$

Notice that the above induction schemes can be applied to any φ, even if `zip` is not mentioned. Also notice that if we can prove φ using the induction scheme for {`xs`, `ys`}, then we can prove φ using the induction schemes for either {`xs`} or {`ys`}; the reverse implication does not hold. Finally, there are formulas that are provable using the induction scheme for {`xs`}, but not using the induction scheme for {`ys`} (and vice-versa).

The above examples highlight why the notion of a minimal measurable subset, or *termination core*, is useful. It turns out that many functions have more than one termination core [4], and All-Termination analysis is concerned with finding all such cores.

In the definition of All-Termination(T), we do not specify a particular programming language; instead, we postulate a universe of programs Prog, which we keep abstract. What we do require is that for every program $F \in$ 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 $\mathcal{F}$, formal parameter names $\mathcal{P}$, and argument values $\mathcal{V}$, we have:

Definition. A *semantic call graph* $\mathcal{C}$ is a pair $(S, \rightarrow)$ where

- $S \subseteq \mathcal{F} \times (\mathcal{P} \rightharpoonup \mathcal{V})$ is the set of *states*, and
- $\rightarrow \subseteq S \times S$ is the *transition relation*.

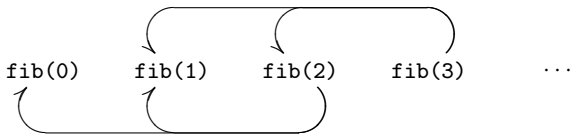
and $\mathcal{P} \rightharpoonup \mathcal{V}$ denotes partial maps from $\mathcal{P}$ to $\mathcal{V}$, *i.e.* from formals to actuals.

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

Example 2. Consider the Fibonacci function:

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

A fragment of the semantic call graph for `fib` might be displayed as follows:



Note that the semantic call graph $\mathcal{C}_F$ for a program F does not contain all the information present in F, since, for example, it is usually not possible to determine the actual result of a function using only $\mathcal{C}_F$. Only the function calls made in computing results can be determined. Nevertheless, in general $\mathcal{C}_F$ is an infinite, uncomputable structure: even determining whether there is a transition between two states is undecidable.

To connect semantic call graphs to measures, we have the following proposition, which is a basic result from the theory of ordinal numbers:

Proposition 3. $\mathcal{C} = (S, \rightarrow)$ is terminating iff there exists a well-ordered set $(X, >)$ and a measure $\mu : S \rightarrow X$, *i.e.*, a map μ such that if $s, t \in S$ and $s \rightarrow t$ then $\mu(s) > \mu(t)$.

The arguments $V(x)$ present in a state $(f, V) \in \mathcal{C}$ are observations available to a measure on $\mathcal{C}$. Thus, in order to restrict the arguments a measure can observe, we just restrict the arguments available in the states of $\mathcal{C}$.

Definition. Given $P \subseteq \mathcal{P}$, we say

(1) The *restriction* of a partial function $V : \mathcal{P} \rightharpoonup \mathcal{V}$ to P is the function

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

(2) The *restriction* of a state $s = (f, V)$ to P is $s \restriction P = (f, V \restriction P)$.

We are now in a position to formally define All-Termination(T).

Definition.

(1) $P \subseteq \mathcal{P}$ is a *measurable set* of formal arguments for $\mathcal{C}$ if there exists a measure $\mu : \mathcal{C} \to X$ such that $\mu(s) = h(s \restriction P)$ for some function h. Note that if $\mathcal{C}$ has any measurable set, then in particular $\mathcal{C}$ is terminating (by Proposition 3).

(2) A *termination analysis* T is a predicate on programs F and sets of formal parameters P such that if $T(F, P)$ then P is a measurable set for $\mathcal{C}$. Let TA be the set of all termination analyses.

(3) A termination analysis T is *monotonic* if, whenever $T(F, P)$ and $P \subseteq Q$, we have $T(F, Q)$.

(4) The function All-Termination : TA $\to$ (Prog $\to 2^{\mathcal{P}}$) is defined as follows:

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

Note that we do not require termination analyses to be monotonic. This underscores the fact that giving more information to a termination analysis (by including more function arguments) may obscure an actual, simpler reason that a function terminates. The polynomial size-change analysis we consider in Section 5 is nonmonotonic in this sense. On the other hand, if P is a measurable set for $\mathcal{C}$ and $P \subseteq Q$, then clearly Q is a measurable set of $\mathcal{C}$ as well. Thus, we (safely) define All-Termination(T)(F) to be upward-closed under set inclusion. An important implication of this setup is that All-Termination(T)(F) can be represented by its minimal elements, the termination cores of F modulo T. Since this representation may be exponentially more concise than the full set of measurable sets, and since most applications only use the termination cores, all of the All-Termination(T) algorithms we consider enumerate only the termination cores.

Our first complexity result for All-Termination(T) depends only on the complexity of T:

Theorem 4.

(1) If $\exists P \,.\, T(F, P)$ is PSpace-hard, so is All-Termination(T).

(2) If Cl is a complexity class such that PSpace $\subseteq$ Cl, and $T \in$ Cl, then All-Termination(T) $\in$ Cl, using the representation by termination cores.

Proof. For (1), note that the decision problem $\exists P \,.\, T(F, P)$ can be reduced to All-Termination(T)(F) in polynomial space by executing All-Termination(T)(F) until it either halts with no output or produces its first output.

For (2), the algorithm is as follows.

Algorithm 5.

$\textsc{All-Termination}(T)(F)$
  for $P \subseteq \mathcal{P}$ do
    if $T(F, P)$ then
      if $\forall Q \subset P \;.\; T(F, Q) =$ False then output P

We consider $\mathcal{P}$ part of the input to $\textsc{All-Termination}(T)(F)$. The for-loop and $\forall Q \subset P$ test are both implementable in PSpace using counters whose size is logarithmic in the size of $2^{\mathcal{P}}$, hence linear in the size of $\mathcal{P}$. □

While this result is of theoretical interest, the algorithm is highly impractical since it executes T at least $2^{|\mathcal{P}|}$ times. Our strategy for constructing practical algorithms is to instrument T so that we only need to run it *once*, after which we can mine the data produced for termination cores.

There is one more general remark to be made before discussing specific analyses. Termination solvers are usually decision procedures for $\exists P \;.\; T(F, P)$, and many solvers do not actually construct a measure μ on $\mathcal{C}$, making it hard to determine the witness for P in the existential. However, given $\mathcal{C}$ and P, we can construct a new semantic call graph whose termination implies that P is a measurable set for $\mathcal{C}$.

Definition. Given a semantic call graph $\mathcal{C} = (S, \rightarrow)$, the *restriction* of $\mathcal{C}$ to P is $\mathcal{C} \restriction P = (\{s \restriction P \;:\; s \in S\}, \rightsquigarrow)$ where

$$s \rightsquigarrow t \iff \exists s' \rightarrow t' \in \mathcal{C} \;.\; s = s' \restriction P,\; t = t' \restriction P$$

Proposition 6. P is a measurable set for $\mathcal{C}$ iff $\mathcal{C} \restriction P$ is terminating.

Proposition 7. T is a termination analysis iff $T(F, P)$ implies $\mathcal{C}_F \restriction P$ is terminating.

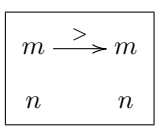
3 The size-change framework

Even though the semantic call graph $\mathcal{C}_F$ does not contain as much information as F itself, it is still uncomputable. 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 [17], which provides an example of this approach. We present the framework in the setting of semantic call graphs by defining a notion of *simulation* between them.

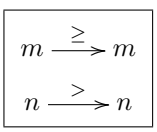
Example 8. Consider the well-known total function `ack`:

```
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. We first observe that in every recursive call to `ack`, either the first argument decreases, or the first argument stays fixed while the second decreases. We display this size-change data as follows:



G_1



G_2

The second observation is that a hypothetical infinite recursion would have to involve an infinite sequence of argument size changes of the form above. But we can show that this is not possible. Clearly if size change G_1 occurs infinitely often, then m decreases infinitely—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.

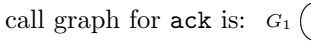
The size-change framework formalizes this kind of reasoning into a decidable analysis on programs.

For simplicity, we postulate a single well-ordering $>$ on all values in $\mathcal{V}$.[1] The notion of size-change "data" above is formalized into a structure called a *size-change graph*. A set of such graphs, as edges between function names, is called an *annotated call graph*:

Notation.

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

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. Finally, we use $\mathcal{P}(\mathcal{G})$ to denote the subset of $\mathcal{P}$ that appear in $\mathcal{G}$.

Example 9. The annotated call graph for `ack` is: 

.

The intuitive argument for the termination of `ack` by size-change was based on (infinite) sequences of size changes. We formalize these sequences as follows.

Definition. A *multipath* π over an ACG $\mathcal{G}$ is a (potentially infinite) sequence of edges from $\mathcal{G}$:

$$\pi = f_0 \xrightarrow{G_1} f_1 \xrightarrow{G_2} f_2 \xrightarrow{G_3} \cdots$$

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. Finally, we write $\pi : f_0 \to f_n$ if $\pi = f_0 \xrightarrow{G_1} \cdots \xrightarrow{G_n} f_n$.

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. 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 $i > 0$.

Example 10. Consider the multipath

$$\pi = \texttt{ack} \xrightarrow{G_1} \texttt{ack} \xrightarrow{G_2} \texttt{ack} \xrightarrow{G_1} \texttt{ack}$$

in $\mathcal{G}_{\texttt{ack}}$. The only thread in π 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 $n \xrightarrow{>} n \xrightarrow{>} n$.

Now we can characterize a termination proof using the size-change framework in terms of threads and multipaths.

[1] Multiple orders can also be handled [18].

Definition.

(1) An infinite thread $\langle x_i \xrightarrow{r_i} x_{i+1} \rangle$ 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.

An ACG $\mathcal{G}$ can be seen as a finite description of the semantic call graph $\mathcal{C}_\mathcal{G}$, which relates states according to the possible size changes given in $\mathcal{G}$:

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

$$(f, V) \rightarrow (g, U) \iff \exists f \xrightarrow{G} g \in \mathcal{G} \;.\forall x \xrightarrow{r} y \in G \;.\; V(x) \; r \; U(y)$$

To see why size-change termination is an appropriate notion of termination, consider the following tight relationship between the termination behavior of $\mathcal{G}$ and $\mathcal{C}_\mathcal{G}$:

Proposition 11. $\mathcal{G}$ is size-change terminating iff $\mathcal{C}_\mathcal{G}$ is terminating.

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. 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 if $s_1 \; R \; s_2$ then

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

Definition. $\mathcal{C}'$ *simulates* $\mathcal{C}$, written $\mathcal{C} \sqsubseteq \mathcal{C}'$, if there exists a simulation R between $\mathcal{C}$ and $\mathcal{C}'$ such that for every state $s \in \mathcal{C}$ there is some state $t \in \mathcal{C}'$ with $s \; R \; t$. We write $\mathcal{C} \approx \mathcal{C}'$ if $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C}' \sqsubseteq \mathcal{C}$.

Intuitively, if $\mathcal{C}'$ simulates $\mathcal{C}$, then $\mathcal{C}'$ admits at least as many behaviors as $\mathcal{C}$. Safety with respect to termination, as stated by the following proposition, is easy to show.

Proposition 12. If $\mathcal{C} \sqsubseteq \mathcal{C}'$ and $\mathcal{C}'$ is terminating then $\mathcal{C}$ is terminating.

We say $\mathcal{G}$ is *safe for* F if $\mathcal{C}_F \sqsubseteq \mathcal{C}_\mathcal{G}$. In general, determining a safe ACG $\mathcal{G}$ for a program F is difficult, and is a problem that the size-change framework does not seek to address (but see [18]). For our purposes, it is sufficient to postulate some function analyze : Prog $\rightarrow$ ACG with the property that analyze(F) is safe for F.

We now turn to some preliminary considerations for solving All-Termination within the size-change framework. The framework provides an example of a termination analysis that does not explicitly construct a measure. While it is possible to effectively construct a measure from a size-change analysis of a program F, and thereby extract a measurable set, the size of the measure is triply exponential [16]. Using the results from Section 2, we can instead use size-change termination to show that $\mathcal{C}_F \restriction P$ is terminating for some P. To do this, we derive from analyze(F) an ACG whose size-change termination implies the termination of $\mathcal{C}_F \restriction P$.

Definition. Given $P \subseteq \mathcal{P}$, we say

(1) The *restriction* of a size change graph G to P is $G \restriction P = \{x \xrightarrow{r} y \in G \;:\; x, y \in P\}$.

(2) The *restriction* of an ACG $\mathcal{G}$ to P is $\mathcal{G} \restriction P = \{f \xrightarrow{G\restriction P} g \ : \ f \xrightarrow{G} g \in \mathcal{G}\}$.

The key property of the restriction operator $\restriction P$ is that it commutes with $\mathcal{C}_-$, as stated in (3) below.

Lemma 13. For all $\mathcal{C}$ and $P \subseteq \mathcal{P}$ we have

(1) $\mathcal{C} \sqsubseteq \mathcal{C} \restriction P$, and

(2) if $\mathcal{C} \sqsubseteq \mathcal{C}'$ then $\mathcal{C} \restriction P \sqsubseteq \mathcal{C}' \restriction P$.

(3) $\mathcal{C}_{\mathcal{G}} \restriction P \approx \mathcal{C}_{\mathcal{G}\restriction P}$, and

Now we can define the size-change termination analysis in a way suitable for studying its All-Termination problem.

Definition. We define the predicate SCT as follows:

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

Theorem 14. SCT is a termination analysis.

Proof. Suppose $\mathrm{SCT}(F, P)$ is true. Using Lemma 13, we have $\mathcal{C}_{\text{analyze}(F)\restriction P} \approx \mathcal{C}_{\text{analyze}(F)} \restriction P$. Recall that $\text{analyze}(F)$ is safe for F, which means $\mathcal{C}_F \sqsubseteq \mathcal{C}_{\text{analyze}(F)}$. By Lemma 13, it follows that $\mathcal{C}_F \restriction P \sqsubseteq \mathcal{C}_{\text{analyze}(F)} \restriction P$. Hence $\mathcal{C}_F \restriction P \sqsubseteq \mathcal{C}_{\text{analyze}(F)\restriction P}$. But by assumption $\text{analyze}(F) \restriction P$ is size-change terminating, so $\mathcal{C}_{\text{analyze}(F)\restriction P}$ is terminating (Proposition 11) and therefore $\mathcal{C}_F \restriction P$ is terminating (Proposition 12). By Proposition 7, we're done. □

4 General size-change analysis

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 [17]. In this section, we review the standard algorithm and develop an instrumented version to compute All-Termination(SCT).

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 may 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. The following definitions serve this purpose.

Definition. We define composition of size-change labels and graphs as follows:

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

Definition. The *evaluation* of a multipath $\langle G_1, \ldots, G_n \rangle \in \mathcal{G}^+$ is

$$[\![G_1, \ldots, G_n]\!] = G_1 \cdot \ \cdots \ \cdot G_n$$

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

Proposition 15. $x \xrightarrow{r} y \in [\![\pi]\!]$ iff there exists a thread $\langle x \xrightarrow{r_0} z_1 \xrightarrow{r_1} \cdots \xrightarrow{r_{n-1}} z_n \xrightarrow{r_n} y \rangle$ in π, with $r = r_1 \cdot \ \cdots \ \cdot r_n$.

Given an ACG $\mathcal{G}$, we can construct its closure under composition:

Definition. The *closure* of $\mathcal{G}$ under $\cdot$ is the least set satisfying

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

The closure $\mathrm{cl}(\mathcal{G})$ has several useful properties:

Proposition 16.

(1) $\mathrm{cl}(\mathcal{G}) = \{f \xrightarrow{[\![\pi]\!]} g \ : \ \pi : f \to g \in \mathcal{G}^+\}$.

(2) if $\mathcal{G}$ is finite and each $G \in \mathcal{G}$ is finite then $\mathrm{cl}(\mathcal{G})$ is finite.

(3) $\mathcal{C}_{\mathrm{cl}(\mathcal{G})}$ is the transitive closure of $\mathcal{C}_{\mathcal{G}}$.

We assume that analyze(F) is finitary, which in particular means that we can compute cl(analyze(F)) as a least fixpoint. Our interest in the closure is that certain of its edges capture the size-change behavior of infinite multipaths in $\mathcal{G}$:

Definition. A size-change graph G is *idempotent* if $G \cdot G = G$.

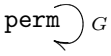
Theorem 17 (Lee *et al.* [17])**.** $\mathcal{G}$ is size-change terminating iff for every $f \xrightarrow{G} g \in \mathrm{cl}(\mathcal{G})$ such that G is idempotent, there is an edge $x \xrightarrow{>} x \in G$.

To give a sense for why consideration of the idempotent elements is sufficient to show size-change termination, we briefly sketch the proof for the right-to-left direction. Suppose $\pi \in \mathcal{G}^\omega$ is an infinite multipath. It is possible to show, using Ramsey's theorem, that for some $f \xrightarrow{G} f \in \mathcal{G}$ with G idempotent, π has an infinite suffix $\pi_1, \pi_2, \ldots$ with each $\pi_i : f \to f$ and $[\![\pi_i]\!] = G$. Clearly if G has an edge $x \xrightarrow{>} x$ then $\pi_1, \pi_2, \ldots$ has infinite descent.

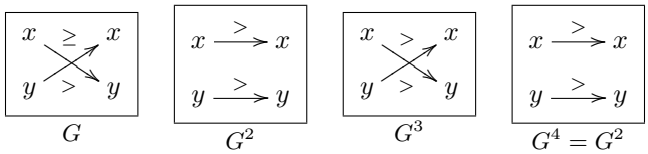
Example 18. Consider the following function `perm`, which permutes its two arguments, decreasing one of them, until one of them is zero.

```
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`'s single recursive call, along with its powers are:



Note that G^2 is idempotent. Consequently, the only size-change graphs in $\mathrm{cl}(\mathcal{G}_{\texttt{perm}})$ 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 $\mathrm{cl}(\mathcal{G}_{\texttt{perm}})$, $\mathcal{G}_{\texttt{perm}}$ is size-change terminating.

The standard algorithm for deciding size-change termination is based on Theorem 17: enumerate cl(analyze(F)), 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 completes. The minimal solutions to these constraints will be exactly the termination cores of F.

To apply Theorem 17 to the All-Termination(SCT) problem, we need to consider multipaths in $(\mathcal{G} \upharpoonright P)^+$ for $P \subseteq \mathcal{P}$. What we want is to be able to reason about these multipaths in terms of the multipaths of $\mathcal{G}^+$, so that we can characterize the size-change termination of $\mathcal{G} \upharpoonright P$ in terms of $\mathcal{G}$. We first observe that every multipath in $(\mathcal{G} \upharpoonright P)^+$ is the *restriction* of a multipath in $\mathcal{G}^+$, as follows.

Definition. 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 $\pi \upharpoonright P = f_0 \xrightarrow{G_1 \upharpoonright P} \cdots \xrightarrow{G_n \upharpoonright P} f_n$, which is a multipath in $(\mathcal{G} \upharpoonright P)^+$.

Proposition 19. If $\pi \in (\mathcal{G} \upharpoonright P)^+$ then there exists a $\pi' \in \mathcal{G}^+$ such that $\pi = \pi' \upharpoonright P$.

Note that this correspondence is not one-to-one: there may be many multipaths π in $\mathcal{G}^+$ with the same restriction $\pi \upharpoonright P$.

Given a particular $\pi \in \mathcal{G}^+$, there is also a connection between the threads of π and the threads of $\pi \upharpoonright P$:

Proposition 20. Let $\pi \in \mathcal{G}^+$ and $P \subseteq \mathcal{P}$. The threads of $\pi \upharpoonright P$ 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 $x \xrightarrow{\geq} y \in G$ and $y \xrightarrow{\geq} z \in H$, 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 $[\![\pi]\!]$, the most we can say is that $x \xrightarrow{r} y \in [\![\pi]\!]$ implies the existence of *some* thread in π from x to y, by Proposition 15. What Proposition 20 tells us is that, if we want to reason about the threads of $\pi \upharpoonright P$ (and hence the edges in $[\![\pi \upharpoonright P]\!]$) in terms of $[\![\pi]\!]$, we need to keep track of which variables contribute to each edge $x \xrightarrow{r} y \in [\![\pi]\!]$. To track this information, we introduce *annotated size-change graphs*.

Notation.

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

Intuitively, if an edge $x \xrightarrow[Q]{r} y$ is in an ASCG $\mathbb{G}$, then we know that there is some thread relating x to y with size-change r, but that thread might go through all 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 size-change graphs:

Definition. The annotated size-change graph corresponding to G is $\lfloor G \rfloor = \{x \xrightarrow[\{x,y\}]{r} y \ : \ x \xrightarrow{r} y \in G\}$.

Just as with size-change graphs, we have a notion of composition and evaluation for annotated size-change graphs.

Definition. Annotated composition and evaluation are defined as follows:

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

We also need the annotated closure $\mathrm{acl}(\mathcal{G})$ of an ACG $\mathcal{G}$, which has similar properties to the closure $\mathrm{cl}(\mathcal{G})$. Note, however, that $\mathrm{acl}(\mathcal{G})$ is likely to be much larger than $\mathrm{cl}(\mathcal{G})$.

Definition. The *annotated closure* of $\mathcal{G}$ under $\odot$ is the least set satisfying

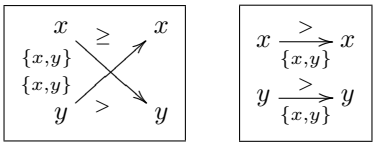
$$\begin{aligned} \mathrm{acl}(\mathcal{G}) = \ & \{f \xrightarrow{\lfloor G \rfloor} g \ : \ f \xrightarrow{G} g \in \mathcal{G}\} \cup \\ & \{f \xrightarrow{\mathbb{G} \odot \mathbb{H}} h \ : \ f \xrightarrow{\mathbb{G}} g, g \xrightarrow{\mathbb{H}} h \in \mathrm{acl}(\mathcal{G})\} \end{aligned}$$

Proposition 21.

(1) $\mathrm{acl}(\mathcal{G}) = \{f \xrightarrow{\lfloor \pi \rfloor} g \ : \ \pi : f \to g \in \mathcal{G}^+\}$.

(2) if $\mathcal{G}$ is finite and each $G \in \mathcal{G}$ is finite then $\mathrm{acl}(\mathcal{G})$ is finite.

Example 22. Returning to the `perm` example, we ask: is $\{\mathtt{x}\}$ 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 1` calls `perm 0 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 $\{\mathtt{y}\}$ 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 $\{\mathtt{x}\}$ and $\{\mathtt{y}\}$ are not measurable sets. We begin with the same graph G we had before, but with annotated edges.



G $\qquad$ $G \odot G$

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 annotated evaluation of a multipath $\pi \in \mathcal{G}^+$ gives us the desired information about the variables used in threads for π, as stated in the following lemma.

Lemma 23. If $\pi \in \mathcal{G}^+$ then:

(1) If $\langle x_1 \xrightarrow{r_1} x_2 \xrightarrow{r_2} \cdots \xrightarrow{r_{n-1}} x_n \rangle$ is a thread in π then $x_1 \xrightarrow[\{x_1, \ldots, x_n\}]{r_1 \cdots r_{n-1}} x_n \in \lfloor \pi \rfloor$.

(2) If $x \xrightarrow[Q]{r} y \in \lfloor \pi \rfloor$ then there exists a thread $\langle x \xrightarrow{r_0} z_1 \xrightarrow{r_1} \cdots \xrightarrow{r_{n-1}} z_n \xrightarrow{r_n} y \rangle$ in π such that $\{x, z_1, \ldots, z_n, y\} = Q$ and $r = r_1 \cdots r_n$.

Proof. Straightforward induction on π. □

The outcome of all this work is the next result, which shows that we can now reason about the multipaths of $(\mathcal{G} \upharpoonright P)^+$ in terms of $\mathcal{G}^+$.

Proposition 24. Let $\pi \in \mathcal{G}^+$. Then $x \xrightarrow{r} y \in [\![\pi \upharpoonright P]\!]$ iff there exists a $Q \subseteq P$ such that $x \xrightarrow[Q]{r} y \in \lfloor \pi \rfloor$.

Proof. Follows from Propositions 15 and 20, and Lemma 23. □

Before we can prove the main result for All-Termination(SCT), we need two more lemmas. The first is a straightforward consequence of viewing (annotated) evaluation and restriction as homomorphisms from multipaths to (annotated) size-change graphs.

Lemma 25. If $\pi \in \mathcal{G}^+$ then $[\![\pi]\!]^n = [\![\pi^n]\!]$, $\lfloor \pi \rfloor^n = \lfloor \pi^n \rfloor$, and $(\pi \upharpoonright P)^n = \pi^n \upharpoonright P$.

To motivate the second lemma, suppose $\pi \in \mathcal{G}^+$. Even if $[\![\pi]\!]$ is idempotent, $\lfloor \pi \rfloor$ might not be. The next lemma is important because it shows that some power of $\lfloor \pi \rfloor$ must be idempotent.

Lemma 26. If $\cdot$ is an associative binary operator and the set $\{a^n \ : \ n \in \mathbb{N}\}$ is finite, then there exists some $n \in \mathbb{N}$ such that a^n is idempotent.

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

Theorem 27. $\mathcal{G} \upharpoonright P$ is size-change terminating iff for every $f \xrightarrow{\mathbb{G}} g \in \mathrm{acl}(\mathcal{G})$ such that $\mathbb{G}$ is idempotent, there is an edge $x \xrightarrow[Q]{\geq} x \in \mathbb{G}$ with $Q \subseteq P$.

Proof. We prove the right-to-left direction (soundness); the other direction is similar. Let $\pi' \in (\mathcal{G} \upharpoonright P)^+$ such that $[\![\pi']\!]$ is idempotent. By Proposition 19, there is some $\pi \in \mathcal{G}^+$ such that $\pi \upharpoonright P = \pi'$. By Lemma 26, there is some n such that $\lfloor \pi \rfloor^n$ is idempotent. By Lemma 25, $\lfloor \pi \rfloor^n = \lfloor \pi^n \rfloor$, so $\lfloor \pi^n \rfloor$ is idempotent. By assumption, there is thus an edge $x \xrightarrow[Q]{\geq} x \in \lfloor \pi^n \rfloor$ with $Q \subseteq P$. Using Proposition 24, we have that $x \xrightarrow{\geq} x \in [\![(\pi^n) \upharpoonright P]\!]$. But $(\pi^n) \upharpoonright P = (\pi \upharpoonright P)^n = (\pi')^n$. Applying Lemma 25, it follows that $x \xrightarrow{\geq} x \in [\![\pi']\!]^n$. But $[\![\pi']\!]$ is idempotent, so $x \xrightarrow{\geq} x \in [\![\pi']\!]$. Appealing to Proposition 16 and Theorem 17, we have shown that $\mathcal{G} \upharpoonright P$ is size-change terminating. □

Corollary 28. Let $F \in$ Prog and $\mathcal{I} = \{\mathbb{G} \in \mathrm{acl}(\mathrm{analyze}(F)) \ : \ \mathbb{G} \text{ idempotent}\}$. Then All-Termination($SCT$)($F$) is the set

$$\{P \subseteq \mathcal{P} \ : \ \forall \mathbb{G} \in \mathcal{I} \ . \ \exists x \xrightarrow[Q]{\geq} x \in \mathbb{G} \ . \ Q \subseteq P\}$$

We can use Corollary 28 as the basis for an algorithm as follows. First, compute $\mathrm{acl}(\mathrm{analyze}(F))$ 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 $\bigvee_{x \xrightarrow[Q]{\geq} x \in \mathbb{G}} \bigwedge_{y \in Q} y$. The collection (conjunction) of these constraints is a constraint system Φ_F whose solutions are the elements of All-Termination(SCT).

Recall that a formula in conjunctive normal form (CNF) is the conjunction of a set of *clauses*, each of which is a disjunction of *literals*. A literal is just a possibly negated variable. A clause is *dual-horn* if it contains at most one negated variable. By introducing variables, the constraint system Φ_F above can be expressed as a conjunction of dual-horn clauses. This is a useful observation because there is an *output-sensitive* algorithm for enumerating the minimal solutions to dual-horn formulas, which we discuss in Section 6. Output-sensitivity means that the running time of the algorithm is bounded by the number of *outputs* it produces. In theory, the constraint system Φ_F may have an exponential number of minimal

solutions, but the applications we have in mind tend to have only a relatively small number of minimal solutions, so an output-sensitive algorithm provides a "pay-as-you-go" solution to the enumeration problem.

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 problem cannot be shown to terminate, there is no need to continue; otherwise, it proceeds to solve the All-Termination(T) problem. We note that for the theorem proving application, 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. Responsiveness is obtained by controlling the least fixpoint computation of acl(analyze(F)) so that we only generate the size change graphs needed to compute cl(analyze(F)). We note that this process differs from the basic size change algorithm only in that we need to record the size change graph annotations required for the annotated closure. Once termination is established, the fixpoint computation for the annotated closure is allowed to proceed to completion.

5 Polynomial size-change analysis

Ben-Amram and Lee observed that the PSpace algorithm for size-change termination (presented in the previous section) can easily lead to exponential running times for reasonable programs [2]. Consequently, they developed a PTime approximation to the size-change termination problem. In this section, we review the algorithm, derive a termination analysis *SCP*, and study its All-Termination problem. The algorithm we present for All-Termination(*SCP*), like the algorithm in the previous section, reduces the problem to enumerating the minimal satisfying assignments of a dual Horn formula. We close the section with a hardness result for the problem.

5.1 The PTime size-change algorithm

The PTime size-change condition is based on the idea of *loop anchors*, formalized as follows.

Definition. $G \in \mathcal{G}$ 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.

Let SCC($\mathcal{G}$) denote the set of nontrivial, strongly-connected components of $\mathcal{G}$; each element of SCC($\mathcal{G}$) is another annotated call graph. The basic idea of the PTime algorithm is to locate the loops through an ACG $\mathcal{G}$ by computing SCC($\mathcal{G}$), find and remove anchors for the loops, and recur:

Algorithm 29 (Ben-Amram and Lee [2])**.**

```
AncTerm(G)
    for H ∈ SCC(G) do
        A := FindAnchors(H)
        if A = ∅ then return False
        if AncTerm(H \ A) = False then return False
    return True
```

Note that after removing anchors in one iteration, new anchors may be found in the next. The key element, of course, is the implementation of FindAnchors.

Theorem 30 (Ben-Amram and Lee [2])**.** If FindAnchors($\mathcal{H}$) returns a set of anchors of $\mathcal{H}$, then AncTerm is sound approximation of size-change termination. If FindAnchors($\mathcal{H}$) returns *all* of the anchors in $\mathcal{H}$, then AncTerm is a decision procedure for size-change termination.

To develop a PTime algorithm, 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*. Writing $\mathrm{src}(G)$ for the set $\{x \ : \ \exists r, y \ . \ x \xrightarrow{r} y \in G\}$, we define:

Definition. 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 31. 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.

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.

(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 simply 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.

Definition.

(1) The *no-descent set* for $\mathcal{G}$ is

$$\mathrm{ND}(\mathcal{G}) = \left\{ x \xrightarrow{\geq} y \in G \in \mathcal{G} \ : \ \exists \pi \in \mathcal{G}^+ \ . \ \begin{array}{l} y \xrightarrow{\geq} x \in [\![\pi]\!] \,, \\ y \xrightarrow{\geq} x \notin [\![\pi]\!] \end{array} \right\}$$

(2) We define $\mathcal{G} \rhd G = (\mathcal{G} \setminus G) \cup \{G \setminus \mathrm{ND}(\mathcal{G})\}$.

(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 annotated call graph.

Definition.

(1) The *transposition* of a size-change graph G is $G^t = \{y \xrightarrow{r} x \ : \ x \xrightarrow{r} y \in G\}$.

(2) The *transposition* of an ACG $\mathcal{G}$ is $\mathcal{G}^t = \{g \xrightarrow{G^t} f \ : \ f \xrightarrow{G} g \in \mathcal{G}\}$.

Proposition 32. G is an anchor for $\mathcal{G}$ iff G^t is an anchor for $\mathcal{G}^t$.

Despite the fact that the 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. Note, however, that thread preservers are closed under union; hence, there is a unique maximum thread preserver.

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

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 can be done in quadratic time. These observations lead to the following anchor-finding procedure, which takes overall quadratic time.

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

We write AncTermSCP for Algorithm 29 instantiated with FindAnchorsSCP.

5.2 The termination analysis *SCP*

In order to state the All-Termination(*SCP*) problem, we first need to define *SCP* as a termination analysis in the sense of Section 2—that is, we need to say what it means for PTime size-change to be restricted to the consideration of some set of formal arguments P. Doing this in an appropriate way turns out to be somewhat delicate. We begin with a candidate definition based on restricted ACGs (*c.f.* Section 3), and show why the candidate fails to be a suitable definition. We then define *SCP* and develop an algorithm for All-Termination(*SCP*).

A natural candidate for defining *SCP* is the predicate T:

$$T(F, P) \iff \textsc{AncTermSCP}(\mathrm{analyze}(F) \restriction P)$$

We first observe, by Theorem 30, that $T(F, P)$ implies $\mathrm{SCT}(F, P)$ (defined in Section 3). Hence, by Theorem 14, T is a termination analysis. Another interesting observation is that T is nonmonotonic. This is because of the restriction that type-1 anchors use only the maximum thread preserver: $\mathrm{MTP}(\mathcal{G})$ may fail to have strict fan-in even though for some thread-preserver P, $\mathcal{G} \restriction P$ does have strict fan-in. Nonmonotonicity leads to problems with the definition of T:

Theorem 33. Deciding $\exists P \ . \ T(F, P)$ is NP-hard.

$$
\begin{array}{rcll}
\tau & ::= & \langle \alpha_1\tau_1,\ \ldots,\ \alpha_n\tau_n\rangle & \text{anchor tree} \\
\alpha & ::= & \mathrm{T}_1(G) & \text{type-1 anchor} \\
 & | & \mathrm{T}_2(G) & \text{type-2 anchor} \\
 & | & \alpha \wedge \alpha' & \text{anchor conjunction} \\
 & | & \alpha^t & \text{anchor transposition} \\
 & | & \top & \text{true}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{G} & \models & \langle \alpha_1\tau_1,\ \ldots,\ \alpha_n\tau_n\rangle \text{ iff} \\
 & & \quad \mathrm{SCC}(\mathcal{G}) = \{\mathcal{G}_1,\ \ldots,\ \mathcal{G}_n\}, \\
 & & \quad \forall \mathcal{G}_i\ .\ \mathcal{G}_i \models \alpha_i,\ \mathcal{G}_i - \mathrm{anchors}(\alpha_i) \models \tau_i \\
\mathcal{G} & \models & \mathrm{T}_1(G) \text{ iff} \\
 & & \quad \exists P\ .\ \left\{ \begin{array}{l} G \in \mathcal{G},\ P \in \mathrm{TP}(\mathcal{G}), \\ \mathcal{G} \restriction P \text{ strict fan-in},\ \exists x \xrightarrow{>} y \in G \restriction P \end{array} \right. \\
\mathcal{G} & \models & \mathrm{T}_2(G) \text{ iff} \\
 & & \quad \exists P\ .\ \left\{ \begin{array}{l} G \in \mathcal{G}, \\ \mathrm{MTP}(\mathcal{G} \restriction P \rhd G \restriction P) \cap \mathrm{src}(G \restriction P) \neq \emptyset \end{array} \right. \\
\mathcal{G} & \models & \alpha_1 \wedge \alpha_2 \text{ iff } \mathcal{G} \models \alpha_1,\ \mathcal{G} \models \alpha_2 \\
\mathcal{G} & \models & \alpha^t \text{ iff } \mathcal{G}^t \models \alpha \\
\mathcal{G} & \models & \top \text{ always}
\end{array}
$$

$$
\frac{\begin{array}{c}\mathrm{SCC}(\mathcal{G}) = \{\mathcal{G}_1,\ \ldots,\ \mathcal{G}_n\} \\ \forall \mathcal{G}_i\ .\ \mathcal{G}_i \vdash_P \alpha_i,\ (\mathcal{G}_i \setminus \mathrm{anchors}(\alpha_i)) \vdash_P \tau_i\end{array}}{\mathcal{G} \vdash_P \langle \alpha_1\tau_1,\ \ldots,\ \alpha_n\tau_n\rangle}\ (\textsc{Tree})
$$

$$
\frac{\begin{array}{cc} G \in \mathcal{G} & \mathcal{G} \restriction \mathrm{MTP}(\mathcal{G}) \text{ strict fan-in} \\ P \in \mathrm{TP}(\mathcal{G}) & \exists x \xrightarrow{>} y \in G \restriction P \end{array}}{\mathcal{G} \vdash_P \mathrm{T}_1(G)}\ (\textsc{Ty1})
$$

$$
\frac{G \in \mathcal{G} \qquad P \cap \mathrm{src}(G) \neq \emptyset \qquad P \in \mathrm{TP}(\mathcal{G} \rhd G)}{\mathcal{G} \vdash_P \mathrm{T}_2(G)}\ (\textsc{Ty2})
$$

$$
\frac{\mathcal{G} \vdash_P \alpha_1 \qquad \mathcal{G} \vdash_P \alpha_2}{\mathcal{G} \vdash_P \alpha_1 \wedge \alpha_2}\ (\textsc{Conj}) \qquad \frac{\mathcal{G}^t \vdash_P \alpha}{\mathcal{G} \vdash_P \alpha^t}\ (\textsc{Trans})
$$

$$
\frac{}{\mathcal{G} \vdash_P \top}\ (\textsc{True}) \qquad \frac{\mathcal{G} \vdash_Q \alpha \qquad Q \subseteq P}{\mathcal{G} \vdash_P \alpha}\ (\textsc{Weaken})
$$

Figure 1: Syntax, semantics, and proof system for anchor trees

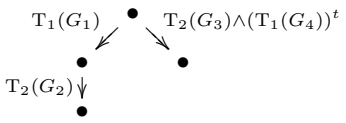
We take this result as evidence that T does not appropriately capture the restriction of polynomial size-change to a set of formal arguments P. This determination is not a formal one: the choice of predicate is in many ways a design issue. In any case, the predicate *SCP* that we define below has the property that $\exists P\ .\ SCP(F, P)$ is no harder to decide than polynomial size-change for analyze(F).

The basis for the *SCP* predicate is the idea of an *anchor tree*, which can be viewed as a certificate produced by AncTermSCP($\mathcal{G}$). An anchor tree is simply a record of the anchors found for each strongly-connected component of $\mathcal{G}$, and in each ensuing recursive call to AncTermSCP. The syntax of anchor trees τ and *anchor formulas* α is given in Figure 1.

Definition. The anchors of an anchor formula α are given by

$$\begin{aligned}
\text{anchors}(\top) &= \emptyset \\
\text{anchors}(\mathrm{T}_1(G)) &= \{G\} \\
\text{anchors}(\mathrm{T}_2(G)) &= \{G\} \\
\text{anchors}(\alpha_1 \wedge \alpha_2) &= \text{anchors}(\alpha_1) \cup \text{anchors}(\alpha_2) \\
\text{anchors}(\alpha^t) &= \text{anchors}(\alpha)
\end{aligned}$$

Example 34. The anchor tree



written $\left\langle \mathrm{T}_1(G_1)\ \langle \mathrm{T}_2(G_2)\ \langle\rangle\rangle\ , \mathrm{T}_2(G_3) \wedge (\mathrm{T}_1(G_4))^t\ \langle\rangle \right\rangle$, records the following hypothetical transcript from AncTermSCP on an ACG $\mathcal{G}$.

- $\mathcal{G}$ has two nontrivial strongly-connected components.
- In the first component, G_1 is a type-1 anchor. After removing G_1 from the first component, the rest of the component remains strongly-connected, and G_2 becomes a type-2 anchor. After removing G_2, no more nontrivial strongly-connected subcomponents remain.
- In the second component, G_3 is a type-2 anchor and ${G_4}^t$ is a type-1 anchor for the component's transposition. After removing G_3 and G_4 from the second component, no more nontrivial strongly-connected subcomponents remain.

An anchor tree τ can be viewed as an assertion about an ACG $\mathcal{G}$. We introduce a satisfaction relation $\models$ between ASCGs and anchor formulas, and between ASCGs and anchor trees, in Figure 1. Note that the conditions for $\mathcal{G} \models \mathrm{T}_2(G)$ are looser than that G is a type-2 anchor for $\mathcal{G}$; in fact, the conditions only require that G be a type-2 anchor for some restriction $\mathcal{G} \restriction P$. We need this generalization in order to prove Lemma 44 below.

Theorem 35.

(1) If $\mathcal{G} \models \alpha$ then each $G \in \text{anchors}(\alpha)$ is an anchor for $\mathcal{G}$.

(2) If $\mathcal{G} \models \tau$ then $\mathcal{G}$ is size-change terminating.

Proof. We just sketch the one interesting case in this proof: justifying that $\mathcal{G} \models \mathrm{T}_2(G)$ implies that G is an anchor for $\mathcal{G}$. Notice that, by definition, $\mathcal{G} \models \mathrm{T}_2(G)$ implies that $G \restriction P$ is a (type-2) anchor for $\mathcal{G} \restriction P$, for some $P \subseteq \mathcal{P}$. To see that G is therefore an anchor for $\mathcal{G}$, let $\pi \in \mathcal{G}^\omega$ be an infinite multipath passing through G infinitely often. Then $\pi \restriction P$ is an infinite multipath in $(\mathcal{G} \restriction P)^\omega$ passing through $G \restriction P$ infinitely often (Proposition 19). Since $G \restriction P$ is an anchor for $\mathcal{G} \restriction P$, there is some suffix of $\pi \restriction P$ that has infinite descent. By Proposition 20 (generalized to infinite multipaths), there is some suffix of π with infinite descent. □

The $\models$ relation establishes a semantics for anchor trees. We present the *SCP* predicate via a *proof system* $\vdash$ for anchor trees, given in Figure 1. The proof system is a generalization of the polynomial size-change algorithm in two ways. First, and most importantly, derivations are *annotated* with a subset $P \subseteq \mathcal{P}$, recording the formal arguments required to justify the derivation. These annotations are written as a subscript of the turnstile: $\mathcal{G} \vdash_P \tau$ is the conclusion of a proof that $\mathcal{G}$ satisfies the anchor tree τ, as long as the variables in P are considered. Crucially, there is an inference rule Weaken, using which we can conclude $\mathcal{G} \vdash_Q \tau$ whenever $\mathcal{G} \vdash_P \tau$ and $P \subseteq Q$; the soundness of this rule provides the main justification for our All-Termination(*SCP*) algorithm.

The second generalization the proof system provides is more subtle. In AncTerm (Algorithm 29), *all* of the anchors returned by FindAnchors are removed before AncTerm recurs. Removing anchors creates additional opportunities for finding thread-preservers: $\mathrm{TP}(\mathcal{G} \setminus \{G\}) \supseteq \mathrm{TP}(\mathcal{G})$. Superficially, having additional thread-preservers seems to make it easier to find anchors. However, a consequence is that $\mathrm{MTP}(\mathcal{G} \setminus \{G\}) \supseteq \mathrm{MTP}(\mathcal{G})$, and in particular, $\mathrm{MTP}(\mathcal{G})$ may have strict fan-in even though $\mathrm{MTP}(\mathcal{G} \setminus \{G\})$ does not. This point again underscores that polynomial size-change is a nonmonotonic analysis: by giving it "better" information (a larger collection of thread preservers), it may produce "worse" results (by failing to prove termination). The outcome of these considerations is that the choice to remove all of the anchors found by FindAnchors is a significant one, and is only one possible strategy. For the proof system $\vdash$, it is simpler and more natural to leave the anchor-removal strategy unspecified.

Whatever strategy for anchor removal is actually used, the outcome will be a particular anchor tree τ for which $\mathcal{G} \vdash_{\mathcal{P}} \tau$ is provable. Note that the annotation here is $\mathcal{P}$; we can apply Weaken to every leaf of a derivation tree to lift the leaf's annotation to $\mathcal{P}$. We take the perspective that we are *instrumenting* an existing PTime analysis so that it produces an anchor tree as a certificate. Let InstrSCP be an *instrumented polynomial size-change algorithm*: a function that takes an ACG $\mathcal{G}$ and, in polynomial time, either produces an anchor tree τ such that $\mathcal{G} \vdash_{\mathcal{P}} \tau$, or the symbol $\bot$, denoting failure. AncTermSCP can be instrumented this way without altering its complexity.

An analysis like polynomial size-change may produce anchor trees that are *redundant*, in the sense that they state that a given size-change is an anchor in multiple ways. While this information is redundant from the perspective of termination analysis alone, it is useful for All-Termination. For example, let $\alpha = \mathrm{T}_1(G) \wedge \mathrm{T}_2(G)$, so that α asserts that G is both a type-1 and a type-2 anchor for some ACG $\mathcal{G}$. Given some $P \neq Q$, it might be the case that for $\mathcal{G} \restriction P$, $G \restriction P$ is only type-1 anchor, and for $\mathcal{G} \restriction Q$, $G \restriction Q$ is only a type-2 anchor. We will use the certificate produced by an analysis to decide where to look for anchors in restricted ACGs like $G \restriction P$, so a certificate with "redundant" anchors like α will help uncover more measurable sets of arguments. We use the *subsumption relation* $\leq$, defined below, to formalize these ideas: if $\tau \leq \tau'$, then τ and τ' both state that the same set of size-change graphs are anchors, but τ' states that they are anchors in more ways than τ does.

Definition.

(1) The *subsumption* relation $\leq$ on anchor formulas is defined by:

$$\begin{array}{rcll} \alpha & \leq & \alpha & \\ \top & \leq & \mathrm{T}_1(G) & \text{for all } G \\ \top & \leq & \mathrm{T}_2(G) & \text{for all } G \\ \alpha_1 \wedge \alpha_2 & \leq & \alpha_1' \wedge \alpha_2' & \text{if } \alpha_1 \leq \alpha_1',\ \alpha_2 \leq \alpha_2' \\ \alpha^t & \leq & (\alpha')^t & \text{if } \alpha \leq \alpha' \end{array}$$

(2) The *subsumption* relation $\leq$ on anchor trees is defined by:

$$\begin{array}{rcl} \langle \alpha_i \tau_i \rangle & \leq & \langle \alpha_i' \tau_i' \rangle \quad \text{if} \\ & & \forall i \ .\ \alpha_i \leq \alpha_i',\ \tau_i \leq \tau_i',\ \mathrm{anchors}(\alpha_i) = \mathrm{anchors}(\alpha_i') \end{array}$$

We can now define *SCP*.

Definition. The *SCP* predicate is defined as

$$SCP(F,P) \iff \exists \tau, \tau' \ . \ \begin{cases} \tau \leq \tau' \\ \textsc{InstrSCP}(\mathrm{analyze}(F)) = \tau' \\ \mathrm{analyze}(F) \vdash_P \tau \end{cases}$$

In summary: the *SCP* termination analysis, intuitively, has two parts. Given F and P, the first part of *SCP* analysis is running an instrumented polynomial size-change analysis, InstrSCP, which produces a certificate τ' if it succeeds. Note that InstrSCP is not given P as input. The second part of *SCP* is looking for a smaller certificate $\tau \leq \tau'$ sufficient to prove F terminating using only the arguments in P. In the next subsection, we show how to find the termination cores for a program using *SCP*, in time exponential in the number of cores produced (or desired). We also show that this is the best result we can hope for.

We write $\alpha \restriction P$ for α with each size change graph G replaced by $G \restriction P$.

Theorem 36. If $\mathcal{G} \vdash_P \tau$ then $\mathcal{G} \restriction P \models \tau \restriction P$.

Proof. See Appendix A. □

5.3 An algorithm for All-Termination(*SCP*)

To describe our All-Termination(*SCP*) algorithm, we need a bit of notation from propositional logic. If φ is a propositional formula and A is a set of variables (truth assignment), we say $A \models \varphi$ iff A satisfies φ. We write $\top$ for the formula satisfied by every variable assignment, $\overline{\varphi}$ for the negation of φ, and $\varphi \Rightarrow \psi$ for $\overline{\varphi} \vee \psi$. Our algorithm depends on a function Φ from ACGs and anchor trees to propositional formulas, given in Figure 5.3 and described below.

Algorithm 37.

All-Termination(*SCP*)(*F*)
  $\mathcal{G} :=$ analyze(F)
  if InstrSCP($\mathcal{G}$) $= \tau$ then return $\min\{A \cap \mathcal{P} \ :\ A \models \Phi(\mathcal{G}, \tau)\}$
  else return $\emptyset$

Theorem 38. If InstrSCP(analyze(F)) $= \tau$ then

$$\{P \ :\ SCP(F,P)\} = \{A \cap \mathcal{P} \ :\ A \models \Phi(\mathcal{G}, \tau)\}$$

Anchor tree constraints

$$\Phi(\mathcal{G}, \langle \alpha_i \tau_i \rangle) = \bigwedge_i \Phi(\mathcal{G}, \alpha_i \tau_i)$$

Anchor branch constraints

$$\begin{aligned}
\Phi(\mathcal{G}, \alpha\tau) &= \Phi(\mathcal{G}, \alpha) \\
&\wedge \Phi(\mathcal{G} - \text{anchors}(\alpha), \tau) \\
&\wedge \bigwedge_{G \in \text{anchors}(\alpha)} \bigvee_{\kappa \in K(G) \cap K(\alpha)} \kappa
\end{aligned}$$

Anchor formula constraints

$$\begin{aligned}
\Phi(\mathcal{G}, \mathrm{T}_1(G)) &= \Theta_{G,1}(\mathcal{G}) \\
&\wedge \kappa_{G,1} \Rightarrow \left(\bigvee_{x \xrightarrow{\geq} y \in G} xy_{G,1} \right) \\
&\wedge \bigwedge_{x \xrightarrow{\geq} y \in G} (xy_{G,1} \Rightarrow x_{G,1}) \\
&\wedge \bigwedge_{x \xrightarrow{\geq} y \in G} (xy_{G,1} \Rightarrow y_{G,1}) \\
&\wedge \bigwedge_{x \in \mathcal{P}(\mathcal{G})} (x_{G,1} \Rightarrow x) \\
\Phi(\mathcal{G}, \mathrm{T}_2(G)) &= \Theta_{G,2}(\mathcal{G} \rhd G) \\
&\wedge \kappa_{G,2} \Rightarrow \left(\bigvee_{x \in \text{src}(G)} x_{G,2} \right) \\
&\wedge \bigwedge_{x \in \mathcal{P}(\mathcal{G})} (x_{G,2} \Rightarrow x) \\
\Phi(\mathcal{G}, \alpha \wedge \alpha') &= \Phi(\mathcal{G}, \alpha) \wedge \Phi(\mathcal{G}, \alpha') \\
\Phi(\mathcal{G}, \alpha^t) &= \Phi(\mathcal{G}^t, \alpha) \\
\Phi(\mathcal{G}, \top) &= \top
\end{aligned}$$

Thread preserver constraints

$$\begin{aligned}
\Theta_{H,i}(\mathcal{G}) &= \bigwedge_{G \in \mathcal{G}} \Theta_{H,i}(G) \\
\Theta_{H,i}(G) &= \bigwedge_{x \in \text{src}(G)} \left(x_{H,i} \Rightarrow \left(\bigvee_{x \xrightarrow{r} y \in G} y_{H,i} \right) \right)
\end{aligned}$$

Control variables

$$\begin{aligned}
K(\top) &= \emptyset \\
K(\mathrm{T}_1(G)) &= \{\kappa_{G,1}\} \\
K(\mathrm{T}_2(G)) &= \{\kappa_{G,2}\} \\
K(\alpha^t) &= K(\alpha) \\
K(\alpha_1 \wedge \alpha_2) &= K(\alpha_1) \cup K(\alpha_2) \\
K(G) &= \{\kappa_{G,1},\ \kappa_{G,2},\ \kappa_{G^t,1},\ \kappa_{G^t,2}\}
\end{aligned}$$

Figure 2: Dual-horn constraints for ALL-TERMINATION(SCP)

The basic idea of the algorithm is as follows. If $\textsc{InstrSCP}(\mathcal{G}) = \tau$, then we know that $\exists P \,.\, \mathcal{G} \vdash_P \tau$. Consider, for example, an anchor $\mathrm{T}_1(G)$ in τ. It must be that $\mathcal{G} \vdash_P \mathrm{T}_1(G)$ for some P, and the derivation of $\mathcal{G} \vdash_P \mathrm{T}_1(G)$ must have as a leaf an application of Ty1. This means that $\mathcal{G} \restriction \mathrm{MTP}(\mathcal{G})$ has strict fan-in. Hence, if we can find any thread preserver Q such that $\exists x \xrightarrow{>} y \in G \restriction Q$, then we can replace the leaf of the derivation with a new application of Ty1, this time annotated with Q. A similar argument can be made for type-2 anchors. In either case, the constraints we must check of an annotation Q to replace the derivation leaf's annotation are that (1) $Q \in \mathrm{TP}(\mathcal{G})$ for some $\mathcal{G}$ and (2) Q contains a variable or two variables from a finite collection of choices. It turns out that these constraints can easily be expressed as propositional formulas φ such that $Q \models \varphi$ iff Q is a valid leaf annotation.

The Weaken rule allows us to choose any annotation we like at the leaves of a derivation, as long as the annotation of the entire derivation is the union of all the leaf annotations. For example, if we have leaves $\mathcal{G} \vdash_P \alpha$ and $\mathcal{G} \vdash_P \alpha'$, but we know that $\mathcal{G} \vdash_Q \alpha$ and $\mathcal{G} \vdash_{Q'} \alpha'$, we can transform a derivation as follows:

$$\frac{\mathcal{G} \vdash_P \alpha \quad \mathcal{G} \vdash_P \alpha'}{\mathcal{G} \vdash_P \alpha \wedge \alpha'} \quad \Longrightarrow \quad \frac{\dfrac{\mathcal{G} \vdash_Q \alpha}{\mathcal{G} \vdash_{Q \cup Q'} \alpha} \quad \dfrac{\mathcal{G} \vdash_{Q'} \alpha'}{\mathcal{G} \vdash_{Q \cup Q'} \alpha'}}{\mathcal{G} \vdash_{Q \cup Q'} \alpha \wedge \alpha'}$$

Tying these two ideas together—that we can replace leaf annotations under certain propositional constraints, and that we can combine leaf annotations using Weaken—we get the basis for the definition of $\Phi(\mathcal{G}, \tau)$. What we want to do is build a single constraint system that will include the constraints at each leaf, but so that the constraints for one leaf do not interfere with the constraints for another. For example, if we require x to be in an annotation P at one leaf, then x needs to be in the annotation for the conclusion of the derivation—but we do not want to require x to be in the annotation for all of the other leaves, since that may severely limit our choice of annotations for those leaves. To prevent interference, the variables in formula generated by $\Phi(\mathcal{G}, \mathrm{T}_1(G))$ and $\Phi(\mathcal{G}, \mathrm{T}_2(G))$ are marked with subscripts $G, 1$ and $G, 2$ to distinguish them from variables for any other SCG/anchor-type pair (we identify the leaves of a derivation by SCG and anchor-type). To ensure that any solution to the system is a union of solutions to the leaf constraints, we include for each variable $x_{G,i}$ the constraint $x_{G,i} \Rightarrow x$.

Constraints characterizing thread-preservers are given by $\Theta_{H,i}$. The subscript H, i is used to mark the variables in $\Theta_{H,i}$ as belonging to the appropriate leaf constraints.

Finally, we use *control variables* $\kappa_{G,i}$ to deal with subsumption. We want any solution to $\Phi(\mathcal{G}, \tau)$ to include all of the anchors in τ, but as explained earlier, we do not need to include every *way* that a given anchor is included in an anchor tree. The control variables are used to turn leaf constraints on or off. At each branch of an anchor tree, we require that the control variable for at least one of the appearances of an anchor is included in the solution.

The entire constraint system has a particularly useful form.

Proposition 39. For all $\mathcal{G}$ and P, $\Phi(\mathcal{G}, P)$ is a dual-horn formula.

As with All-Termination(SCT), this means that we can use an algorithm that enumerates the minimal satisfying assignments of dual-horn formulae as a back-end for All-Termination(SCP). We present algorithms for dual-horn minimization in Section 6.

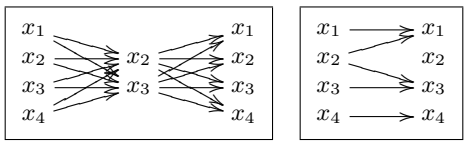
Having shown that All-Termination(SCP) is reducible to dual-horn minimization, we now show that the converse holds as well. Since (under standard complexity assumptions) there is no output-polynomial algorithm for enumerating all the minimal assignments of a dual-horn formula [11], this implies that there is also no output-polynomial algorithm for All-Termination(SCP).

Theorem 40. For every satisfiable dual-horn formula φ, there exists an ACG $\mathcal{G}$ and anchor tree τ such that

$$\{A \ : \ A \models \varphi\} = \textsc{All-Termination}(SCP)$$

assuming that InstrSCP uses the aggressive anchor removal strategy of polynomial size-change analysis.

Proof. We cannot give the full proof here, as it is fairly intricate. Instead, we sketch the main idea. A dual-horn formula φ can be partitioned into two sets of clauses: those with no negated variables (*positive clauses*), and those with one negated variable (*implication clauses*). Roughly speaking, we introduce one size-change graph for each implication clause, and two for each positive clause, in such a way that, for each graph or graph pair, the requirements for a type-2 anchor are satisfied with annotation P iff P is a satisfying assignment for the corresponding clause. Suppose the variables appearing in φ are $x_1, \ldots, x_4$. We show the size-change graphs for two example clauses below.



$(x_2 \vee x_3)$ $(\overline{x_2} \vee x_1 \vee x_3)$

In these examples, all edges are labeled by $>$. Since for any derivation $\mathcal{G} \vdash_P \mathrm{T}_2(G)$, we have $P \cap \mathrm{src}(G)$, if either of the two size-change graphs for $(x_2 \vee x_3)$ is used as a type-2 anchor, then x_2 or x_3 must be in the corresponding annotation. Likewise, for $(\overline{x_2} \vee x_1 \vee x_3)$, the requirement that annotations for type-2 anchors be thread-preservers means that if x_2 is in the annotation, either x_1 or x_3 must be as well.

Note that the examples above make use of only the type-2 anchor conditions. The challenge in constructing an adequate anchor tree for φ is in ruling out the other possibilities—type-1 anchors as well as the transposition of either anchor type. If we do not rule out these other possibilities, the full equality stated by the theorem might fail to hold, and in particular, $\textsc{All-Termination}(SCP)$ might find spurious minimal assignments. To give a sense for how to rule out anchor conditions, consider for instance that type-1 anchors require the MTP to have strict fan-in. Thus, we need only arrange that the MTP of the ACG we construct does not. Ruling out all of the undesirable anchor conditions, while retaining the use of type-2 anchors as in the examples above, is more difficult. □

6 Dual-horn minimization

In this section, we present two algorithms for enumerating the minimal satisfying assignments (hereafter: *solutions*) of a dual-horn formula.

It is well-known that dual-horn satisfiability can be decided in linear time. Let DHSat denote this decision procedure. In Figure 3, we give a procedure FindMin which takes a set of variables $\mathcal{X}$ and a (satisfiable) dual horn formula φ and returns an element of

$$\min\{A \subseteq \mathcal{X} \ : \ A \models \varphi\}$$

Hence, FindMin locates a solution minimal *with respect to* $\mathcal{X}$. This property is important because the formulas produced by the All-Termination front-ends make use of variables that do not correspond to formal arguments in the original programs. The complexity of FindMin is $O(n^2)$, where n is the size of the input ($\mathcal{X}$ and φ). An immediate corollary is that we can find an element of $\textsc{All-Termination}(SCP)$ in no more time than it takes to run polynomial size-change analysis, whose complexity is $O(n^3)$.

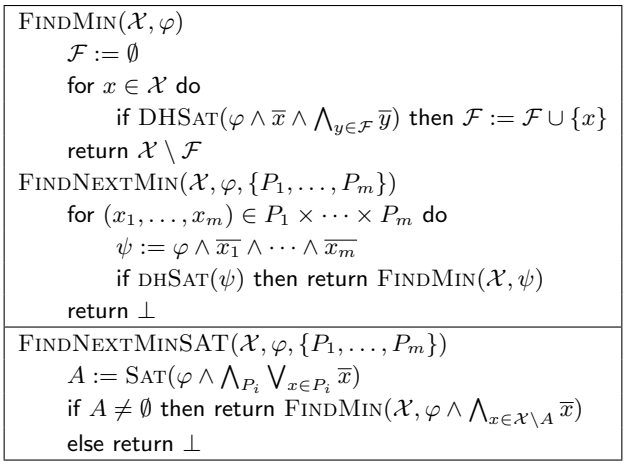


Figure 3: Dual-horn minimization: standalone and SAT-based

If we want to find additional minimal solutions, we can do so, but each one comes at increasing asymptotic cost. The procedure FindNextMin in Figure 3 takes a set of variables $\mathcal{X}$, a dual-horn formula φ, and a set of minimal solutions $P_1, \ldots, P_m$ to φ, and returns a minimal solution to φ different from any P_i. It is based on the observation that if P is a minimal solution of φ, and Q is a solution to φ such that $P \not\subseteq Q$ (so there exists an $x \in P$ such that $x \notin Q$), then there is some minimal solution $R \subseteq Q$ that is different from P. We check the existence of such a Q with DHSat, then find a minimal $R \subseteq Q$ with FindMin. The complexity of FindNextMin is $O(n^{m+1})$. Hence, generating additional minimal assignments becomes progressively more expensive, but we can obtain any constant number of minimal assignments in polynomial time. An approximation algorithm for vertex cover can be used to potentially reduce the number of case we currently consider in the loop of FindNextMin [3].

Finally, we introduce FindNextMinSAT, which uses a SAT solver to check for the existence of additional minimal elements. This has the advantage that we can exploit the recent advances in SAT-solving technology, but the disadvantage is that SAT solving is an NP-complete problem. Note that the queries given to the SAT routine through consecutive calls to FindNextMinSAT grow monotonically, each one including the last one as a subformula. Hence, an incremental SAT-solver should be used, which we expect will lead to significant performance benefits.

7 Related work

The termination problem dates back to Turing, who called it the "Printing Problem" [22], and there has been steady interest in termination ever since. Here we can only briefly touch upon the work most directly related to ours.

The strong relationship between termination and both recursion and induction was developed in the context of automated theorem proving by Boyer and Moore [4]. They observed that termination can be used to justify 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 [14]. Also, the idea of All-Termination can be traced back to Boyer and Moore [4]. However, the

approach they used to find measurable subsets just iterates over their termination analysis in the naive way, hence it has exponential complexity and has essentially nothing in common with the work presented here.

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 [1, 12, 20, 5]. 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 than that, dealing only with systems whose behavior is linear [19, 21], although there are extensions to programs with polynomial behavior [9]. Also, abstraction-refinement has been applied to termination analysis [7], and these methods have been applied to find termination bugs in device drivers [8].

Termination analysis has also been applied to functional programming languages. For example, CCG (calling context graph) analysis [18] has been implemented in ACL2s [10]; it was applied to the ACL2 regression suite, which has over 10,000 function definitions, and automatically detected termination over 98% of the time. The idea here is to transform a program into what might be called termination skeletons that terminate iff the original program does. These skeletons are produced by simplifying the input program in various ways using a combination of static analysis and theorem proving. Even though they are equi-terminating, skeletons tend to be very different from and simpler than the program that gave rise to them, *e.g.*, they may have different control-flow. The skeletons are statically annotated with various measures and analyzed with various termination analyses, including the size change method. A version of this analysis that also produces certificates has been implemented for Isabelle [15].

Recently, the problem of conditional termination was been studied [6]. Whereas we are interested in how we can add behaviors to programs while maintaining termination, they are interested in how to remove behaviors to ensure termination. This leads to the obvious question: what about All-Conditional-Termination(T)? Similarly, the non-termination problem [13] gives rise to the All-Non-Termination(T) problem.

8 Conclusions and Future Work

We introduced the All-Termination(T) problem and analyzed the complexity when T ranges over the general size-change (SCT) and polynomial size-change (SCP) analyses. We showed that All-Termination(SCT) is a PSpace-complete problem, and that no output-polynomial algorithm exists for All-Termination(SCP). We also introduced several algorithms for solving SCT and SCP. The algorithms have the property that they impose no overhead on solving the basic termination problem; they can be used to generate a subset of all possible reasons (up to a user provided bound); and some of them exploit the power of modern SAT-solving technology.

We identify several promising directions for future work. On the theoretical side, this includes analyzing the All-Termination(T) problem for the many other termination analyses that have been developed. Also worth considering the are All-Conditional-Termination(T) and All-Non-Termination(T) problems. Another idea is to combine All-Termination(T) and All-Non-Termination(T) analyses to solve the All-Termination problem by under and over approximating. Here it may help to use several different (non)termination solvers. One can also consider reasons that are more expressive than sets of formals, *e.g.*, reasons might be formulas over various logics. Also, induction schemes often depend on the combination of functions appearing in a theorem, which leads

to the question of whether using ALL-TERMINATION(T)-inspired analyses will enable us to automatically generate more powerful induction schemes. On the engineering side, the obvious challenge is to build and evaluate useful tools, *e.g.*, by incorporating these ideas into theorem proving systems.

A Proof of Theorem 36

To prove that *SCP* is a valid termination analysis, the most important step is proving the soundness of $\vdash$ with respect to $\models$. We need several easy lemmas, stated here without proof. We write $\alpha \restriction P$ for α with each size change graph G replaced by $G \restriction P$.

Lemma 41. If $Q \subseteq \mathcal{P}$ then $\mathrm{TP}(\mathcal{G} \restriction Q) = \mathrm{TP}(\mathcal{G}) \cap 2^Q$.

Proof. Let $P \in \mathrm{TP}(\mathcal{G}) \cap 2^Q$. Suppose $f \xrightarrow{G \restriction Q} g \in \mathcal{G} \restriction Q$ and $x \in \mathrm{src}(G) \cap P$. Then $x \in \mathrm{src}(G)$ and $f \xrightarrow{G} g \in \mathcal{G}$. But $P \in \mathrm{TP}(\mathcal{G})$, so there exists some $x \xrightarrow{r} y \in G$ such that $y \in P$. Since $P \subseteq Q$, it follows that $x \xrightarrow{r} y \in G \restriction Q$. Hence $P \in \mathrm{TP}(\mathcal{G} \restriction Q)$.

The argument in the other direction is similar. □

Lemma 42. If $\mathcal{G}$ has strict fan-in then $\mathcal{G} \restriction P$ has strict fan-in.

Proof. Trivial: if $H \in \mathcal{G} \restriction P$ then $H = G \restriction P$ for some $G \in \mathcal{G}$. Since G has strict fan-in and $H \subseteq G$, it follows that H has strict fan-in as well. □

Lemma 43. $\mathrm{ND}(\mathcal{G}) \supseteq \mathrm{ND}(\mathcal{G} \restriction P)$.

Proof. Immediate consequence of Proposition 20. □

Lemma 44. If $\mathcal{G} \restriction P \models \alpha \restriction P$ then $\mathcal{G} \models \alpha$.

Proof. Easy induction on the structure of α, using the fact that if $Q \subseteq P$, then $(\mathcal{G} \restriction P) \restriction Q = \mathcal{G} \restriction Q$. □

The next lemma is the heart of the soundness argument: it shows the soundness of $\vdash$ for anchor formulas. The soundness theorem following it follows easily by induction on derivations.

Lemma 45. If $\mathcal{G} \vdash_P \alpha$ then $\mathcal{G} \restriction P \models \alpha \restriction P$.

Proof. Induction on derivations. We give the three interesting cases.

Case TY1: By assumption, $G \in \mathcal{G}$, $\mathrm{MTP}(\mathcal{G})$ has strict fan-in, $P \in \mathrm{TP}(\mathcal{G})$, and $\exists x \xrightarrow{\geq} y \in G \restriction P$. By Lemma 42, it follows that $\mathcal{G} \restriction P$ has strict fan-in. By Lemma 41 we have $P \in \mathrm{TP}(\mathcal{G} \restriction P)$.

Case TY2: By assumption, $G \in \mathcal{G}$, $P \cap G \neq \emptyset$ and $P \in \mathrm{TP}(\mathcal{G} \rhd G)$. By Lemma 41, $P \in \mathrm{TP}((\mathcal{G} \rhd G) \restriction P)$. But we have

$$\begin{aligned}(\mathcal{G} \rhd G) \restriction P &= (\mathcal{G} - G + (G - \mathrm{ND}(\mathcal{G}))) \restriction P \\ &= \mathcal{G} \restriction P - G \restriction P + (G - \mathrm{ND}(\mathcal{G})) \restriction P \\ &= \mathcal{G} \restriction P - G \restriction P + (G \restriction P - \mathrm{ND}(\mathcal{G}))\end{aligned}$$

Hence if $H \in \mathcal{G} \restriction P - G \restriction P + (G \restriction P - \mathrm{ND}(\mathcal{G}))$ then $P \in \mathrm{TP}(H)$.

Now suppose $H \in \mathcal{G} \restriction P \rhd G \restriction P$ and $x \in \mathrm{src}(H) \cap P$. By definition, $H \in \mathcal{G} \restriction P - G \restriction P + (G \restriction P - \mathrm{ND}(\mathcal{G} \restriction P))$. If $H \in \mathcal{G} \restriction P - G \restriction P$, then $P \in \mathrm{TP}(H)$ as concluded above. Otherwise, $H = G \restriction P - \mathrm{ND}(\mathcal{G} \restriction P)$. By Lemma 43, $\mathrm{ND}(\mathcal{G} \restriction P) \subseteq \mathrm{ND}(\mathcal{G})$. Hence

$$G \restriction P - \mathrm{ND}(\mathcal{G} \restriction P) \supseteq G \restriction P - \mathrm{ND}(\mathcal{G})$$

Since $P \in \mathrm{TP}(G \restriction P - \mathrm{ND}(\mathcal{G}))$ as above, we have $P \in \mathrm{TP}(G \restriction P - \mathrm{ND}(\mathcal{G} \restriction P))$.

Therefore, $P \in \mathrm{TP}(\mathcal{G} \restriction P \rhd G \restriction P)$, which together with the initial assumptions shows that $\mathcal{G} \restriction P \models \mathrm{T}_2(G \restriction P)$.

Case WEAKEN: By assumption, $\mathcal{G} \vdash_Q \alpha$ for some $Q \subseteq P$. By induction, $\mathcal{G} \restriction Q \models \alpha \restriction Q$. But $\mathcal{G} \restriction Q = (\mathcal{G} \restriction P) \restriction Q$, so it follows that $(\mathcal{G} \restriction P) \restriction Q \models (\alpha \restriction P) \restriction Q$. Using Lemma 44, we conclude that $\mathcal{G} \restriction P \models \alpha \restriction P$. □

Theorem (36)**.** If $\mathcal{G} \vdash_P \tau$ then $\mathcal{G} \restriction P \models \tau \restriction P$.

Proof. Trivial induction on derivations, using Lemma 45. □

B Proof of Theorem 38

We define a "renaming" function ρ which adds subscripts to variables in a variable assignment A, as follows:

$$\rho_{H,i}(A) = \{x_{H,i} \ : \ x \in A\}$$

We also define $\mathrm{vars}(\varphi)$ to be the set of variables occurring in φ.

Lemma 46. $\rho_{H,i}(A) \models \Theta_{H,i}(G)$ iff $A \in \mathrm{TP}(G)$.

Proof. Straightforward unrolling of definitions. □

Lemma 47. $\rho_{H,i}(A) \models \Theta_{H,i}(\mathcal{G})$ iff $A \in \mathrm{TP}(\mathcal{G})$.

Proof. Immediate corollary of Lemma 46 □

Lemma 48. $A \models \varphi$ iff $A \cap \mathrm{vars}(\varphi) \models \varphi$.

Proof. Simple unrolling of the definitions. □

Lemma 49.

$$\mathrm{vars}(\Phi(\mathcal{G}, \alpha)) \subseteq \begin{cases} & \mathcal{P} \cup K(\alpha) \\ \cup & \{x_{G,1} \ : \ x \in \mathcal{P}, \kappa_{G_1} \in K(\alpha)\} \\ \cup & \{xy_{G,1} \ : \ x, y \in \mathcal{P}, \kappa_{G,1} \in K(\alpha)\} \\ \cup & \{x_{G,2} \ : \ x \in \mathcal{P}, \kappa_{G,2} \in K(\alpha)\} \end{cases}$$

Proof. Easy induction on the structure of α. □

Corollary 50. $\kappa_{G,i} \in \mathrm{vars}(\Phi(\mathcal{G}, \alpha))$ iff $\kappa_{G,i} \in K(\alpha)$.

Corollary 51. If $K(\alpha_1) \cap K(\alpha_2) = \emptyset$ then for any $\mathcal{G}$,

$$\mathrm{vars}(\Phi(\mathcal{G}, \alpha_1)) \cap \mathrm{vars}(\Phi(\mathcal{G}, \alpha_2)) \subseteq \mathcal{P}$$

Lemma 52. If $A_1 \models \Phi(\mathcal{G}, \alpha_1)$ and $A_2 \models \Phi(\mathcal{G}, \alpha_2)$ with $K(\alpha_1) \cap K(\alpha_2) = \emptyset$, then $A_1 \cup A_2 \models \Phi(\mathcal{G}, \alpha_1)$ and $A_1 \cup A_2 \models \Phi(\mathcal{G}, \alpha_2)$.

Proof. By Corollary 51, $\mathrm{vars}(\Phi(\mathcal{G},\alpha_1)) \cap \mathrm{vars}(\Phi(\mathcal{G},\alpha_2)) \subseteq \mathcal{P}$. But variables from $\mathcal{P}$ appear only in the positive phase in $\Phi(\mathcal{G},\alpha_1)$ and $\Phi(\mathcal{G},\alpha_2)$. Let $C \in \Phi(\mathcal{G},\alpha_1)$ be a clause. Then there is some $\lambda \in C$ such that $A_1 \models \lambda$. If λ appears in the positive phase, then $A_1 \cup A_2 \models \lambda$. Otherwise, $\lambda = \overline{x}$ for some variable $x \notin \mathrm{vars}(\Phi(\mathcal{G},\alpha_2))$. Hence here too $A_1 \cup A_2 \models \lambda$. It follows that $A_1 \cup A_2 \models C$ for all $C \in \Phi(\mathcal{G},\alpha_1)$, and therefore that $A_1 \cup A_2 \models \Phi(\mathcal{G},\alpha_1)$. The argument that $A_1 \cup A_2 \models \Phi(\mathcal{G},\alpha_2)$ is symmetric. □

Lemma 53. If $\langle \alpha_1\tau_1,\ \ldots,\ \alpha_n\tau_n\rangle$ is well-formed,
$A_i \models \Phi(\mathcal{G},\alpha_i)$,
$B_i \models \Phi(\mathcal{G} - \mathrm{anchors}(\alpha_i),\tau_i)$,
$A = \bigcup_{1\le i\le n} A_i \cup B_i$
then, for all $1 \le i \le n$,

$$A \models \Phi(\mathcal{G},\alpha_i) \text{ and } A \models \Phi(\mathcal{G} - \mathrm{anchors}(\alpha_i),\tau_i)$$

Proof. Since $\langle \alpha_1\tau_1,\ \ldots,\ \alpha_n\tau_n\rangle$ is well-formed, we have by Corollary 51 that

$$\mathrm{vars}(\Phi(\mathcal{G},\alpha_i)) \cap \mathrm{vars}(\Phi(\mathcal{G} - \mathrm{anchors}(\alpha_i),\tau_i)) \subseteq \mathcal{P}$$

for all $1 \le i \le n$. By an argument similar to that of Lemma 52, it follows that

$$A_i \cup B_i \models \Phi(\mathcal{G},\alpha_i) \text{ and } A_i \cup B_i \models \Phi(\mathcal{G} - \mathrm{anchors}(\alpha_i),\tau_i)$$

for all $1 \le i \le n$. Now let $1 \le i < j \le n$ and

$$\phi_i = \Phi(\mathcal{G},\alpha_i) \cup \Phi(\mathcal{G} - \mathrm{anchors}(\alpha_i),\tau_i),$$
$$\phi_j = \Phi(\mathcal{G},\alpha_j) \cup \Phi(\mathcal{G} - \mathrm{anchors}(\alpha_j),\tau_j)$$

By the well-formedness of $\langle \alpha_1\tau_1,\ \ldots,\ \alpha_n\tau_n\rangle$, we have by Corollary 51 that $\mathrm{vars}(\phi_i) \cap \mathrm{vars}(\phi_j) \subseteq \mathcal{P}$. Running again the argument from Lemma 52, we have $A \models \Phi(\mathcal{G},\alpha_i)$ and $A \models \Phi(\mathcal{G} - \mathrm{anchors}(\alpha_i),\tau_i)$. □

Lemma 54. If $\mathcal{G} \vdash \alpha$, α is well-formed, $\alpha' \le \alpha$, and $P \subseteq \mathcal{P}(\mathcal{G})$, then

$$\mathcal{G} \vdash_P \alpha' \iff \exists A\ .\ \begin{cases} A \models \Phi(\mathcal{G},\alpha), \\ A \subseteq \mathrm{vars}(\Phi(\mathcal{G},\alpha)) \cup \mathcal{P}(\mathcal{G}), \\ A \cap \mathcal{P} = P, \\ A \cap K(\alpha) = K(\alpha') \end{cases}$$

Proof. We first prove the left-to-right direction, by induction on the derivation of $\mathcal{G} \vdash_P \alpha'$.

Case TRUE: Trivial.

Case TY1: By assumption, $\alpha' = \mathrm{T}_1(G)$ for some $G \in \mathcal{G}$, with $P \in \mathrm{TP}(\mathcal{G})$ and $\exists x \xrightarrow{\ge} y \in G \restriction P$. By Lemma 47, $\rho_{G,1}(P) \models \Theta_{G,1}(G)$. Let $A = \rho_{G,1}(P) \cup P \cup \{\kappa_{G,1}, xy_{G,1}\}$. Then $A \models \Phi(\mathcal{G},\alpha')$, and $A \cap \mathcal{P} = P$. But since $\alpha' \le \alpha$ and $\alpha' = \mathrm{T}_1(G)$, we have $\alpha = \alpha'$. Hence $A \models \Phi(\mathcal{G},\alpha)$. Note that $K(\alpha') = K(\alpha) = \{\kappa_{G,1}\}$, so $A \cap K(\alpha) = K(\alpha')$ as required. Finally, observe that $A \subseteq \mathrm{vars}(\Phi(\mathcal{G},\alpha)) \cup \mathcal{P}(\mathcal{G})$.

Case TY2: Similar to TY1.

Case CONJ: By assumption, $\alpha' = \alpha'_1 \wedge \alpha'_2$ for some α'_1, α'_2 such that $\mathcal{G} \vdash_P \alpha'_1$ and $\mathcal{G} \vdash_P \alpha'_2$. Since $\alpha' \le \alpha$, it follows that $\alpha = \alpha_1 \wedge \alpha_2$ for some α_1, α_2 such that $\alpha'_1 \le \alpha_1$ and $\alpha'_2 \le \alpha_2$. By induction, there exist A_1, A_2 such that $A_i \models \Phi(\mathcal{G},\alpha_i)$, $A_i \cap \mathcal{P} = P$, $A_i \cap K(\alpha_i) = K(\alpha'_i)$, and $A_i \subseteq \mathrm{vars}(\Phi(\mathcal{G},\alpha_i)) \cup \mathcal{P}(\mathcal{G})$. for $i \in \{1,2\}$. Since $\alpha_1 \wedge \alpha_2$ is well-formed,

anchors(α_1) $\cap$ anchors(α_2) $= \emptyset$. Using Lemma 52, it follows that $A_1 \cup A_2 \models \Phi(\mathcal{G}, \alpha_1 \wedge \alpha_2)$, and clearly $(A_1 \cup A_2) \cap \mathcal{P} = (A_1 \cap \mathcal{P}) \cup (A_2 \cap \mathcal{P}) = P$. Using Lemma 50, we can observe that $A_i \cap (K(\alpha_1) \cup K(\alpha_2)) = A_i \cap K(\alpha_i)$ for $i \in \{1, 2\}$. Therefore,

$$\begin{aligned}
& (A_1 \cup A_2) \cap K(\alpha_1 \wedge \alpha_2) \\
= \; & (A_1 \cup A_2) \cap (K(\alpha_1) \cup K(\alpha_2)) \\
= \; & (A_1 \cap (K(\alpha_1) \cup K(\alpha_2))) \cup (A_2 \cap (K(\alpha_1) \cup K(\alpha_2))) \\
= \; & (A_1 \cap K(\alpha_1)) \cup (A_2 \cap K(\alpha_2)) \\
= \; & K(\alpha_1') \cup K(\alpha_2') \\
= \; & K(\alpha_1' \wedge \alpha_2')
\end{aligned}$$

Finally, notice that

$$\begin{aligned}
A_1 \cup A_2 \; &\subseteq \; \mathrm{vars}(\Phi(\mathcal{G}, \alpha_1)) \cup \mathrm{vars}(\Phi(\mathcal{G}, \alpha_1)) \cup \mathcal{P}(\mathcal{G}) \\
&= \; \mathrm{vars}(\Phi(\mathcal{G}, \alpha_1 \wedge \alpha_2)) \cup \mathcal{P}(\mathcal{G})
\end{aligned}$$

Case Trans: Immediate by induction.

Case Weaken: By assumption, $\mathcal{G} \vdash_Q \alpha'$ for some $Q \subseteq P$. By induction, there exists an A such that $A \models \Phi(\mathcal{G}, \alpha)$, $A \subseteq \mathrm{vars}(\Phi(\mathcal{G}, \alpha)) \cup \mathcal{P}(\mathcal{G})$, $A \cap \mathcal{P} = Q$, and $A \cap K(\alpha) = K(\alpha')$. Recall that $P \subseteq \mathcal{P}(\mathcal{G})$. Hence $A \cup P \subseteq \mathrm{vars}(\Phi(\mathcal{G}, \alpha)) \cup \mathcal{P}(\mathcal{G})$. Since the variables in $P \setminus Q$ are in $\mathcal{P}$, they only appear positively in $\Phi(\mathcal{G}, \alpha)$. Hence, $A \cup P \models \Phi(\mathcal{G}, \alpha)$. Finally, $(A \cup P) \cap \mathcal{P} = P$ and $(A \cup P) \cap K(\alpha) = A \cap K(\alpha) = K(\alpha')$.

For the right-to-left direction, we prove by induction on the structure of α'.

Case: $\boxed{\top}$ Trivial.

Case: $\boxed{\mathrm{T}_1(G)}$ Since $\mathrm{T}_1(G) \leq \alpha$, it must be that $\alpha = \mathrm{T}_1(G)$. By assumption, $\mathcal{G} \vdash \mathrm{T}_1(G)$, so in particular $G \in \mathcal{G}$ and $\mathcal{G} \upharpoonright \mathrm{MTP}(\mathcal{G})$ has strict fan-in. Also by assumption, $A \models \Phi(\mathcal{G}, \mathrm{T}_1(G))$ and $\kappa_{G,1} \in K(\mathrm{T}_1(G)) \subseteq A$. Hence $A \models \Theta_{G,1}(\mathcal{G})$. Let $Q \subseteq \mathcal{P}$ such that $\rho_{G,1}(Q) = A \cap \mathrm{vars}(\Theta_{G,1}(\mathcal{G}))$. Note that we have $Q \subseteq A$ and hence $Q \subseteq P$. By Lemma 48, $\rho_{G,1}(Q) \models \Theta_{G,1}(\mathcal{G})$. By Lemma 47, $Q \in \mathrm{TP}(\mathcal{G})$. Since $\kappa_{G,1} \in A$, we have that $A \models \bigvee_{x \xrightarrow{\geq} y \in G} xy_{G,1}$. Let $xy_{G,1} \in A$ such that $x \xrightarrow{\geq} y \in G$. Since $A \models xy_{G,1} \Rightarrow x_{G_1}$ and $A \models xy_{G,1} \Rightarrow y_{G_1}$, it follows that $x_{G,1}$ and $y_{G,1}$ are in A; hence x and y are in Q. We may thus apply Ty1 to show $\mathcal{G} \vdash_Q \mathrm{T}_1(G)$. Since $Q \subseteq P$, we can then apply Weaken to show $\mathcal{G} \vdash_P \mathrm{T}_1(G)$.

Case: $\boxed{\mathrm{T}_2(G)}$ Similar to $\mathrm{T}_1(G)$.

Case: $\boxed{\alpha_1' \wedge \alpha_2'}$ It must be that $\alpha = \alpha_1 \wedge \alpha_2$ such that $\alpha_1' \leq \alpha_1$ and $\alpha_2' \leq \alpha_2$. By assumption, $A \models \Phi(\mathcal{G}, \alpha_1 \wedge \alpha_2)$, so $A \models \Phi(\mathcal{G}, \alpha_1)$ and $A \models \Phi(\mathcal{G}, \alpha_2)$. Let $A_1 = A \cap \mathrm{vars}(\Phi(\mathcal{G}, \alpha_1))$ and $A_2 = A \cap \mathrm{vars}(\Phi(\mathcal{G}, \alpha_2))$. By Lemma 48, $A_1 \models \Phi(\mathcal{G}, \alpha_1)$ and $A_2 \models \Phi(\mathcal{G}, \alpha_2)$. We have that $A \cap K(\alpha_1 \wedge \alpha_2) = K(\alpha_1' \wedge \alpha_2')$. Hence $(A \cap K(\alpha_1)) \cup (A \cap K(\alpha_2)) = K(\alpha_1') \cup K(\alpha_2')$. Since $K(\alpha_1') \subseteq K(\alpha_1)$ and $K(\alpha_2') \subseteq K(\alpha_2)$, it follows that $A \cap K(\alpha_1) = K(\alpha_1')$ and $A \cap K(\alpha_2) = K(\alpha_2')$. But $K(\alpha_1) \subseteq \mathrm{vars}(\Phi(\mathcal{G}, \alpha_1))$, so $A_1 \cap K(\alpha_1) = K(\alpha_1')$ and likewise for A_2. Thus we can apply induction twice, using A_1 with α_1, α_1' and A_2 with α_2, α_2', to conclude that $\mathcal{G} \vdash_P \alpha_1'$ and $\mathcal{G} \vdash_P \alpha_2'$. Applying Conj, we have $\mathcal{G} \vdash_P \alpha_1' \wedge \alpha_2'$.

Case: $\boxed{(\alpha')^t}$ Immediate by induction. □

Lemma 55. If $J \subseteq K(\alpha)$ then $\alpha \restriction J \leq \alpha$ and $K(\alpha \restriction J) = J$.

Lemma 56. If $\alpha' \leq \alpha$ then $\text{anchors}(\alpha') = \text{anchors}(\alpha)$ iff for all $G \in \text{anchors}(\alpha)$, $K(G) \cap K(\alpha') \cap K(\alpha) \neq \emptyset$.

Lemma 57. If $\mathcal{G} \vdash \tau$, τ is well-formed, and $P \subseteq \mathcal{P}$, then

$$\exists \tau' \leq \tau \;.\; \mathcal{G} \vdash_P \tau' \iff \exists A \;.\; \begin{cases} A \models \Phi(\mathcal{G}, \tau), \\ A \subseteq \text{vars}(\Phi(\mathcal{G}, \tau)) \cup \mathcal{P}(\mathcal{G}), \\ A \cap \mathcal{P} = P \end{cases}$$

Proof. By induction on the structure of τ.

Let $\tau = \langle \alpha_1 \tau_1, \ \ldots, \ \alpha_n \tau_n \rangle$. Since we assume that $\mathcal{G} \vdash \tau$, we have in particular that $\text{SCC}(\mathcal{G}) = \{\mathcal{G}_1, \ \ldots, \ \mathcal{G}_n\}$, and for each $\mathcal{G}_i$, $\mathcal{G}_i \vdash \alpha_i$ and $\mathcal{G}_i - \text{anchors}(\alpha_i) \vdash \tau_i$.

We first prove the right-to-left direction. Suppose $A \models \Phi(\mathcal{G}, \tau)$ and $A \cap \mathcal{P} = P$. Fix an i such that $1 \leq i \leq n$. By assumption, $A \models \Phi(\mathcal{G}, \alpha_i \tau_i)$. Hence,

(1) $\forall G \in \text{anchors}(\alpha_i) \;.\; A \models \bigvee_{\kappa \in K(G) \cap K(\alpha_i)} \kappa$

(2) $A \models \Phi(\mathcal{G}, \alpha_i)$

(3) $A \models \Phi(\mathcal{G} - \text{anchors}(\alpha_i), \tau_i)$

Let $J = A \cap K(\alpha_i)$. By Lemma 55, $\alpha_i \restriction J \leq \alpha_i$ and $K(\alpha_i \restriction J) = J$. Using (1), for all $G \in \text{anchors}(\alpha_i)$, we have $A \cap K(G) \cap K(\alpha_i) \neq \emptyset$. But

$$\begin{aligned} A \cap K(G) \cap K(\alpha_i) &= J \cap K(G) \cap K(\alpha_i) \\ &= K(\alpha_i \restriction J) \cap K(G) \cap K(\alpha_i) \end{aligned}$$

Hence, by Lemma 56, we have $\text{anchors}(\alpha_i \restriction J) = \text{anchors}(\alpha_i)$. Using (2) and Lemma 54, $\mathcal{G}_i \vdash_P \alpha_i \restriction J$. Using (3), by induction, there exists a $\tau_i' \leq \tau_i$ such that $\mathcal{G}_i - \text{anchors}(\alpha_i) \vdash_P \tau_i'$. Thus,

$$\langle (\alpha_1 \restriction J)\tau_1', \ \ldots, \ (\alpha_n \restriction J)\tau_n' \rangle \leq \langle \alpha_1 \tau_1, \ \ldots, \ \alpha_n \tau_n \rangle$$

and

$$\mathcal{G} \vdash_P \langle (\alpha_1 \restriction J)\tau_1', \ \ldots, \ (\alpha_n \restriction J)\tau_n' \rangle$$

For the left-to-right direction, suppose $\tau' \leq \tau$ such that $\mathcal{G} \vdash_P \tau'$. This means in particular that $\tau' = \langle \alpha_1' \tau_1', \ \ldots, \ \alpha_n' \tau_n' \rangle$ with $\alpha_i' \leq \alpha_i$, $\tau_i' \leq \tau_i$, and $\text{anchors}(\alpha_i') = \text{anchors}(\alpha_i)$ for all $1 \leq i \leq n$. Fix an i with $1 \leq i \leq n$. By Lemma 56, for all $G \in \text{anchors}(\alpha_i)$, we have $K(G) \cap K(\alpha_i) \cap K(\alpha_i') \neq \emptyset$. By Lemma 54, there is some A_i such that $A_i \models \Phi(\mathcal{G}_i, \alpha_i)$, $A_i \cap \mathcal{P} = P$, and $A_i \cap K(\alpha_i) = K(\alpha_i')$. Hence,

$$A_i \models \bigwedge_{G \in \text{anchors}(\alpha)} \bigvee_{\kappa \in K(G) \cap K(\alpha)} \kappa$$

We also have, by induction, that there is some B_i with $B_i \models \Phi(\mathcal{G}_i - \text{anchors}(\alpha_i), \tau_i)$ and $B_i \cap \mathcal{P} = P$. Let $A = \bigcup_{1 \leq i \leq n} A_i \cup B_i$. Finally, using Lemma 53, we conclude $A \models \Phi(\mathcal{G}, \tau)$ and $A \cap \mathcal{P} = P$. □

Theorem (38)**.** If $\textsc{InstrSCP}(\text{analyze}(F)) = \tau$ then

$$\{P \ : \ \mathit{SCP}(F, P)\} = \{A \cap \mathcal{P} \ : \ A \models \Phi(\mathcal{G}, \tau)\}$$

Proof. Using Lemmas 48 and 57, we have

$$\{P \ : \ \mathcal{G} \vdash_P \tau', \ \tau' \leq \tau\} = \{A \cap \mathcal{P} \ : \ A \models \Phi(\mathcal{G}, \tau)\}$$

under the assumption $\mathcal{G} \vdash \tau$. The result follows trivially. □

References

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[2] A. M. Ben-Amram and C. S. Lee. Program termination analysis in polynomial time. *ACM Trans. Program. Lang. Syst.*, 29(1):5, 2007.

[3] R. Ben-Eliyahu and R. Dechter. On computing minimal models. *Annals of Mathematics and Artificial Intelligence*, 18:3–27, 1996.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[5] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[6] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *Computer Aided Verification*, LNCS. Springer, 2008.

[7] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *Static Analysis: 12th International Symposium, SAS 2005*, volume 3672 of *LNCS*, pages 87–102, September 2005.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *ACM Conference on Programming Language Design and Implementation*, pages 415–426. ACM Press, 2006.

[9] P. Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *LNCS*, pages 1–24. Springer, 2005.

[10] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore. ACL2s: The ACL2 Sedan. *Electr. Notes Theor. Comput. Sci.*, 174(2):3–18, 2007.

[11] E. C. S. Dimitris J. Kavvadias, Martha Sideri. On the maximal models of horn formulas. Technical report, Computer Technology Inst., 2002.

[12] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA*, volume 3091 of *LNCS*, pages 210–220. Springer–Verlag, 2004.

[13] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *ACM Symposium on Principles of Programming Languages*, pages 147–158. ACM, 2008.

[14] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[15] A. Krauss. Certified size-change termination. In F. Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 460–475. Springer, 2007.

[16] C. S. Lee. Ranking functions for size-change termination. submitted.

[17] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, pages 81–92. ACM Press, 2001.

[18] P. Manolios and D. Vroon. Termination analysis with calling context graphs. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 401–414. Springer, 2006.

[19] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.

[20] R. Thiemann and J. Giesl. Size-change termination for term rewriting. Technical Report AIB-2003-02, RWTH Aachen, Jan. 2003.

[21] A. Tiwari. Termination of linear programs. In *Computer-Aided Verification*, LNCS 3114, pages 70–82. Springer, July 2004.

[22] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42 of *Series 2*, pages 230–265, 1936.