Ranking Abstractions

Aziem Chawdhary[1], Byron Cook[2], Sumit Gulwani[2], Mooly Sagiv[3], and Hongseok Yang[1]

[1] Queen Mary, University of London
[2] Microsoft Research
[3] Tel Aviv University

Abstract. We propose an abstract interpretation algorithm for proving that a program terminates on all inputs. The algorithm uses a novel abstract domain which uses ranking relations to conservatively represent relations between intermediate program states. One of the attractive aspects of the algorithm is that it abstracts information that is usually not important for proving termination such as program invariants and yet it distinguishes between different reasons for termination which are not usually maintained in existing abstract domains. We have implemented a prototype of the algorithm and shown that in practice it is fast and precise.

1 Introduction

This paper develops sound algorithms for inferring that C programs terminate on all possible inputs. The oldest trick in the book of termination proofs for programs (e.g., [18]) is the ranking function proof. In this method, we find a function p that maps program states into a well-founded ordered set, such that $p(\sigma) > p(\sigma')$ whenever σ' is a state reachable from state σ.

Despite the enormous progress in synthesizing ranking functions (e.g., [3]), modern programming language features such as nested loops lead to non-linear behaviours which make it hard to apply existing techniques to synthesize ranking functions in a sound and precise way directly to the C code.

Recently, [17] introduced the *disjunctive well-foundedness* principle in order to split the termination argument into multiple ranking relations, corresponding to different situations in the program. The main idea is to use a finite set of ranking functions $r_1, r_2, \ldots, r_n$ each of which is well-founded, and to require in addition that the relation between any two intermediate states in the program is included in one of the relations, i.e.,

$$\tau^+ \quad \subseteq \quad \bigcup_{i=1}^{n} r_i \tag{1}$$

where τ is the transition system describing the meaning of the program and τ^+ is the non-reflexive transitive closure of τ. This principle localizes termination proofs by allowing the use of simpler ranking function synthesizers to handle more complicated termination proofs.

However, [17] leaves two open problems: (a) what is the best way to find the set of ranking functions $r_1, r_2, \ldots, r_n$ and (b) how to effectively check the condition in

Eq. 1. Notice that this is a safety question which can be attacked by any abstract interpreter [10]. However it may be expensive to check the condition by the abstract interpreter or the interpreter may fail due to imprecision.

In this paper we solve these two problems together in a novel way. The first problem is solved by developing abstract domains which are parameterized by sets of ranking functions. The meaning of each of the relations (ranking functions) overapproximates the relations between intermediate states in the program. We employ standard iterative fixpoint computations to compute a set of ranking functions or determine that the program may diverge. The ranking synthesizer is invoked with larger and larger relations obtained by composing the current approximation with every possible command. Notice that calling the ranking synthesizer allows us to abstract away information that is not necessary for termination, but maintains enough distinctions between different ranking functions. When a fixpoint is reached the condition in Eq. 1 is guaranteed to hold and thus there is no need to perform the inclusion check above. The efficiency provided by our domain is underlined by result which we prove, that, for a particular base abstract domain, fixpoint calculations are guaranteed to converge, at most, in two steps. For more refined domains we lose the guarantee of two, but in our experimental results we find that fixpoints converge in few iterations.

Related Work Program termination has been studied extensively with many impressive algorithms for automatically inferring termination for functional (e.g.,[13]), logic (e.g., [6, 4]) and imperative programs (e.g., [3, 7, 19, 1]). The result in [17] encourages the use of existing safety analyzers in order to prove termination (e.g., SLAM [7] or Octagon [2]). The point of departure of this work is to define a new abstract domain, designed with termination in mind, rather than to re-use existing domains for safety. Termination analysis requires a precise treatment of disjunction, and information about well-foundedness, and we suggest that domains which target these properties will be more appropriate for termination analysis than domains designed for wholly other purposes. Our work follows [7, 2] by employing the disjunctive well foundedness principle [17] in order to split the termination argument into multiple ranking relations corresponding to different situations in the program.

By tailoring our abstract domain to termination we obtain a very efficient termination prover for imperative programs. In particular it is faster than TERMINATOR, which relies on SLAM [7], and variance analyses based on Octagon or Polyhedra [2]. The variance analysis we describe in this paper uses rank functions natively, in contrast to the non-native variance analyses proposed in [2] which were constructed from existing domains for invariance. In contrast to [2] we directly abstract ranking relations which allow us to be more precise in the cases where the underlying abstract domain used for invariance analysis is too coarse (e.g., non-disjunctive) and our analysis can be more efficient when the underlying domain records complicated invariants that are not needed for proving termination. In contrast to [7], we iteratively compute ranking functions without the use of counterexample guided refinement.

Our abstract domain is related to the abstraction used in size-change termination [13]. In both cases, program fragments are abstracted in terms of measures decreased or preserved by the fragments. The major difference is that our domain contains only those abstract elements that mean terminating program fragments (unless the elements are $\top$), whilst size-change termination analyses can have an (non-$\top$) abstract element that denotes a diverging program fragment. As a result, size-change termination analyses have to check whether (the concretization of) an abstracted program terminates, whereas our analysis can skip this rather expensive checking.

2 Informal description of the analysis

In this section we informally describe the new analysis using an example. Later, in Section 3, we provide a more formal description.

Consider the program:

```
1    while (x>0 ∧ y>0) {
2        if (∗) then { x=x−1; y=∗; } else { y=y−1; }
3    }
```

This program illustrates the limitation of known termination analyses. The Octagon-based and Polyhedra-based termination analyses from [2] can quickly (*i.e.* in 0.02s) infer that the relation $`x \geq 0 \wedge `x \geq x$ holds between any state at ℓ=2 and any previous state at ℓ=2, where $`x$ and $x$ denote previous and current values of $x$ respectively. (Note that $`x$ is denoting *some* previous value of x, and not necessarily the *last* value). Unfortunately, this relation is insufficient to prove termination of the loop, as it is not (disjunctively) well-founded—the condition sufficient for proving termination as described in [2].

In contrast TERMINATOR can prove the example terminating, but at a great cost (16s). TERMINATOR finds the following disjunctively well-founded relation at ℓ=2:

$$(`x \geq 0 \wedge `x{-}1 \geq x) \vee (`y \geq 0 \wedge `y{-}1 \geq y)$$

To find this relation TERMINATOR performs three rounds of refinement on the relation itself and 9 rounds of abstraction/refinement for the checking of the 3 candidate assertions, resulting in the discovery of 21 transition predicates.

The termination analysis in this paper gives us TERMINATOR's accuracy at the speed of the Octagon-based termination analysis. The new analysis finds the relation

$$(`x \geq 0 \wedge `x{-}1 \geq x) \vee (`y \geq 0 \wedge `y{-}1 \geq y \wedge `x{=}x)$$

in 0.02s.

Concretely, the new analysis uses a disjunctive domain of ranking relations conjoined with the information about unchanged variables. That is: disjunctions of relations of the form $T_e \wedge V_X$, where

$$V_X \overset{\text{def}}{=} \bigwedge_{x \in X} `x{=}x, \qquad T_e \overset{\text{def}}{=} `e \geq 0 \wedge `e{-}1 \geq e,$$

and $`e$ is the expression $e$ with all variables $x$ replaced by their corresponding preprimed versions $`x$. Let R represent the transition relation of the loop body of our program in DNF:

$$R \overset{\text{def}}{=} C_1 \vee C_2,$$
$$C_1 \overset{\text{def}}{=} `x > 0 \wedge `y > 0 \wedge x{=}`x{-}1, \qquad C_2 \overset{\text{def}}{=} `x > 0 \wedge `y > 0 \wedge x{=}`x \wedge y{=}`y{-}1.$$

Our analysis begins by taking each disjunct in R and performing rank-function synthesis on it. In this case we get

$$\mathsf{RFS}(C_1) = x \quad \text{and} \quad \mathsf{RFS}(C_2) = y.$$

For each disjunct, the analysis also computes a set of variables whose values do not change. In this example, it determines that C_1 can change both x and y, but C_2 does not change variable x. Thus, we begin our analysis with the initial abstract state $A_0 \overset{\text{def}}{=} T_x \vee (T_y \wedge V_{\{x\}})$, that is,

$$A_0 \quad = \quad (\text{`}x \geq 0 \wedge \text{`}x{-}1 \geq x) \vee (\text{`}y \geq 0 \wedge \text{`}y{-}1 \geq y \wedge \text{`}x{=}x).$$

Note that A_0 overapproximates the loop body R.

The meaning of this initial abstract state (*i.e.* $\gamma(A_0)$) is set of all finite sequences of program states $s_i s_{i+1} \ldots s_{i+n}$ such that

$$\big(s_i(x){\geq}0 \wedge s_i(x){-}1{\geq}s_{i+n}(x)\big) \vee \big(s_i(y){\geq}0 \wedge s_i(y){-}1{\geq}s_{i+n}(y) \wedge s_i(x){=}s_{i+n}(x)\big).$$

The analysis then computes the next abstract state A_1 that overapproximates the relational composition of A_0 and R. It takes each disjunction from A_0 and each disjunction from R, composes them, performs rank function synthesis, infers variables that do not change, and constructs the union of the new ranking relations together with A_0. In this case we find:

$$\begin{array}{rr}
\mathsf{RFS}(T_x; C_1) = x & \mathsf{RFS}(T_x; C_2) = x \\
\mathsf{RFS}((T_y \wedge V_{\{x\}}); C_1) = x & \mathsf{RFS}((T_y \wedge V_{\{x\}}); C_2) = y
\end{array}$$

We also find that the last composition $(T_y \wedge V_{\{x\}}; C_2)$ does not change x. Thus,

$$A_1 \quad = \quad \big(A_0 \vee T_x \vee T_x \vee T_x \vee (T_y \wedge V_{\{x\}})\big) \quad = \quad A_0.$$

Since A_0 is a fixpoint and A_0 overapproximates R, we know that $\forall i > 0.\ R^i \subseteq A_0$, that is, $R^+ \subseteq A_0$. Thus, because A_0 is disjunctively well-founded, [17] tells us that R is well-founded—meaning that the loop of our program guarantees termination.

Note that rank function synthesis is extremely efficient, meaning that our implementation of the analysis can compute the relation A_0 for $\ell = 2$ as fast as the Octagon-based termination analyzer (*i.e.* in 0.02s) [2]. In contrast to the Octagon-based analyzer, however, we compute a relation that is sufficiently strong to establish termination.

To sum up, the essence of our method is that we symbolically execute the body of the loop, and then perform abstraction by calling a rank synthesis engine. This in effect abstracts all information except those that are relevant to termination.

3 Formal description

In this section we provide a rigorous description of the proposed termination analysis.

3.1 Programming language

We consider a simple while language in the paper. Let Vars be a finite set of program variables $x, y, z, \ldots$ and let r represent real numbers.

$$\begin{array}{rcl} e & ::= & x \mid r \mid e + e \mid r \times e \\ b & ::= & e{=}e \mid e{\neq}e \mid b \wedge b \mid b \vee b \mid \neg b \\ a & ::= & x{:=}e \mid x{:=}{*} \mid \texttt{assume}(b) \\ c & ::= & a \mid c; c \mid \texttt{while}\, b\, c \mid c \,[]\, c \end{array}$$

Note that the language has two forms of assignments, normal assignment $x{:=}e$ and nondeterministic random assignment $x{:=}{*}$. The nondeterministic assignment is used to model some features of a common programming language, for example C, that are not covered by our language above. Also notice that the language does not include the conditional statement. It can be encoded with `assume` and the nondeterministic choice operator $[]$: $(\texttt{if}\ b\ c_0\ c_1) \stackrel{\text{def}}{=} \big((\texttt{assume}(b); c_0)\,[]\,(\texttt{assume}(\neg b); c_1)\big)$.

The semantics of our language is standard. We remind the reader of only the storage model used in the semantics:

$$\mathsf{St} \stackrel{\text{def}}{=} \mathsf{Vars} \to \mathsf{Real}.$$

This model shows that we assume real variables in this paper. However, changing the type of variables from reals to integers or rationals will not affect the results of the paper, except the ones for the fast termination in Lemma 1 and Theorem 2.

3.2 Abstract domain

Our analysis is parameterized by a domain for representing relations on states. The domain is specified by the following data:

1. A set D and a monotone function $\gamma_r : D \to \mathcal{P}(\mathsf{St}\times\mathsf{St})$ (where the target $\mathcal{P}(\mathsf{St}\times\mathsf{St})$ is ordered by the subset relation).
2. An abstract identity element d_{id} in D, that satisfies

$$\Delta_{\mathsf{St}} \subseteq \gamma_r(d_{\mathsf{id}})$$

   where Δ_{St} is the identity relation on St.
3. An operator $\mathsf{RFS}\colon D \to \mathcal{P}_{\mathsf{fin}}(D) \uplus \{\top\}$, which synthesizes ranking functions. We assume the following two conditions for this operator:
   (a) RFS computes an overapproximation:

$$\mathsf{RFS}(d){\neq}\top \implies \gamma_r(d) \subseteq \bigcup\{\gamma_r(d') \mid d' \in \mathsf{RFS}(d)\}.$$

   (b) $\mathsf{RFS}(d)$ denotes a well-founded relation:

$$\mathsf{RFS}(d){\neq}\top \implies \bigcup\{\gamma_r(d') \mid d' \in \mathsf{RFS}(d)\} \text{ is well-founded.}$$

4. An abstract transfer function $\mathsf{trans}(a)$ for each atomic commands a (i.e., assignments or assume statements). The function $\mathsf{trans}(a)$ has type $D \to \mathcal{P}_{\mathsf{fin}}(D)$, and satisfies

$$\forall d \in D.\ (\gamma_r(d); [\![a]\!]) \subseteq \bigcup\{\gamma_r(d') \mid d' \in \mathsf{trans}(a)(d)\}$$

where the semicolon means the usual composition of relations and $[\![a]\!]$ is the standard relational meaning of the atomic command a.

5. An abstract composition operator $\mathsf{comp}\colon D \times D \to D$ such that

$$\gamma_r(d); \gamma_r(d') \quad \subseteq \quad \gamma_r(\mathsf{comp}(d, d')).$$

Intuitively, the data above means that we have a set D of relations, some of which are well-founded. This set comes with an algorithm RFS, which overapproximates a relation by a ranking relation. It also has operators, trans and comp, that soundly model all the atomic commands and concrete relation composition. One example of D is the set of conjunction of linear constraints. In this case, we can use a linear rank synthesis engine, which we denote LinearRankSyn, and define RFS as will be shown in Section 3.4.

The abstract domain $\mathcal{A}$ of our analyzer is:

$$\mathcal{A} \ \stackrel{\text{def}}{=}\ (\mathcal{P}_{\mathsf{fin}}(D))^{\top} \quad \text{(i.e., } \mathcal{P}(D) \uplus \{\top\}\text{)}.$$

It is ordered by the the subset order $\sqsubseteq$ extended with $\top$. That is, $A \sqsubseteq A'$ iff

$$A' = \top, \qquad \text{or} \qquad (A, A' \in \mathcal{P}_{\mathsf{fin}}(D) \ \text{ and } \ A \subseteq A').$$

Each abstract element A in $\mathcal{A}$ denotes a set of finite or infinite sequences of states, which we call *traces*. The element $\top$ denotes the set of all traces, including infinite ones, and non-$\top$ elements A denote a set of *finite* nonempty traces whose initial and final states are related by some d in A. Let $\gamma_r(A)$ be $\bigcup\{\gamma_r(d) \mid d \in A\}$, the disjunction of d's in A, and define $\mathcal{T}$ to be the set of all nonempty traces:

$$\mathcal{T} \ \stackrel{\text{def}}{=}\ \mathsf{St}^{+} \cup \mathsf{St}^{\infty}.$$

The formal meaning of A is given by a concretization function γ:

$$\begin{aligned} \gamma \ &: \ \mathcal{A} \to \mathcal{P}(\mathcal{T}) \\ \gamma(A) &\stackrel{\text{def}}{=} \mathbf{if}\ (A{=}\top)\ \mathbf{then}\ \mathcal{T}\ \mathbf{else}\ \{\tau \mid \tau \text{ is nonempty, finite, and } \tau_0[\gamma_r(A)]\tau_{|\tau|-1}\} \end{aligned}$$

where $|\tau|$ is the length of the trace τ, and τ_n is the n-th state of the trace τ, and notation $s[r]s'$ means that s, s' are related by r. For instance, when $[x : n, y : m]$ is a state mapping x and y to n and m, a finite trace

$$[x : 1, y : 1][x : 2, y : 2][x : 5, y : 3][x : -2, y : 2]$$

belongs to $\gamma(\{`x{-}1 \geq x,\ `y{-}1 \geq y\})$, because x has a smaller value in the final state than in the initial state.

Our domain $\mathcal{A}$ is a complete semi-lattice. The join of a family $\{A_i\}_{i \in I}$ of elements in $\mathcal{A}$ is given by the union of all A_i's, if none of A_i's is $\top$ and the union is finite. Otherwise, the join is $\top$.

3.3 Generic analysis

Our generic analyzer is an abstract interpretation, defined in a denotational style.

For functions $f\colon D \to \mathcal{A}$ and $g\colon D \times D \to D$, let $f^\dagger, g^\dagger$ be their liftings on $\mathcal{A}$:

$$\begin{aligned} f^\dagger : \mathcal{A} \to \mathcal{A} \qquad & g^\dagger : \mathcal{A} \times \mathcal{A} \to \mathcal{A} \\ f^\dagger(A) &\overset{\text{def}}{=} \textbf{if } (A{=}\top) \textbf{ then } \top \textbf{ else } \textstyle\bigsqcup_{d \in A} f(d) \\ g^\dagger(A,B) &\overset{\text{def}}{=} \textbf{if } (A{=}\top \vee B{=}\top) \textbf{ then } \top \textbf{ else } \textstyle\bigsqcup_{d \in A, d' \in B} \{g(d,d')\}. \end{aligned}$$

Using these liftings, we define the generic analyzer as follows: [4]

$$\begin{aligned} [\![c]\!]^\# &: \mathcal{A} \to \mathcal{A} \\ [\![a]\!]^\# A &\overset{\text{def}}{=} (\mathsf{trans}(a))^\dagger A \\ [\![c_0; c_1]\!]^\# A &\overset{\text{def}}{=} ([\![c_1]\!]^\# \circ [\![c_0]\!]^\#) A \\ [\![c_0 \mathbin{[]} c_1]\!]^\# A &\overset{\text{def}}{=} [\![c_0]\!]^\# A \sqcup [\![c_1]\!]^\# A \\ [\![\mathtt{while}\, b\, c]\!]^\# A &\overset{\text{def}}{=} \textbf{let } F \overset{\text{def}}{=} \lambda A'.[\![\mathtt{assume}(b); c]\!]^\# (\{d_{\mathsf{id}}\} \sqcup A') \\ &\qquad \textbf{in } [\![\mathtt{assume}(\neg b)]\!]^\# \big(\mathsf{comp}^\dagger(A, \mathsf{fix}\,(\mathsf{RFS}^\dagger \circ F))\big) \end{aligned}$$

Intuitively, the argument A represents a set of finite or infinite traces that happen before the command c. The analyzer computes an overapproximation of all traces that are obtained by continuing the execution of c from the end of traces in A.

Our definition assumes an operator fix. The fix operator takes a function of the form $\mathsf{RFS}^\dagger \circ F : \mathcal{A} \to \mathcal{A}$, and returns an abstract element A in the image of $\mathsf{RFS}^\dagger$ such that

$$A = \top \ \ \vee \ \ \big(A \neq \top \wedge (\mathsf{RFS}^\dagger \circ F)(A) \neq \top \wedge \gamma_r((\mathsf{RFS}^\dagger \circ F)(A)) \subseteq \gamma_r(A)\big).$$

One can use the standard fixpoint iteration to define fix,[5] because the above condition holds for all post fixpoints A of $(\mathsf{RFS}^\dagger \circ F)$ (that are in the image of $\mathsf{RFS}^\dagger$). However, this is not mandatory. In fact, a more optimized fix operator is used in the analysis of Section 3.4, which in some cases does not even compute a post fixpoint.

The most interesting case of the analysis is the loop. The best way to understand this case is to assume that fix is the standard fixpoint operator and to see a sequence generated during the iterative fixpoint computation:

$$\begin{aligned} A_0 &= \{\}, \\ A_1 &= A_0 \sqcup (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \\ &= (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \\ A_2 &= A_1 \sqcup (\mathsf{RFS}^\dagger \circ F)\big(\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\}\big) \\ &= (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^2\{d_{\mathsf{id}}\}, \\ A_3 &= A_2 \sqcup (\mathsf{RFS}^\dagger \circ F)\big(\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^2\{d_{\mathsf{id}}\}\big) \\ &= (\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^2\{d_{\mathsf{id}}\} \sqcup (\mathsf{RFS}^\dagger \circ F)^3\{d_{\mathsf{id}}\}, \\ &\qquad \cdots \end{aligned}$$

[4] In the definition, we view $\mathsf{RFS}, \mathsf{trans}(a)$ as functions of type $D \to (\mathcal{P}_{\mathsf{fin}}(D))^\top$.

[5] In this case, $\mathsf{fix}\,(\mathsf{RFS}^\dagger \circ F)$ is defined by the limit of the sequence $\{A_n\}$ where $A_0 = \{\}$ and $A_{n+1} = A_n \sqcup (\mathsf{RFS}^\dagger \circ F)(A_n)$.

Here we used the fact that $\mathsf{RFS}^\dagger \circ F$ preserves $\sqcup$. Note that in each step, we apply the lifted rank-synthesis algorithm $\mathsf{RFS}^\dagger$ to the analysis result of the loop body $F(A_n)$. This application of RFS throws away all the information from $F(A_n)$, except the one necessary for proving termination. Another thing to note is that the input A is not used in this fixpoint computation at all. As the expansion of A_3 shows, the fixpoint computation effectively starts with $(\mathsf{RFS}^\dagger \circ F)\{d_{\mathsf{id}}\}$, which means the results of running the loop body once on all states. The input A is pre-composed later to the computed fixpoint. This change of the starting point is crucial for the soundness of our analysis, because it ensures that the analyzer overapproximates the relation between any states (not just initial states) at a loop and the following states at the same loop (so that we can apply a known termination proof rule based on disjunctively well-founded relations [17]).

Given a program c, the analyzer works as follows:

$$\begin{array}{l}\textsc{Analysis}(c) \stackrel{\text{def}}{=} \textbf{let } A = [\![c]\!]^\#(\{d_{\mathsf{id}}\}) \\ \qquad\qquad\qquad\quad \textbf{in if } (A \neq \top) \textbf{ then } (\textbf{return} \text{ “Terminates”}) \textbf{ else } (\textbf{return} \text{ “Unknown”}).\end{array}$$

Theorem 1. *If* $\textsc{Analysis}(c)$ *returns “Terminates”, then c terminates on all states.*

The proof of this theorem is given in the full version of the paper [5]. There we also clarify what we mean by “terminates on all states”, by defining a concrete trace semantics of commands based on Cousot's work [9].

3.4 Linear Rank Abstraction

The linear rank abstraction is an instance of our generic analysis, by the domain of linear constraints and a linear ranking synthesis algorithm LINEARRANKSYN

Let r represent real numbers. Consider constraints C defined by the grammar below:

$$\begin{array}{lcl} E & ::= & x \mid {}^{\backprime}x \mid x' \mid r \mid E+E \mid r \times E \\ P & ::= & E=E \mid E \neq E \mid E<E \mid E>E \mid E \leq E \mid E \geq E \\ C & ::= & P \mid \mathsf{true} \mid C \wedge C \end{array}$$

This grammar ensures that all the constraints are the conjunction of linear constraints. Note that a constraint can have three kinds of variables; a normal variable x denoting the current value of program variable x; a pre-primed variable ${}^{\backprime}x$ storing the initial value of x; post-primed variables y' that usually denotes values which were once stored in program variables during computation. We assume that there are finitely many normal variables (Vars) and finitely many pre-primed variables (${}^{\backprime}\mathsf{Vars}$), and that there is a one-to-one correspondence between these two kinds of variables. For post-primed variables, however, we assume an infinite set.

Each constraint means a relation on St. For each state s, let ${}^{\backprime}s$ be a function from ${}^{\backprime}\mathsf{Vars}$ to Real such that for every pre-primed variable ${}^{\backprime}x$, ${}^{\backprime}s({}^{\backprime}x)$ is $s(x)$ for the corresponding normal variable x. The meaning function γ_r of constraints C is defined as follows:

$$\gamma_r(C) \stackrel{\text{def}}{=} \{(s_0, s_1) \mid ({}^{\backprime}s_0, s_1 \models \exists X'.C)\}$$

where X' is the set of post-primed variables in C and $\models$ is the usual satisfaction relation in first-order logic. Note that all post-primed variables in the constraint C are implicitly existentially-quantified.

The linear rank abstraction uses the set of constraints C as the parameter set D of the generic analysis. The identity element d_{id} is the identity relation

$$d_{\mathsf{id}} \overset{\text{def}}{=} \bigwedge_{x \in \mathsf{Vars}} `x{=}x.$$

Assume that we are given an enumeration $x_0, \ldots, x_n$ of all program variables in Vars. Call an expression E *normalized,* when (1) E does not contain any pre or post primed variables and (2) it is of the form $a_{i_0} \times x_{i_0} + \ldots a_{i_k} \times x_{i_k} + a$ with $a_{i_0} = 1$ or -1 and $i_0 < i_1 \ldots < i_k$. Note that in a normalized expression E, the coefficient of the first variable in E according to the given enumeration is 1 or -1. Conceptually, LinearRankSyn implements a function of the type:[6]

$$D \to (\{(E, r) \mid E \text{ is normalized and } r \text{ is a positive real}\}) \uplus \{\top\}.$$

The output $\top$ indicates that the algorithm fails to discover a ranking function, because (the implementation of) the algorithm is incomplete or the input constraint defines a non-well-founded relation between pre-primed variables and normal variables. The other output (E, r) means that the algorithm succeeds to find a ranking function which overapproximates the given constraint. Concretely, for a normalized expression E and a positive real r, let

$$T_{E,r} \overset{\text{def}}{=} (`E \geq 0 \ \wedge \ `E{-}r \geq E),$$

where expression $`E$ is $E$ with all normal variables $x$ replaced by corresponding pre-primed variables $`x$. The output (E, r) of LinearRankSyn(C) means that

$$(\exists X'.C) \implies T_{E,r}$$

where X' is the set of all post-primed variables in C.

Assume that we have chosen a fixed positive real dec for the analysis, which is very small (in particular smaller than 1). Using LinearRankSyn and dec, we define the operator RFS as follows:

$$\mathsf{RFS}(C) \overset{\text{def}}{=} \begin{cases} \{\} & \text{if } C \vdash \mathsf{false} \\ \{T_{E,\mathsf{dec}}\} & \text{else if } \textsc{LinearRankSyn}(C){=}(E,r) \text{ and } r \geq \mathsf{dec} \\ \top & \text{otherwise} \end{cases}$$

where $\vdash$ is a sound (but not necessarily complete) theorem prover. Note that the result of RFS is always of the form $T_{E,\mathsf{dec}}$, so the second subscript of T is not necessary. From now on, we write T_E for $T_{E,\mathsf{dec}}$.

[6] Usually the implementation of linear rank synthesis returns a tuple (E, r, b) where E is an expression without any pre or post primed variable whose value is decreasing, r is a decrement, and b is a lower bound of E. Our analysis picks the absolute value a of the coefficient of the first variable x_i in E, transforms E/a to a normal form E', and regards $(E' - b/a, r/a)$ as an output from LinearRankSyn.

The abstract transfer functions for atomic commands are defined following Floyd's strongest postcondition semantics:

$$\begin{aligned}
[\![x{:=}*]\!]^{\#} C &\stackrel{\text{def}}{=} \{C[x'/x]\} \quad (x' \text{ is fresh})\\
[\![x{:=}e]\!]^{\#} C &\stackrel{\text{def}}{=} \{C[x'/x] \wedge x{=}(e[x'/x])\} \quad (x' \text{ is fresh})\\
[\![\mathtt{assume}(b)]\!]^{\#} C &\stackrel{\text{def}}{=} \textbf{if } (C \wedge b \vdash \mathsf{false}) \textbf{ then } \{\}\\
&\qquad \textbf{else } \{C_0, \ldots, C_n \mid C_0 \vee \ldots \vee C_n = \mathsf{norm}(C \wedge b)\}.
\end{aligned}$$

Here norm is the standard transformation that takes a formula in the propositional logic and transforms the formula to disjunctive normal form.

Next, we define the abstract composition comp. Let fresh be an operator on constraints C that renames all post-primed variables fresh. Let ‘Vars be the set of pre-primed variables. The abstract composition is defined as follows

$$\mathsf{comp}(C_0, C_1) \quad \stackrel{\text{def}}{=} \quad \textbf{let } \big(C_2 = \mathsf{fresh}(C_1)\big) \textbf{ in } \Big(C_0[Y'/\mathsf{Vars}] \wedge C_2[Y'/\text{‘}\mathsf{Vars}]\Big).$$

The variable set Y' in the definition denotes a set of fresh post-primed variables, that has as many elements as Vars. The two substitutions there replace a normal variable x and the corresponding pre-primed variable ‘x by the same post-primed variable x'.

Finally, we specify a fix operator. For each function $(\mathsf{RFS}^{\dagger} \circ F)$ on sets of constraints C, let $\{G_n\}_n$ be the standard fixpoint iteration sequence: $G_0 = \{\}$ and $G_{n+1} = G_n \sqcup (\mathsf{RFS}^{\dagger} \circ F)(G_n)$. Given G, our fix operator returns the first G_n such that

$$G_n{=}\top \quad \vee \quad \Big(G_n{\neq}\top \;\wedge\; G_{n+1}{\neq}\top \;\wedge\; \forall C \in G_{n+1}.\, \exists C' \in G_n.\, C \vdash C'\Big).$$

This definition assumes that some G_n satisfies the above property. If such a G_n does not exist, the fix operator is not defined, so the analysis can diverge during the fixpoint computation. In Theorem 2, we will discharge this assumption and prove the termination of the linear rank abstraction.

Example 1. Consider the program c below:

$$\mathtt{while}\ (x > 0 \wedge y > 0)\ (x{:=}x{-}1 \,[]\, y{:=}y{-}1).$$

Given c, the analysis starts the fixpoint computation from the empty set $A_0 = \{\}$. The first iteration of the fixpoint computation is done in two steps. First, it applies the abstract transfer function of the loop body to $\{d_{\mathsf{id}}\} \cup A_0 = \{d_{\mathsf{id}}\}$:

$$\begin{aligned}
&[\![\mathtt{assume}(x{>}0{\wedge}y{>}0); (x{:=}x{-}1 \,[]\, y{:=}y{-}1)]\!]^{\#}(\{d_{\mathsf{id}}\})\\
&= [\![x{:=}x{-}1 \,[]\, y{:=}y{-}1]\!]^{\#}\{d_{\mathsf{id}} \wedge x{>}0 \wedge y{>}0\}\\
&= [\![x{:=}x{-}1]\!]^{\#}\{d_{\mathsf{id}} \wedge x{>}0 \wedge y{>}0\} \;\cup\; [\![y{:=}y{-}1]\!]^{\#}\{d_{\mathsf{id}} \wedge x{>}0 \wedge y{>}0\}\\
&= [\![x{:=}x{-}1]\!]^{\#}\{\text{‘}x{=}x \wedge \text{‘}y{=}y \wedge x{>}0 \wedge y{>}0\} \;\cup\; [\![y{:=}y{-}1]\!]^{\#}\{\text{‘}x{=}x \wedge \text{‘}y{=}y \wedge x{>}0 \wedge y{>}0\}\\
&= \{\text{‘}x{=}x' \wedge \text{‘}y{=}y \wedge x'{>}0 \wedge y{>}0 \wedge x{=}x'{-}1, \quad \text{‘}x{=}x \wedge \text{‘}y{=}y' \wedge x{>}0 \wedge y'{>}0 \wedge y{=}y'{-}1\}.
\end{aligned}$$

Next, the analysis calls LINEARRANKSYN twice with each of the two elements in the result set above. These function calls return x and y, from which the analysis constructs two ranking relations below:

$$T_x \stackrel{\text{def}}{=} (`x \geq 0 \;\wedge\; `x - \mathsf{dec} \geq x) \quad \text{and} \quad T_y \stackrel{\text{def}}{=} (`y \geq 0 \;\wedge\; `y - \mathsf{dec} \geq y).$$

The result A_1 of the first iteration is $\{T_x, T_y\}$.

The second fixpoint iteration computes:

$$A_1 \;\sqcup\; (\mathsf{RFS}^\dagger \circ [\![\texttt{assume}(x{>}0 \wedge y{>}0); (x{:=}x{-}1 \,[]\, y{:=}y{-}1)]\!]^\#) A_1.$$

We show that the abstract element on the right hand side of the join, denoted A_2', is again A_1, so that the fixpoint computation converges here. To compute A_2', the analyzer first transforms A_1 according to the abstract meaning of the loop body. This results in a set with four elements:

$$\begin{aligned}\{\, & T_x[x'/x] \;\wedge\; x'{>}0 \;\wedge\; y{>}0 \;\wedge\; x{=}x'{-}1, \quad T_x[y'/y] \;\wedge\; x{>}0 \;\wedge\; y'{>}0 \;\wedge\; y{=}y'{-}1, \\ & T_y[x'/x] \;\wedge\; x'{>}0 \;\wedge\; y{>}0 \;\wedge\; x{=}x'{-}1, \quad T_y[y'/y] \;\wedge\; x{>}0 \;\wedge\; y'{>}0 \;\wedge\; y{=}y'{-}1 \,\}.\end{aligned}$$

The first two elements come from transforming T_x according to the left and right branches of the loop body. The other two elements are obtained similarly from T_y. Next, the analysis calls LINEARRANKSYN with all the four elements above. These four calls return x, x, y and y, which represent well-founded relations T_x, T_x, T_y, T_y. Thus, A_2' is the same as T_x and T_y, and the fixpoint computation stops here.

After the fixpoint computation, the analysis composes the identity relation $\{d_{\mathsf{id}}\}$ with the result of the fixpoint computation:

$$\begin{aligned}\mathsf{comp}^\dagger(\{d_{\mathsf{id}}\}, \{T_x, T_y\}) &= \{`x{=}x_0' \wedge `y{=}y_0' \wedge T_x[x_0'/`x],\; `x{=}x_0' \wedge `y{=}y_0' \wedge T_y[y_0'/`y]\} \\ &= \{T_x,\; T_y\}.\end{aligned}$$

Finally, we apply $[\![\texttt{assume}(\neg(x > 0 \wedge y > 0))]\!]^\#$ to the set above, which gives a set with four constraints:

$$\{\; T_x \wedge x \leq 0, \quad T_x \wedge y \leq 0, \quad T_y \wedge x \leq 0, \quad T_y \wedge y \leq 0 \;\}.$$

Since the result is not $\top$, the analysis concludes that the given program c terminates. $\square$

In the example above, the fixpoint computation converges after two iterations. In the first iteration, which computes A_1, it finds ranking functions, and in the next iteration, it confirms that the ranking functions are preserved by the loop. In fact, we can prove that the fixpoint computation of the analysis always follows the same pattern, and finishes in two iterations. Suppose that LINEARRANKSYN is well-behaved, such that

1. RFS always computes an optimal ranking function, in the sense that

$$(\mathsf{RFS}(C) = \{T_E\} \;\wedge\; \gamma_r(C) \subseteq \gamma_r(T_{E+b})) \implies b \geq 0,$$

2. RFS depends only on the (relational) meaning of its argument.

Lemma 1. *For all commands c and normalized expressions E, if there is a constraint $C \in [\![c]\!]^\#\{T_E\}$ such that* $\mathsf{RFS}(C) = \{T_F\}$ *and $\gamma_r(C) \neq \emptyset$, then F is of the form $E - b$ for some nonnegative b.*

Proof. The proof appears in the full version of this paper [5]. □

Theorem 2 (Fast Convergence). *Suppose that the theorem prover* $\vdash$ *is complete. Then, for all commands* c*, the fixpoint iteration of*

$$G = \lambda A.\ (\mathsf{RFS}^{\dagger} \circ [\![c]\!]^{\#})(\{d_{\mathsf{id}}\} \sqcup A)$$

terminates at most in two steps. Specifically, $G^2(\{\})$ *is* $\top$*, or the result of* $\mathsf{fix}\, G$ *is* $\{\}$ *or* $G(\{\})$*.*

Proof. Suppose that $G^2(\{\})$ is not $\top$. This implies that both $G(\{\})$ and $G^2(\{\})$ are finite sets of T_E's for normalized expressions E, because $G(= \mathsf{RFS}^{\dagger} \circ [\![c]\!]^{\#})$ preserves $\top$. If $G(\{\})$ is empty, $\{\}$ is the fixpoint of G, thus becoming the result of $\mathsf{fix}\, G$, as claimed in the theorem. To prove the other nonempty case, suppose that $G(\{\})$ is a nonempty finite collection $A = \{T_{E_1}, \ldots, T_{E_n}\}$. We need to show that for each T_F in $G(A)$, there exists $T_{E_i} \in A$ such that $T_F \vdash T_{E_i}$, which is equivalent to $\gamma_r(T_F) \subseteq \gamma_r(T_{E_i})$ due to the completeness assumption about the prover. Pick T_F in $G(A)$. Since $G(= \mathsf{RFS}^{\dagger} \circ [\![c]\!]^{\#})$ preserves the join operator, there exists T_{E_i} in A such that $T_F \in G(\{T_{E_i}\})$. This means that $\mathsf{RFS}(C) = \{T_F\}$ for some constraint C in $[\![c]\!]^{\#}(T_{E_i})$. Note that since RFS filters out all the provably inconsistent constraints and the prover is assumed complete, $\gamma_r(C)$ is not empty. Thus, by Lemma 1, there is a nonnegative b such that $F = E - b$. This gives the required $\gamma_r(T_F) \subseteq \gamma_r(T_E)$. □

Note that the theorem suggests that we could have used a different fix operator that does not call the prover at all and just returns $G^2(\{\})$. We do not take this alternative in the paper, since it is too specific for the RFS operator in this section; if RFS also keeps track of equality information, this two-step convergence result no longer holds.

Refinement with simple equalities The linear rank abstraction cannot prove the termination of the program in Section 2. When the linear rank abstraction is run for the program, it finds the ranking functions x and y for the true and false branches of the program, but loses the information that the else branch does not change the value of x, which is crucial for the termination proof. As a result, the linear rank abstraction returns $\top$, and reports, incorrectly, the possibility of nontermination.

One way to solve this problem and improve the precision of the linear rank abstraction is to use a more precise RFS operator that additionally keeps simple forms of equalities. Concretely, this refinement keeps all the definitions of the linear rank abstraction, except that it replaces the rank synthesizer RFS of the linear rank abstraction by RFS$'$ below:

$$\mathsf{RFS}'(C) \overset{\mathrm{def}}{=} \mathbf{if}\ (\mathsf{RFS}(C){=}\top)\ \mathbf{then}\ \top\ \mathbf{else}\ \big\{T_E \wedge (\wedge_{(C \vdash \text{‘}x=x)} \text{‘}x{=}x) \mid T_E \in \mathsf{RFS}(C)\big\}.$$

When this refined analysis is given the program in Section 2, it follows the informal description in that section and proves the termination of the program.

4 Experimental evaluation

In order to evaluate the utility of our approach we have implemented the analysis in this paper, and then compared it to several known termination tools. The tools used in the experiments are as follows:

LR) LINEARRANKTERM is the new variance analysis that implements the linear rank abstraction with simple equalities in Section 3.4. This tool is implemented using CIL [15] allowing the analysis of programs written in C. However, no notion of shape is used in these implementations, restricting the input to only arithmetic programs. The tool uses RANKFINDER [16] as its linear rank synthesis engine and uses the Simplify prover [11] to filter out inconsistent states and check the implication between abstract states.

O) OCTATERM is the variance analysis [2] induced by the octagon analysis OCTANAL [14].

P) POLYTERM is the variance analysis [2] similarly induced from the polyhedra analysis POLY based on the New Polka Polyhedra library [12].

T) TERMINATOR [8].

These tools, except for TERMINATOR, were all run on a 2GHz AMD64 processor using Linux 2.6.16. TERMINATOR was executed on a 3GHz Pentium 4 using Windows XP SP2. Using different machines is unfortunate but somewhat unavoidable due to constraints on software library dependencies, etc. Note, however, that TERMINATOR running on the faster machine was still slower overall, so the qualitative results are meaningful. In any case, the running times are somewhat incomparable since on failed proofs TERMINATOR produces a counterexample path, but LINEARRANKTERM, OCTATERM and POLYTERM give a suspect pair of states

Fig. 1 contains the results from the experiments performed with these analyses.[7] For example, Fig. 1(a) shows the outcome of the provers on example programs included in the OCTANAL distribution. Example 3 is an abstracted version of heapsort, and Example 4 of bubblesort.

Fig. 1(b) contains the results of experiments on fragments of Windows device drivers. These examples are small because we currently must hand-translate them before applying all of the tools but TERMINATOR.

Fig. 1(c) contains the results from experiments with the 4 tools on examples from the POLYRANK distribution.[8] The examples can be characterized as small but famously difficult (*e.g.* McCarthy's 91 function). Note that LINEARRANKTERM performs poorly on these examples because of the limitations of RANKFINDER. Many of these examples involve phase changes or tricky arithmetic in the algorithm.

From these experiments we can see that LINEARRANKTERM is very fast and precise. The prototype we have developed indicates that a termination analyzer using abstractions based on ranking functions shows a lot of promise.

[7] The programs used in our experiments except the ones for drivers are available in http://www.dcs.qmul.ac.uk/∼aziem/esop. Unfortunately, we could not put the driver examples in the web page, because that might cause a problem related to intellectual property.

[8] Note also that there is no benchmark number 5 in the original distribution. We have used the same numbering scheme as in the distribution so as to avoid confusion.

	1	2	3	4	5	6
LR	0.01 ✓	0.01 ✓	0.08 ✓	0.09 ✓	0.02 ✓	0.06 ✓
O	0.11 ✓	0.08 ✓	6.03 ✓	1.02 ✓	0.16 ✓	0.76 ✓
P	1.40 ✓	1.30 ✓	10.90 ✓	2.12 ✓	1.80 ✓	1.89 ✓
T	6.31 ✓	4.93 ✓	T/O -	T/O -	33.24 ✓	3.98 ✓

(a) Results from experiments with termination tools on arithmetic examples from the Octagon Library distribution.

	1	2	3	4	5	6	7	8	9	10
LR	0.23 ✓	0.20 ⊘	0.00 ⊘	0.04 ✓	0.00 ✓	0.03 ✓	0.07 ✓	0.03 ✓	0.01 ⊘	0.03 ✓
O	1.42 ✓	1.67 ⊘	0.47 ⊘	0.18 ✓	0.06 ✓	0.53 ✓	0.50 ✓	0.32 ✓	0.14 ⊘	0.17 ✓
P	4.66 ✓	6.35 ⊘	1.48 ⊘	1.10 ✓	1.30 ✓	1.60 ✓	2.65 ✓	1.89 ✓	2.42 ⊘	1.27 ✓
T	10.22 ✓	31.51 ⊘	20.65 ⊘	4.05 ✓	12.63 ✓	67.11 ✓	298.45 ✓	444.78 ✓	T/O -	55.28 ✓

(b) Results from experiments with termination tools on small arithmetic examples taken from Windows device drivers. Note that the examples are small as they must currently be hand-translated for the three tools.

	1	2	3	4	6	7	8	9	10	11	12
LR	0.19 ✓	0.02 ✓	0.01 †	0.02 †	0.02 †	0.01 †	0.04 †	0.01 †	0.03 †	0.02 †	0.01 †
O	0.30 †	0.05 †	0.11 †	0.50 †	0.10 †	0.17 †	0.16 †	0.12 †	0.35 †	0.86 †	0.12 †
P	1.42 ✓	0.82 ✓	1.06 †	2.29 †	2.61 †	1.28 †	0.24 †	1.36 ✓	1.69 †	1.56 †	1.05 †
T	435.23 ✓	61.15 ✓	T/O -	T/O -	75.33 ✓	T/O -	T/O -	T/O -	T/O -	T/O -	10.31 †

(c) Results from experiments with termination tools on arithmetic examples from the POLYRANK distribution.

Fig. 1. Experiments with 4 termination provers/analyses. **LR** is used to represent LINEARRANKTERM, **O** is used to represent OCTATERM, an Octagon-based variance analysis. **P** is POLYTERM, a Polyhedra-based variance analysis. The **T** represents TERMINATOR [8]. Times are measured in seconds. The timeout threshold was set to 500s. ✓="a proof was found". †="false counterexample returned". T/O = "timeout". ⊘="termination bug found". Note that pointers and aliasing from the device driver examples were removed by a careful hand translation when passed to the tools **O**, **P** and **LR**. Note that a time of 0.00 means that the analysis was too fast to be measured by the timing utilities used.

Acknowledgements. We would like to thank Peter O'Hearn for encouragements and insightful comments on our work, Andrey Rybalchenko for explaining the subtleties of linear ranking functions and RANKFINDER, and Amir Ben-Amram and Neil Jones for helping us to understand the size-change termination. We also acknowledge detailed comments on the paper from Amir and anonymous referees, which help us to improve the presentation of the paper. Chawdhary was supported by a Microsoft PhD studentship, and Yang was supported by EPSRC.

References

1. I. Balaban, A. Pnueli, and L. Zuck. Ranking abstraction as companion to predicate abstraction. In *FORTE'05*, 2005.
2. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance analyses from invariance analyses. In *POPL'07*, 2007.
3. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'05*, 2005.
4. M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis through combination of type based norms. *ACM Trans. Progam. Lang. Syst.*, 29(2), 2007.
5. A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. Manuscript, 2008. Available at http://www.dcs.qmul.ac.uk/∼aziem/paper/esop08-full.pdf.
6. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1), 1999.
7. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06*, 2006.
8. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV'06*, 2006.
9. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Comput. Sci.*, 277(1–2):47–103, 2002.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, 1979.
11. D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003.
12. B. Jeannet. NewPolka polyhedra library. http://pop-art.inrialpes.fr/people/bjeannet/newpolka/index.html.
13. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'01*, 2001.
14. A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Comput.*, 19:31–100, 2006.
15. G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL:intermediate language and tools for analysis and transformation of C programs. In *CC'02*, 2002.
16. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04*, 2004.
17. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'04*, 2004.
18. A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1948. Reprinted in: The early British computer conferences, vol. 14 of Charles Babbage Institute Reprint Series for the History of Computing, MIT Press, 1989.
19. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. *Logic Journal of IGPL*, Sept. 2006.