Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs*

Cristina David, Daniel Kroening, and Matt Lewis

University of Oxford

Abstract. Proving program termination is typically done by finding a well-founded ranking function for the program states. Existing termination provers typically find ranking functions using either linear algebra or templates. As such they are often restricted to finding linear ranking functions over mathematical integers. This class of functions is insufficient for proving termination of many terminating programs, and furthermore a termination argument for a program operating on mathematical integers does not always lead to a termination argument for the same program operating on fixed-width machine integers. We propose a termination analysis able to generate nonlinear, lexicographic ranking functions and nonlinear recurrence sets that are correct for fixed-width machine arithmetic and floating-point arithmetic. Our technique is based on a reduction from program *termination* to second-order *satisfaction*. We provide formulations for termination and non-termination in a fragment of second-order logic with restricted quantification which is decidable over finite domains [1]. The resulting technique is a sound and complete analysis for the termination of finite-state programs with fixed-width integers and IEEE floating-point arithmetic.

Keywords: Termination, Non-Termination, Lexicographic Ranking Functions, Bit-vector Ranking Functions, Floating-Point Ranking Functions.

1 Introduction

The halting problem has been of central interest to computer scientists since it was first considered by Turing in 1936 [2]. Informally, the halting problem is concerned with answering the question "does this program run forever, or will it eventually terminate?"

Proving program termination is typically done by finding a *ranking function* for the program states, i.e. a monotone map from the program's state space to a well-ordered set. Historically, the search for ranking functions has been constrained in various syntactic ways, leading to incompleteness, and is performed over abstractions that do not soundly capture the behaviour of physical computers. In this paper, we present a sound and complete method for deciding whether a program with a fixed amount of storage terminates. Since such programs are

* Supported by UK EPSRC EP/J012564/1 and ERC project 280053.

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 183–204, 2015.
DOI: 10.1007/978-3-662-46669-8_8

necessarily finite state, our problem is much easier than Turing's, but is a better fit for analysing computer programs.

When surveying the area of program termination chronologically, we observe an initial focus on monolithic approaches based on a single measure shown to decrease over all program paths [3,4], followed by more recent techniques that use termination arguments based on Ramsey's theorem [5,6,7]. The latter proof style builds an argument that a transition relation is disjunctively well founded by composing several small well-foundedness arguments. The main benefit of this approach is the simplicity of local termination measures in contrast to global ones. For instance, there are cases in which linear arithmetic suffices when using local measures, while corresponding global measures require nonlinear functions or lexicographic orders.

One drawback of the Ramsey-based approach is that the validity of the termination argument relies on checking the *transitive closure* of the program, rather than a single step. As such, there is experimental evidence that most of the effort is spent in reachability analysis [7,8], requiring the support of powerful safety checkers: there is a trade-off between the complexity of the termination arguments and that of checking their validity.

As Ramsey-based approaches are limited by the state of the art in safety checking, recent research shifts back to more complex termination arguments that are easier to check [8,9]. Following the same trend, we investigate its extreme: *unrestricted* termination arguments. This means that our ranking functions may involve nonlinearity and lexicographic orders: we do not commit to any particular syntactic form, and do not use templates. Furthermore, our approach allows us to *simultaneously* search for proofs of *non-termination*, which take the form of recurrence sets.

Figure 1 summarises the related work with respect to the restrictions they impose on the transition relations as well as the form of the ranking functions computed. While it supports the observation that the majority of existing termination analyses are designed for linear programs and linear ranking functions, it also highlights another simplifying assumption made by most state-of-the-art termination provers: that bit-vector semantics and integer semantics give rise to the same termination behaviour. Thus, most existing techniques treat fixed-width machine integers (bit-vectors) and IEEE floats as mathematical integers and reals, respectively [7,10,3,11,12,8].

By assuming bit-vector semantics to be identical to integer semantics, these techniques ignore the wrap-around behaviour caused by overflows, which can be unsound. In Section 2, we show that integers and bit-vectors exhibit incomparable behaviours with respect to termination, i.e. programs that terminate for integers need *not* terminate for bit-vectors and vice versa. Thus, abstracting bit-vectors with integers may give rise to *unsound* and *incomplete* analyses.

We present a technique that treats linear and nonlinear programs uniformly and it is not restricted to finding linear ranking functions, but can also compute lexicographic nonlinear ones. Our approach is constraint-based and relies on second-order formulations of termination and non-termination. The obvious issue is that, due to its expressiveness, second-order logic is very difficult to reason in, with many second-order theories becoming undecidable even when the corresponding first-order theory is decidable. To make solving our constraints tractable, we formulate termination and non-termination inside a fragment of second-order logic with restricted quantification, for which we have built a solver in [1]. Our method is sound and complete for bit-vector programs – for any program, we find a proof of either its termination or non-termination.

Ranking argument	Program							
	Rationals/Integers		Reals		Bit-vectors		Floats	
	L	NL	L	NL	L	NL	L	NL
Linear lexicographic	[13,4,9,3]	–	[14]	–	✓	✓	✓	✓
Linear non-lexicographic	[10,7,15,11,12,8]	[12]	[14]	–	✓ [16]	✓ [16]	✓	✓
Nonlinear lexicographic	–	–	–	–	✓	✓	✓	✓
Nonlinear non-lexicographic	[12]	[12]	–	–	✓	✓	✓	✓

Legend: ✓ = we can handle; – = no available works; L = linear; NL = nonlinear

Fig. 1. Summary of related termination analyses

The main contributions of our work can be summarised as follows:

- We rephrased the termination and non-termination problems as second-order satisfaction problems. This formulation captures the (non-)termination properties of all of the loops in the program, including nested loops. We can use this to analyse all the loops at once, or one at a time. Our treatment handles termination and non-termination uniformly: both properties are captured in the same second-order formula.
- We designed a bit-level accurate technique for computing ranking functions and recurrence sets that correctly accounts for the wrap-around behaviour caused by under- and overflows in bit-vector and floating-point arithmetic. Our technique is not restricted to finding linear ranking functions, but can also compute lexicographic nonlinear ones.
- We implemented our technique and tried it on a selection of programs handling both bit-vectors and floats. In our implementation we made use of a solver for a fragment of second-order logic with restricted quantification that is decidable over finite domains [1].

Limitations. Our algorithm proves termination for transition systems with finite state spaces. The (non-)termination proofs take the form of ranking functions and program invariants that are expressed in a quantifier-free language. This formalism is powerful enough to handle a large fragment of C, but is not rich enough to analyse code that uses unbounded arrays or the heap. Similar to other termination analyses [9], we could attempt to alleviate the latter limitation by abstracting programs with heap to arithmetic ones [17]. Also, we have not yet added support for recursion or `goto` to our encoding.

2 Motivating Examples

Figure 1 illustrates the most common simplifying assumptions made by existing termination analyses:

(i) programs use only linear arithmetic.
(ii) terminating programs have termination arguments expressible in linear arithmetic.
(iii) the semantics of bit-vectors and mathematical integers are equivalent.
(iv) the semantics of IEEE floating-point numbers and mathematical reals are equivalent.

To show how these assumptions are violated by even simple programs, we draw the reader's attention to the programs in Figure 2 and their curious properties:

- Program (a) breaks assumption (i) as it makes use of the bit-wise & operator. Our technique finds that an admissible ranking function is the linear function $R(x) = x$, whose value decreases with every iteration, but cannot decrease indefinitely as it is bounded from below. This example also illustrates the lack of a direct correlation between the linearity of a program and that of its termination arguments.
- Program (b) breaks assumption (ii), in that it has no linear ranking function. We prove that this loop terminates by finding the nonlinear ranking function $R(x) = |x|$.
- Program (c) breaks assumption (iii). This loop is terminating for bit-vectors since x will eventually overflow and become negative. Conversely, the same program is non-terminating using integer arithmetic since $x > 0 \rightarrow x+1 > 0$ for any integer x.
- Program (d) also breaks assumption (iii), but "the other way": it terminates for integers but not for bit-vectors. If each of the variables is stored in an unsigned k-bit word, the following entry state will lead to an infinite loop:

$$M = 2^k - 1, \quad N = 2^k - 1, \quad i = M, \quad j = N - 1$$

- Program (e) breaks assumption (iv): it terminates for reals but not for floats. If x is sufficiently large, rounding error will cause the subtraction to have no effect.
- Program (f) breaks assumption (iv) "the other way": it terminates for floats but not for reals. Eventually x will become sufficiently small that the nearest representable number is 0.0, at which point it will be rounded to 0.0 and the loop will terminate.

Up until this point, we considered examples that are not soundly treated by existing techniques as they don't fit in the range of programs addressed by these techniques. Next, we look at some programs that are handled by existing termination tools via dedicated analyses. We show that our method handles them uniformly, without the need for any special treatment.

```
while (x > 0) {
  x = (x - 1) & x;
}
```

(a) Taken from [16]

```
while (x != 0) {
  x = -x / 2;
}
```

(b)

```
while(x > 0) {
  x++;
}
```

(c)

```
while (i<M || j<N) {
  i = i + 1;
  j = j + 1;
}
```

(d) Taken from [18]

```
float x;

while (x > 0.0) {
  x -= 1.0;
}
```

(e)

```
float x;

while (x > 0.0) {
  x *= 0.5;
}
```

(f)

```
while (x != 0) {
  if (x > 0)
    x--;
  else
    x++;
}
```

(g) Taken from [9]

```
y = 1;

while (x > 0) {
  x = x - y;
}
```

(h)

```
while (x>0 && y>0 && z>0){
  if (y > x) {
    y = z;
    x = nondet();
    z = x - 1;
  } else {
    z = z - 1;
    x = nondet();
    y = x - 1;
  }
}
```

(i) Taken from [19]

Fig. 2. Motivational examples, mostly taken from the literature

– Program (g) is a linear program that is shown in [9] not to admit (without prior manipulation) a lexicographic linear ranking function. With our technique we can find the nonlinear ranking function $R(x) = |x|$.
– Program (h) illustrates conditional termination. When proving program termination we are simultaneously solving two problems: the search for a termination argument, and the search for a supporting invariant [20]. For this loop, we find the ranking function $R(x) = x$ together with the supporting invariant $y = 1$.
– For program (i) we find a nonlinear lexicographic ranking function $R(x, y, z) = (x < y, z)$.[1] We are not aware of any linear ranking function for this program.

As with all of the termination proofs presented in this paper, the ranking functions above were all found completely automatically.

3 Preliminaries

Given a program, we first formalise its termination argument as a ranking function (Section 3.1). Subsequently, we discuss bit-vector semantics and illustrate differences between machine arithmetic and integer arithmetic that show that the abstraction of bit-vectors to mathematical integers is unsound (Section 3.2).

3.1 Termination and Ranking Functions

A program P is represented as a transition system with state space X and transition relation $T \subseteq X \times X$. For a state $x \in X$ with $T(x, x')$ we say x' is a successor of x under T.

Definition 1 (Unconditional termination). *A program is said to be* unconditionally terminating *if there is no infinite sequence of states* $x_1, x_2, \ldots \in X$ *with* $\forall i.\ T(x_i, x_{i+1})$.

We can prove that the program is unconditionally terminating by finding a ranking function for its transition relation.

Definition 2 (Ranking function). *A function* $R : X \to Y$ *is a* ranking function *for the transition relation* T *if* Y *is a well-founded set with order* $>$ *and* R *is injective and monotonically decreasing with respect to* T. *That is to say:*

$$\forall x, x' \in X. T(x, x') \Rightarrow R(x) > R(x')$$

[1] This termination argument is somewhat subtle. The Boolean values *false* and *true* are interpreted as 0 and 1, respectively. The Boolean $x < y$ thus eventually decreases, that is to say once a state with $x \geq y$ is reached, x never again becomes greater than y. This means that as soon as the "else" branch of the if statement is taken, it will continue to be taken in each subsequent iteration of the loop. Meanwhile, if $x < y$ has not decreased (i.e., we have stayed in the same branch of the "if"), then z does decrease. Since a Boolean only has two possible values, it cannot decrease indefinitely. Since $z > 0$ is a conjunct of the loop guard, z cannot decrease indefinitely, and so R proves that the loop is well founded.

Definition 3 (Linear function). *A* linear function $f : X \to Y$ *with* $\dim(X) = n$ *and* $\dim(Y) = m$ *is of the form:*

$$f(\boldsymbol{x}) = M\boldsymbol{x}$$

where M *is an* $n \times m$ *matrix.*

In the case that $\dim(Y) = 1$, this reduces to the inner product

$$f(\boldsymbol{x}) = \boldsymbol{\lambda} \cdot \boldsymbol{x} + c\ .$$

Definition 4 (Lexicographic ranking function). *For* $Y = Z^m$*, we say that a ranking function* $R : X \to Y$ *is* lexicographic *if it maps each state in* X *to a tuple of values such that the loop transition leads to a decrease with respect to the lexicographic ordering for this tuple. The total order imposed on* Y *is the lexicographic ordering induced on tuples of* Z*'s. So for* $y = (z_1, \ldots, z_m)$ *and* $y' = (z'_1, \ldots, z'_m)$*:*

$$y > y' \iff \exists i \leq m . z_i > z'_i \wedge \forall j < i . z_j = z'_j$$

We note that some termination arguments require lexicographic ranking functions, or alternatively, ranking functions whose co-domain is a countable ordinal, rather than just $\mathbb{N}$.

3.2 Machine Arithmetic vs. Peano Arithmetic

Physical computers have bounded storage, which means they are unable to perform calculations on mathematical integers. They do their arithmetic over fixed-width binary words, otherwise known as bit-vectors. For the remainder of this section, we will say that the bit-vectors we are working with are k-bits wide, which means that each word can hold one of 2^k bit patterns. Typical values for k are 32 and 64.

Machine words can be interpreted as "signed" or "unsigned" values. Signed values can be negative, while unsigned values cannot. The encoding for signed values is two's complement, where the most significant bit b_{k-1} of the word is a "sign" bit, whose weight is $-(2^k - 1)$ rather than $2^k - 1$. Two's complement representation has the property that $\forall x. - x = (\sim x) + 1$, where $\sim(\bullet)$ is bitwise negation. Two's complement also has the property that addition, multiplication and subtraction are defined identically for unsigned and signed numbers.

Bit-vector arithmetic is performed modulo 2^k, which is the source of many of the differences between machine arithmetic and Peano arithmetic[2]. To give an example, $(2^k - 1) + 1 \equiv 0 \pmod{2^k}$ provides a counterexample to the statement $\forall x . x + 1 > x$, which is a theorem of Peano arithmetic but not of modular arithmetic. When an arithmetic operation has a result greater than 2^k, it is said

[2] ISO C requires that unsigned arithmetic is performed modulo 2^k, whereas the overflow case is undefined for signed arithmetic. In practice, the undefined behaviour is implemented just as if the arithmetic had been unsigned.

to “overflow”. If an operation does not overflow, its machine-arithmetic result is the same as the result of the same operation performed on integers.

The final source of disagreement between integer arithmetic and bit-vector arithmetic stems from width conversions. Many programming languages allow numeric variables of different types, which can be represented using words of different widths. In C, a `short` might occupy 16 bits, while an `int` might occupy 32 bits. When a k-bit variable is assigned to a j-bit variable with $j < k$, the result is truncated mod 2^j. For example, if x is a 32-bit variable and y is a 16-bit variable, y will hold the value 0 after the following code is executed:

```
x = 65536;
y = x;
```

As well as machine arithmetic differing from Peano arithmetic on the operators they have in common, computers have several “bitwise” operations that are not taken as primitive in the theory of integers. These operations include the Boolean operators `and, or, not, xor` applied to each element of the bit-vector. Computer programs often make use of these operators, which are nonlinear when interpreted in the standard model of Peano arithmetic[3].

4 Termination as Second-Order Satisfaction

The problem of program verification can be reduced to the problem of finding solutions to a second-order constraint [21,22]. Our intention is to apply this approach to termination analysis. In this section we show how several variations of both the termination and the non-termination problem can be uniformly defined in second-order logic.

Due to its expressiveness, second-order logic is very difficult to reason in, with many second-order theories becoming undecidable even when the corresponding first-order theory is decidable. In [1], we have identified and built a solver for a fragment of second-order logic with restricted quantification, which we call second-order SAT (see Definition 5).

Definition 5 (Second-Order SAT).

$$\exists S_1 \ldots S_m . Q_1 x_1 \ldots Q_n x_n . \sigma$$

Where the S_i's range over predicates, the Q_i's are either $\exists$ or $\forall$, the x_i's range over Boolean values, and σ is a quantifier-free propositional formula whose free variables are the x_i's. Each S_i has an associated arity $\mathrm{ar}(S_i)$ *and* $S_i \subseteq \mathbb{B}^{\mathrm{ar}(S_i)}$. *Note that $Q_1 x_1 \ldots Q_n x_n . \sigma$ is the special case of a propositional formula with first-order quantification, i.e. QBF.*

We note that by existentially quantifying over Skolem functions, formulae with arbitrary first-order quantification can be brought into the synthesis fragment [23], so the fragment is semantically less restrictive than it looks.

[3] Some of these operators can be seen as linear in a different algebraic structure, e.g. `xor` corresponds to addition in the Galois field $\mathrm{GF}(2^k)$.

In the rest of this section, we show that second-order SAT is expressive enough to encode both termination and non-termination.

4.1 An Isolated, Simple Loop

We will begin our discussion by showing how to encode in second-order SAT the (non-)termination of a program consisting of a single loop with no nesting. For the time being, a loop $L(G, T)$ is defined by its guard G and body T such that states x satisfying the loop's guard are given by the predicate $G(x)$. The body of the loop is encoded as the transition relation $T(x, x')$, meaning that state x' is reachable from state x via a single iteration of the loop body. For example, the loop in Figure 2a is encoded as:

$$G(x) = \{x \mid x > 0\}$$
$$T(x, x') = \{\langle x, x'\rangle \mid x' = (x - 1) \,\&\, x\}$$

We will abbreviate this with the notation:

$$G(x) \triangleq x > 0$$
$$T(x, x') \triangleq x' = (x - 1) \,\&\, x$$

Unconditional Termination. We say that a loop $L(G, T)$ is unconditionally terminating iff it eventually terminates regardless of the state it starts in. To prove unconditional termination, it suffices to find a ranking function for $T \cap (G \times X)$, i.e. T restricted to states satisfying the loop's guard.

Theorem 1. *The loop $L(G, T)$ terminates from every start state iff formula* [**UT**] *(Definition 6, Figure 3) is satisfiable.*

As the existence of a ranking function is equivalent to the satisfiability of the formula [**UT**], a satisfiability witness is a ranking function and thus a proof of L's unconditional termination.

Returning to the program from Figure 2a, we can see that the corresponding second-order SAT formula [**UT**] is satisfiable, as witnessed by the function $R(x) = x$. Thus, $R(x) = x$ constitutes a proof that the program in Figure 2a is unconditionally terminating.

Note that different formulations for unconditional termination are possible. We are aware of a proof rule based on transition invariants, i.e. supersets of the transition relation's transitive closure [21]. This formulation assumes that the second-order logic has a primitive predicate for disjunctive well-foundedness. By contrast, our formulation in Definition 6 does not use a primitive disjunctive well-foundedness predicate.

Non-termination. Dually to termination, we might want to consider the non-termination of a loop. If a loop terminates, we can prove this by finding a ranking function witnessing the satisfiability of formula [**UT**]. What then would a proof of non-termination look like?

Definition 6 (Unconditional Termination Formula [UT]).

$$\exists R.\forall x, x'.G(x) \wedge T(x, x') \rightarrow R(x) > 0 \wedge R(x) > R(x')$$

Definition 7 (Non-Termination Formula – Open Recurrence Set [ONT]).

$$\begin{aligned}\exists N, x_0.\forall x.\exists x'.N(x_0) \wedge & \\ & N(x) \rightarrow G(x) \wedge \\ & N(x) \rightarrow T(x, x') \wedge N(x')\end{aligned}$$

Definition 8 (Non-Termination Formula – Closed Recurrence Set [CNT]).

$$\begin{aligned}\exists N, x_0.\forall x, x'.N(x_0) \wedge & \\ & N(x) \rightarrow G(x) \wedge \\ & N(x) \wedge T(x, x') \rightarrow N(x')\end{aligned}$$

Definition 9 (Non-Termination Formula – Skolemized Open Recurrence Set* [SNT]*).

$$\begin{aligned}\exists N, C, x_0.\forall x.N(x_0) \wedge & \\ & N(x) \rightarrow G(x) \wedge \\ & N(x) \rightarrow T(x, C(x)) \wedge N(C(x))\end{aligned}$$

Fig. 3. Formulae encoding the termination and non-termination of a single loop

Since our program's state space is finite, a transition relation induces an infinite execution iff some state is visited infinitely often, or equivalently $\exists x.T^+(x, x)$. Deciding satisfiability of this formula directly would require a logic that includes a transitive closure operator, $\bullet^+$. Rather than introduce such an operator, we will characterise non-termination using the second-order SAT formula [**ONT**] (Definition 7, Figure 3) encoding the existence of an *(open) recurrence set*, i.e. a nonempty set of states N such that for each $s \in N$ there exists a transition to some $s' \in N$ [24].

Theorem 2. *The loop $L(G, T)$ has an infinite execution iff formula* [**ONT**] *(Definition 7) is satisfiable.*

If this formula is satisfiable, N is an open recurrence set for L, which proves L's non-termination. The issue with this formula is the additional level of quantifier alternation as compared to second-order SAT (it is an $\exists\forall\exists$ formula). To eliminate the innermost existential quantifier, we introduce a Skolem function C that chooses the successor x', which we then existentially quantify over. This results in formula [**SNT**] (Definition 9, Figure 3).

Theorem 3. *Formula* [**ONT**] *(Definition 7) and formula* [**SNT**] *(Definition 9) are equisatisfiable.*

This extra second-order term introduces some complexity to the formula, which we can avoid if the transition relation T is deterministic.

Definition 10 (Determinism). *A relation T is deterministic iff each state x has exactly one successor under T:*

$$\forall x.\exists x'.T(x,x') \wedge \forall x''.T(x,x'') \rightarrow x'' = x'$$

In order to describe a deterministic *program* in a way that still allows us to sensibly talk about termination, we assume the existence of a special sink state s with no outgoing transitions and such that $\neg G(s)$ for any of the loop guards G. The program is deterministic if its transition relation is deterministic for all states except s.

When analysing a deterministic loop, we can make use of the notion of a *closed recurrence set* introduced by Chen et al. in [25]: for each state in the recurrence set N, *all* of its successors must be in N. The existence of a closed recurrence set is equivalent to the satisfiability of formula [**CNT**] in Definition 8, which is already in second-order SAT without needing Skolemization.

We note that if T is deterministic, every open recurrence set is also a closed recurrence set (since each state has at most one successor). Thus, the non-termination problem for deterministic transition systems is equivalent to the satisfiability of formula [**CNT**] from Figure 3.

Theorem 4. *If T is deterministic, formula* [**ONT**] *(Definition 7) and formula* [**CNT**] *(Definition 8) are equisatisfiable.*

So if our transition relation is deterministic, we can say, without loss of generality, that non-termination of the loop is equivalent to the existence of a closed recurrence set. However, if T is non-deterministic, it may be that there is an open recurrence set but not closed recurrence set. To see this, consider the following loop:

```
while(x != 0) {
  y = nondet();
  x = x-y;
}
```

It is clear that this loop has many non-terminating executions, e.g. the execution where nondet() always returns 0. However, each state has a successor that exits the loop, i.e. when nondet() returns the value currently stored in x. Thus, this loop has an open recurrence set, but no closed recurrence set and hence we cannot give a proof of its non-termination with [**CNT**] and instead must use [**SNT**].

4.2 An Isolated, Nested Loop

Termination. If a loop $L(G,T)$ has another loop $L'(G',T')$ nested inside it, we cannot directly use [**UT**] to express the termination of L. This is because the single-step transition relation T must include the transitive closure of the inner loop T'^*, and we do not have a transitive closure operator in our logic. Therefore to encode the termination of L, we construct an over-approximation $T_o \supseteq T$ and use this in formula [**UT**] to specify a ranking function. Rather than explicitly construct T_o using, for example, abstract interpretation, we add constraints to our formula that encode the fact that T_o is an over-approximation of T, and that it is precise enough to show that R is a ranking function.

As the generation of such constraints is standard and covered by several other works [21,22], we will not provide the full algorithm, but rather illustrate it through the example in Figure 4. For the current example, the termination formula is given on the right side of Figure 4: T_o is a summary of L_1 that over-approximates its transition relation; R_1 and R_2 are ranking functions for L_1 and L_2, respectively.

Non-termination. Dually to termination, when proving non-termination, we need to under-approximate the loop's body and apply formula [**CNT**]. Under-approximating the inner loop can be done with a nested existential quantifier, resulting in ∃∀∃ alternation, which we could eliminate with Skolemization. However, we observe that unlike a ranking function, the defining property of a recurrence set is *non relational* – if we end up in the recurrence set, we do not care exactly where we came from as long as we know that it was also somewhere in the recurrence set. This allows us to cast non-termination of nested loops as the formula shown in Figure 6, which does not use a Skolem function.

```
L1 :
while (i<n){
  j = 0;

L2 :
  while (j<=i){
    j = j + 1;
  }

  i = i + 1;
}
```

$$\begin{aligned}
&\exists T_o, R_1, R_2. \forall i, j, n, i', j', n'.\\
&\qquad i < n \rightarrow T_o(\langle i, j, n\rangle, \langle i, 0, n\rangle) \wedge\\
&\qquad j \leq i \wedge T_o(\langle i', j', n'\rangle, \langle i, j, n\rangle) \rightarrow R_2(i, j, n) > 0 \wedge\\
&\qquad\qquad R_2(i, j, n) > R_2(i, j+1, n) \wedge\\
&\qquad\qquad T_o(\langle i', j', n'\rangle, \langle i, j+1, n\rangle) \wedge\\
&\qquad i < n \wedge S(\langle i, j, n\rangle, \langle i', j', n'\rangle) \wedge j' > i' \rightarrow R_1(i, j, n) > 0 \wedge\\
&\qquad\qquad R_1(i, j, n) > R_1(i+1, j, n)
\end{aligned}$$

Fig. 4. A program with nested loops and its termination formula

Definition 11 (Conditional Termination Formula [CT]).

$$\begin{aligned}
&\exists R, W. \forall x, x'. I(x) \wedge G(x) \rightarrow W(x) \wedge\\
&\qquad G(x) \wedge W(x) \wedge T(x, x') \rightarrow W(x') \wedge R(x) > 0 \wedge R(x) > R(x')
\end{aligned}$$

Fig. 5. Formula encoding conditional termination of a loop

If the formula on the right-hand side of the figure is satisfiable, then L_1 is non-terminating, as witnessed by the recurrence set N_1 and the initial state x_0 in which the program begins executing. There are two possible scenarios for L_2's termination:

- If L_2 is terminating, then N_2 is an inductive invariant that reestablished N_1 after L_2 stops executing: $\neg G_2(x) \wedge N_2(x) \wedge P_2(x, x') \rightarrow N_1(x')$.
- If L_2 is non-terminating, then $N_2 \wedge G_2$ is its recurrence set.

4.3 Composing a Loop with the Rest of the Program

Sometimes the termination behaviour of a loop depends on the rest of the program. That is to say, the loop may not terminate if started in some particular state, but that state is not actually reachable on entry to the loop. The program as a whole terminates, but if the loop were considered in isolation we would not be able to prove that it terminates. We must therefore encode a loop's interaction with the rest of the program in order to do a sound termination analysis.

Let us assume that we have done some preprocessing of our program which has identified loops, straight-line code blocks and the control flow between these. In particular, the control flow analysis has determined which order these code blocks execute in, and the nesting structure of the loops.

Conditional Termination. Given a loop $L(G, T)$, if L's termination depends on the state it begins executing in, we say that L is *conditionally terminating.* The information we require of the rest of the program is a predicate I which over-approximates the set of states that L may begin executing in. That is to say, for each state x that is reachable on entry to L, we have $I(x)$.

Theorem 5. *The loop $L(G, T)$ terminates when started in any state satisfying $I(x)$ iff formula* **[CT]** *(Definition 11, Figure 5) is satisfiable.*

If formula **[CT]** is satisfiable, two witnesses are returned:

```
L1:
while (G1) {
  P1;
L2:
  while (G2) {
    B2;
  }
  P2;
}
```

$$\begin{aligned} &\exists N_1, N_2, x_0 . \forall x, x' . \\ &N_1(x_0) \wedge \\ &N_1(x) \rightarrow G_1(x) \wedge \\ &N_1(x) \wedge P_1(x, x') \rightarrow N_2(x') \wedge \\ &G_2(x) \wedge N_2(x) \wedge B_2(x, x') \rightarrow N_2(x') \wedge \\ &\neg G_2(x) \wedge N_2(x) \wedge P_2(x, x') \rightarrow N_1(x') \end{aligned}$$

Fig. 6. Formula encoding non-termination of nested loops

- W is an inductive invariant of L that is established by the initial states I if the loop guard G is met.
- R is a ranking function for L as restricted by W – that is to say, R need only be well founded on those states satisfying $W \wedge G$. Since W is an inductive invariant of L, R is strong enough to show that L terminates from any of its initial states.

W is called a *supporting invariant* for L and R proves termination relative to W. We require that $I \wedge G$ is strong enough to establish the base case of W's inductiveness.

Conditional termination is illustrated by the program in Figure 2h, which is encoded as:

$$I(\langle x, y\rangle) \triangleq y = 1$$
$$G(\langle x, y\rangle) \triangleq x > 0$$
$$T(\langle x, y\rangle, \langle x', y'\rangle) \triangleq x' = x - y \wedge y' = y$$

If the initial states I are ignored, this loop cannot be shown to terminate, since any state with $y = 0$ and $x > 0$ would lead to a non-terminating execution.

However, formula [**CT**] is satisfiable, as witnessed by:

$$R(\langle x, y\rangle) = x$$
$$W(\langle x, y\rangle) \triangleq y = 1$$

This constitutes a proof that the program as a whole terminates, since the loop always begins executing in a state that guarantees its termination.

4.4 Generalised Termination and Non-termination Formula

At this point, we know how to construct two formulae for a loop L: one that is satisfiable iff L is terminating and another that is satisfiable iff it is non-terminating. We will call these formulae ϕ and ψ, respectively:

$$\exists P_T. \forall x, x'. \phi(P_T, x, x')$$
$$\exists P_N. \forall x. \psi(P_N, x)$$

We can combine these:

$$(\exists P_T. \forall x, x'. \phi(P_T, x, x')) \vee (\exists P_N. \forall x.\, \psi(P_N, x))$$

Which simplifies to:

Definition 12 (Generalised Termination Formula [GT]).

$$\exists P_T, P_N. \forall x, x', y.\, \phi(P_T, x, x') \vee \psi(P_N, y)$$

Since L either terminates or does not terminate, this formula is a tautology in second-order SAT. A solution to the formula would include witnesses P_N and P_T, which are putative proofs of non-termination and termination respectively. Exactly one of these will be a genuine proof, so we can check first one and then the other.

4.5 Solving the Second-Order SAT Formula

In order to solve the second-order generalised formula [**GT**], we use the solver described in [1]. For any satisfiable formula, the solver is guaranteed to find a satisfying assignment to all the second-order variables.

In the context of our termination analysis, such a satisfying assignment returned by the solver represents either a proof of termination or non-termination, and takes the form of an imperative program written in the language $\mathcal{L}$. An $\mathcal{L}$-program is a list of instructions, each of which matches one of the patterns shown in Figure 7. An instruction has an opcode (such as `add` for addition) and one or more operands. An operand is either a constant, one of the program's inputs or the result of a previous instruction. The $\mathcal{L}$ language has various arithmetic and logical operations, as well as basic branching in the form of the `ite` (if-then-else) instruction.

Integer arithmetic instructions:

```
add a b     sub a b     mul a b     div a b
neg a       mod a b     min a b     max a b
```

Bitwise logical and shift instructions:

```
and  a b    or   a b    xor a b
lshr a b    ashr a b    not a
```

Unsigned and signed comparison instructions:

```
le  a b     lt  a b     sle  a b
slt a b     eq  a b     neq  a b
```

Miscellaneous logical instructions:

```
implies a b     ite a b c
```

Floating-point arithmetic:

```
fadd a b    fsub a b    fmul a b    fdiv a b
```

Fig. 7. The language $\mathcal{L}$

5 Soundness, Completeness and Complexity

In this section, we show that $\mathcal{L}$ is expressive enough to capture (non-)termination proofs for every bit-vector program. By using this result, we then show that our analysis terminates with a valid proof for every input program.

Lemma 1. *Every function $f : X \to Y$ for finite X and Y is computable by a finite $\mathcal{L}$-program.*

Proof. Without loss of generality, let $X = Y = \mathbb{N}_b^k$ the set of k-tuples of natural numbers less than b. A very inefficient construction which computes the first coordinate of the output y is:

```
t1 = f(0)
t2 = v1 == 1
t3 = ITE(t2, f(1), t1)
t4 = v1 == 2
t5 = ITE(t4, f(2), t3)
...
```

Where the `f(n)` are literal constants that are to appear in the program text. This program is of length $2b - 1$, and so all k co-ordinates of the output y are computed by a program of size at most $2bk - k$.

Corollary 1. *Every finite subset $A \subseteq B$ is computable by a finite $\mathcal{L}$-program by setting $X = B, Y = 2$ in Lemma 1 and taking the resulting function to be the characteristic function of A.*

Theorem 6. *Every terminating bit-vector program has a ranking function that is expressible in $\mathcal{L}$.*

Proof. Let $v_1, \ldots, v_k$ be the variables of the program P under analysis, and let each be b bits wide. Its state space $\mathcal{S}$ is then of size 2^{bk}. A ranking function $R : \mathcal{S} \to \mathcal{D}$ for P exists iff P terminates. Without loss of generality, $\mathcal{D}$ is a well-founded total order. Since R is injective, we have that $\|\mathcal{D}\| \geq \|\mathcal{S}\|$. If $\|\mathcal{D}\| > \|\mathcal{S}\|$, we can construct a function $R' : \mathcal{S} \to \mathcal{D}'$ with $\|\mathcal{D}'\| = \|\mathcal{S}\|$ by just setting $R' = R|_{\mathcal{S}}$, i.e. R' is just the restriction of R to $\mathcal{S}$. Since $\mathcal{S}$ already comes equipped with a natural well ordering we can also construct $R'' = \iota \circ R'$ where $\iota : \mathcal{D}' \to \mathcal{S}$ is the unique order isomorphism from $\mathcal{D}'$ to $\mathcal{S}$. So assuming that P terminates, there is some ranking function R'' that is just a permutation of $\mathcal{S}$. If the number of variables $k > 1$, then in general the ranking function will be lexicographic with dimension $\leq k$ and each co-ordinate of the output being a single b-bit value.

Then by Lemma 1 with $X = Y = \mathcal{S}$, there exists a finite $\mathcal{L}$-program computing R''.

Theorem 7. *Every non-terminating bit-vector program has a non-termination proof expressible in $\mathcal{L}$.*

Proof. A proof of non-termination is a triple $\langle N, C, x_0 \rangle$ where $N \subseteq \mathcal{S}$ is a (finite) recurrence set and $C : \mathcal{S} \to \mathcal{S}$ is a Skolem function choosing a successor for each $x \in N$. The state space $\mathcal{S}$ is finite, so by Lemma 1 both N and C are computed by finite $\mathcal{L}$-programs and x_0 is just a ground term.

Theorem 8. *The generalised termination formula* **[GT]** *for any loop L is a tautology when P_N and P_T range over $\mathcal{L}$-computable functions.*

Proof. For any P, P', σ, σ, if $P \models \sigma$ then $(P, P') \models \sigma \vee \sigma'$.

By Theorem 6, if L terminates then there exists a termination proof P_T expressible in $\mathcal{L}$. Since ϕ is an instance of **[CT]**, $P_T \models \phi$ (Theorem 5) and for any P_N, $(P_T, P_N) \models \phi \vee \psi$.

Similarly if L does not terminate for some input, by Theorem 7 there is a non-termination proof P_N expressible in $\mathcal{L}$. Formula ψ is an instance of **[SNT]** and so $P_N \models \psi$ (Theorem 3), hence for any P_T, $(P_T, P_N) \models \phi \vee \psi$.

So in either case (L terminates or does not), there is a witness in $\mathcal{L}$ satisfying $\phi \vee \psi$, which is an instance of **[GT]**.

Theorem 9. *Our termination analysis is sound and complete – it terminates for all input loops L with a correct termination verdict.*

Proof. By Theorem 8, the specification spec is satisfiable. In [1], we show that the second-order SAT solver is semi-complete, and so is guaranteed to find a satisfying assignment for spec. If L terminates then P_T is a termination proof (Theorem 5), otherwise P_N is a non-termination proof (Theorem 3). Exactly one of these purported proofs will be valid, and since we can check each proof with a single call to a SAT solver we simply test both and discard the one that is invalid.

6 Experiments

To evaluate our algorithm, we implemented a tool that generates a termination specification from a C program and calls the second-order SAT solver in [1] to obtain a proof. We ran the resulting termination prover, named JUGGERNAUT, on 47 benchmarks taken from the literature and SV-COMP'15 [26]. We omitted exactly those SV-COMP'15 benchmarks that made use of arrays or recursion. We do not have arrays in our logic and we had not implemented recursion in our frontend (although the latter can be syntactically rewritten to our input format).

To provide a comparison point, we also ran ARMC [27] on the same benchmarks. Each tool was given a time limit of 180 s, and was run on an unloaded 8-core 3.07 GHz Xeon X5667 with 50 GB of RAM. The results of these experiments are given in Figure 8.

It should be noted that the comparison here is imperfect, since ARMC is solving a different problem – it checks whether the program under analysis would terminate if run with unbounded integer variables, while we are checking whether the program terminates with bit-vector variables. This means that ARMC's verdict differs from ours in 3 cases (due to the differences between integer and bit-vector semantics). There are a further 7 cases where our tool is able to find a proof and ARMC cannot, which we believe is due to our more expressive proof language. In 3 cases, ARMC times out while our tool is able to find a termination proof. Of these, 2 cases have nested loops and the third has an infinite number of terminating lassos. This is not a problem for us, but can be difficult for provers that enumerate lassos.

On the other hand, ARMC is *much* faster than our tool. While this difference can partly be explained by much more engineering time being invested in ARMC, we feel that the difference is probably inherent to the difference in the two approaches – our solver is more general than ARMC, in that it provides a complete proof system for both termination and non-termination. This comes at the cost of efficiency: JUGGERNAUT is slow, but unstoppable.

Benchmark	Expected	ARMC Verdict	ARMC Time	JUGGERNAUT Verdict	JUGGERNAUT Time
loop1.c	✓	✓	0.06 s	✓	1.3 s
loop2.c	✓	✓	0.06 s	✓	1.4 s
loop3.c	✓	✓	0.06 s	✓	1.8 s
loop4.c	✓	✓	0.12 s	✓	2.9 s
loop5.c	✓	✓	0.12 s	✓	5.3 s
loop6.c	✓	✓	0.05 s	✓	1.2s
loop7.c [20]	✓	?	0.05 s	✓	8.3 s
loop8.c	✓	?	0.06 s	✓	1.3 s
loop9.c	✓	✓	0.11 s	✓	1.6 s
loop10.c	✓	✗	0.05 s	✓	1.3 s
loop11.c	✗	✓	0.05 s	✗	1.4 s
loop43.c [9]	✓	✓	0.07 s	✓	1.5 s
loop44.c [9]	✗	?	0.05 s	✗	10.5 s
loop45.c [9]	✓	✓	0.12 s	✓	4.3 s
loop46.c [9]	✓	?	0.05 s	✓	1.5 s
loop47.c	✓	✓	0.10 s	✓	1.8 s
loop48.c	✓	✓	0.06 s	✓	1.4 s
loop49.c	✗	?	0.05 s	✗	1.3 s
svcomp1.c [28]	✓	✓	0.11 s	✓	2.3 s
svcomp2.c	✓	✓	0.05 s	✓	1.5 s
svcomp3.c [19]	✓	✓	0.15 s	✓	146.4 s
svcomp4.c [4]	✗	✗	0.09 s	✗	2.1 s
svcomp5.c [29]	✓	✓	0.38 s	–	T/O
svcomp6.c [20]	✓	–	T/O	✓	29.1 s
svcomp7.c [20]	✓	✓	0.09 s	✓	5.5 s
svcomp8.c [30]	✓	?	0.05 s	–	T/O
svcomp9.c [9]	✓	✓	0.10 s	✓	1.5 s
svcomp10.c [9]	✓	✓	0.11 s	✓	4.5 s
svcomp11.c [9]	✓	✓	0.20 s	✓	14.6 s
svcomp12.c [31]	✓	–	T/O	✓	10.9 s
svcomp13.c	✓	?	0.07 s	✓	35.1 s
svcomp14.c [32]	✓	–	T/O	✓	30.8 s
svcomp15.c [33]	✓	?	0.12 s	–	T/O
svcomp16.c [33]	✓	✓	0.06 s	✓	2.2 s
svcomp17.c [8]	✓	✓	0.05 s	–	T/O
svcomp18.c [34]	✓	?	0.27 s	–	T/O
svcomp25.c	✓	?	0.05 s	–	T/O
svcomp26.c	✓	✓	0.26 s	✓	3.2 s
svcomp27.c [18]	✗	✓	0.11 s	–	T/O
svcomp28.c [18]	✓	✓	0.13 s	–	T/O
svcomp29.c [3]	✓	?	0.05 s	–	T/O
svcomp37.c	✓	✓	0.16 s	✓	2.1 s
svcomp38.c	✓	✓	0.10 s	–	T/O
svcomp39.c	✓	✓	0.25 s	–	T/O
svcomp40.c [35]	✓	?	0.07 s	✓	25.5 s
svcomp41.c [35]	✓	?	0.07 s	✓	25.5 s
svcomp42.c	✓	✓	0.22 s	–	T/O
Correct			28		35
Incorrect for bit-vectors			3		0
Unknown			13		0
Timeout			3		12

Key: ✓ = terminating, ✗ = non-terminating, ? = unknown (tool terminated with an inconclusive verdict)

Fig. 8. Experimental results

Of the 47 benchmarks, 2 use nonlinear operations in the program (loop6 and loop11), and 5 have nested loops (svcomp6, svcomp12, svcomp18, svcomp40, svcomp41). JUGGERNAUT handles the nonlinear cases correctly and rapidly. It solves 4 of the 5 nested loops in less than 30 s, but times out on the 5th.

In conclusion, these experiments confirm our conjecture that second-order SAT can be used effectively to prove termination and non-termination. In particular, for programs with nested loops, nonlinear arithmetic and complex termination arguments, the versatility given by a general purpose solver is very valuable.

7 Conclusions and Related Work

There has been substantial prior work on automated program termination analysis. Figure 1 summarises the related work with respect to the assumptions they make about programs and ranking functions. Most of the techniques are specialised in the synthesis of linear ranking functions for linear programs over integers (or rationals) [7,15,10,3,11,4,9,8]. Among them, Lee et al. make use of transition predicate abstraction, algorithmic learning, and decision procedures [15], Leike and Heizmann propose linear ranking templates [14], whereas Bradley et al. compute lexicographic linear ranking functions supported by inductive linear invariants [4].

While the synthesis of termination arguments for linear programs over integers is indeed well covered in the literature, there is very limited work for programs over machine integers. Cook et al. present a method based on a reduction to Presburger arithmetic, and a template-matching approach for predefined classes of ranking functions based on reduction to SAT- and QBF-solving [16]. Similarly, the only work we are aware of that can compute nonlinear ranking functions for imperative loops with polynomial guards and polynomial assignments is [12]. However, this work extends only to polynomials.

Given the lack of research on termination of nonlinear programs, as well as programs over bit-vectors and floats, our work focused on covering these areas. One of the obvious conclusions that can be reached from Figure 1 is that most methods tend to specialise on a certain aspect of termination proving that they can solve efficiently. Conversely to this view, we aim for generality, as we do not restrict the form of the synthesised ranking functions, nor the form of the input programs.

As mentioned in Section 1, approaches based on Ramsey's theorem compute a set of local termination conditions that decrease as execution proceeds through the loop and require expensive reachability analyses [5,6,7]. In an attempt to reduce the complexity of checking the validity of the termination argument, Cook et al. present an iterative termination proving procedure that searches for lexicographic termination arguments [9], whereas Kroening et al. strengthen the termination argument such that it becomes a transitive relation [8]. Following the same trend, we search for lexicographic nonlinear termination arguments that can be verified with a single call to a SAT solver.

Proving program termination implies the simultaneous search for a termination argument and a supporting invariant. Brockschmidt et al. share the same representation of the state of the termination proof between the safety prover and the ranking function synthesis tool [20]. Bradley et al. combine the generation of ranking functions with the generation of invariants to form a single constraint solving problem such that the necessary supporting invariants for the ranking function are discovered on demand [4]. In our setting, both the ranking function and the supporting invariant are iteratively constructed in the same refinement loop.

While program termination has been extensively studied, much less research has been conducted in the area of proving non-termination. Gupta et al. dynamically enumerate lasso-shaped candidate paths for counterexamples, and then statically prove their feasibility [24]. Chen et al. prove non-termination via reduction to safety proving [25]. Their iterative algorithm uses counterexamples to a fixed safety property to refine an under-approximation of a program. In order to prove both termination and non-termination, Harris et al. compose several program analyses (termination provers for multi-path loops, non-termination provers for cycles, and global safety provers) [33]. We propose a uniform treatment of termination and non-termination by formulating a generalised second-order formula whose solution is a proof of one of them.

References

1. David, C., Kroening, D., Lewis, M.: Second-order propositional satisfiability. CoRR abs/1409.4925 (2014)
2. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42, 230–265 (1936)
3. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
5. Codish, M., Genaim, S.: Proving termination one loop at a time. In: WLPE, pp. 48–59 (2003)
6. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41 (2004)
7. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426 (2006)
8. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
9. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 47–61. Springer, Heidelberg (2013)
10. Ben-Amram, A.M., Genaim, S.: On the linear ranking problem for integer linear-constraint loops. In: POPL, pp. 51–62 (2013)

11. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 365–380. Springer, Heidelberg (2013)
12. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
13. Ben-Amram, A.M., Genaim, S.: Ranking functions for linear-constraint loops. J. ACM 61(4), 26 (2014)
14. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 172–186. Springer, Heidelberg (2014)
15. Lee, W., Wang, B.-Y., Yi, K.: Termination analysis with algorithmic learning. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 88–104. Springer, Heidelberg (2012)
16. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)
17. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL, pp. 211–222 (2010)
18. Nori, A.V., Sharma, R.: Termination proofs from tests. In: ESEC/SIGSOFT FSE, pp. 246–256 (2013)
19. Ben-Amram, A.M.: Size-change termination, monotonicity constraints and ranking functions. Logical Methods in Computer Science 6(3) (2010)
20. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 413–429. Springer, Heidelberg (2013)
21. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI, pp. 405–416 (2012)
22. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, pp. 281–292 (2008)
23. Van Benthem, J., Doets, K.: Higher-order logic. In: Handbook of Philosophical Logic, pp. 189–243. Springer Netherlands (2001)
24. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: POPL, pp. 147–158 (2008)
25. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014)
26. SV-COMP (2015), http://sv-comp.sosy-lab.org/2015/
27. Rybalchenko, A.: ARMC, http://www7.in.tum.de/~rybal/armc/
28. Avery, J.: Size-change termination and bound analysis. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 192–207. Springer, Heidelberg (2006)
29. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
30. Chen, H.Y., Flur, S., Mukhopadhyay, S.: Termination proofs for linear simple loops. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 422–438. Springer, Heidelberg (2012)
31. Dershowitz, N., Lindenstrauss, N., Sagiv, Y., Serebrenik, A.: A general framework for automatic termination analysis of logic programs. Appl. Algebra Eng. Commun. Comput. 12(1/2), 117–156 (2001)

32. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, pp. 375–385 (2009)
33. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 304–319. Springer, Heidelberg (2010)
34. Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination of imperative programs using Max-SMT. In: FMCAD, pp. 218–225 (2013)
35. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis. LNCS, vol. 7935, pp. 43–62. Springer, Heidelberg (2013)