# Bit-Precise Procedure-Modular Termination Analysis

Hong-Yi Chen, Department of Computer Science, University of Oxford, United Kingdom
Cristina David, Department of Computer Science, University of Oxford, United Kingdom
Daniel Kroening, Department of Computer Science, University of Oxford, United Kingdom
Peter Schrammel, School of Engineering and Informatics, University of Sussex, United Kingdom
Björn Wachter, SSW-Trading GmbH, Oststeinbek, Germany

Non-termination is the root cause of a variety of program bugs, such as hanging programs and denial-of-service vulnerabilities. This makes an automated analysis that can prove the absence of such bugs highly desirable. To scale termination checks to large systems, an interprocedural termination analysis seems essential. This is a largely unexplored area of research in termination analysis, where most effort has focussed on small but difficult single-procedure problems.

We present a modular termination analysis for C programs using template-based interprocedural summarisation. Our analysis combines a context-sensitive, over-approximating forward analysis with the inference of under-approximating preconditions for termination. Bit-precise termination arguments are synthesised over lexicographic linear ranking function templates. Our experimental results show the advantage of interprocedural reasoning over monolithic analysis in terms of efficiency, while retaining comparable precision.

CCS Concepts: •**Theory of computation** → **Program analysis;** *Program verification;*

## 1. INTRODUCTION

Termination bugs compromise safety-critical software systems by making them unresponsive. In particular, termination bugs can be exploited in denial-of-service attacks [CVE 2009]. Termination guarantees are therefore instrumental for ensuring software reliability. Termination provers are static analysis tools that aim to construct a proof of termination for a given input program and the implementations of these tools have made tremendous progress in the past few years. They compute proofs for complex loops that may require linear lexicographic (e.g., [Ben-Amram and Genaim 2013; Ben-Amram and Genaim 2014; Leike and Heizmann 2014]) or non-linear termination arguments (e.g., Bradley et al. [2005b]) in a completely automatic way. However, there remain major practical challenges in analysing the termination of real-world code.

First of all, as observed by Falke et al. [2012], most approaches in the literature are specialised to linear arithmetic over unbounded mathematical integers. Even though unbounded arithmetic may reflect the intuitively-expected program behaviour, the program actually executes over bounded machine integers. The semantics of C allows unsigned integers to wrap around when they over/underflow. Hence, arithmetic on $k$-bit-wide unsigned integers must be performed modulo $2^k$. According to the C standards, over/underflows of signed integers are undefined behaviour, but practically also wrap around on most architectures. Thus, accurate termination analysis requires a *bit-precise* analysis of the program semantics (i.e., an analysis that captures the semantics of programs down to each individual bit by using machine arithmetic). Tools must be

This work is supported by EPSRC EP/H017585/1, by *VeTeSS: Verification and Testing to Support Functional Safety Standards*, Artemis Joint Undertaking 295311, H2020 FET OPEN 712689 SC$^2$ and ERC project 280053 “CPROVER”.

DOI: 0000001.0000001

configurable with architectural specifications such as the width of data types and endianness. The following examples illustrate that termination behaviour on machine integers can be completely different than on mathematical integers. For example, the program fragment

```
void foo1(unsigned n)
{
  for(unsigned x=0; x<=n; x++);
}
```

does terminate with mathematical integers, but does *not* terminate with machine integers if n equals the largest unsigned integer. On the other hand, the program fragment

```
void foo2(unsigned x)
{
  while(x>=10) x++;
}
```

does not terminate with mathematical integers, but terminates with machine integers because unsigned machine integers wrap around.

A second challenge is to make termination analysis scale to larger programs. The annual Software Verification Competition (SV-COMP) [Beyer 2016] includes a division in termination analysis, which reflects a representative picture of the state of the art. The SV-COMP 2016 termination benchmarks contain challenging termination problems on smaller programs with at most 453 instructions (average 53), feature at most seven functions (average three) and four loops (average one).

In this paper, we present a technique that we have successfully run on programs that are one order of magnitude larger, containing up to 5000 instructions. Larger instances require different algorithmic techniques to scale, and we conjecture that the key technique is a modular, interprocedural analysis which decomposes a problem into sub-problems rather than a monolithic analysis which would put all its resources into solving the problem all at once.

Modular termination analysis raises several conceptual and practical challenges that do not arise in monolithic termination analysers. For example, when proving termination of a program, a possible approach is to prove that all procedures in the program terminate *universally*, i.e., in any possible calling context. However, this criterion is too optimistic, as termination of individual procedures often depends on the calling context, i.e., procedures terminate *conditionally* only in specific calling contexts.

The approach that we take is verifying universal program termination in a top-down manner by proving termination of each procedure relative to its calling contexts, and propagating upwards which calling contexts guarantee termination of the procedure. It is too difficult to determine these contexts precisely; analysers thus compute preconditions for termination. A *sufficient precondition* identifies pre-states in which the procedure will definitely terminate, and is thus suitable for proving termination. By contrast, a *necessary precondition* identifies pre-states in which the procedure may terminate. Its negation are states in which the procedure will not terminate, which is useful for proving non-termination.

In this paper we focus on the computation of sufficient preconditions for termination. Preconditions enable information reuse, and thus scalability, as it is frequently possible to avoid repeated analysis of parts of the code base, e.g., libraries whose procedures are called multiple times or parts of the code that did not undergo modifications between successive analysis runs.

**Contributions:**

(1) We propose an algorithm for *interprocedural termination analysis*. The approach is based on a template-based static analysis using SAT solving. It combines context-sensitive, summary-based interprocedural analysis with the inference of preconditions for termination based on template abstractions. Note that, while each of the components in isolation is not novel, it is non-trivial to combine them in a way that achieves both scalability and precision. We focus on non-recursive programs, which cover a large portion of software written, especially in domains such as embedded systems.
(2) We provide an implementation of the approach in 2LS, a static analysis tool for C programs [Schrammel and Kroening 2016]. Our instantiation of the algorithm uses template polyhedra and lexicographic, linear ranking functions templates. The analysis is bit-precise and purely relies on SAT-solving techniques.
(3) We report the results of an experimental evaluation on 597 procedural SV-COMP benchmarks with 1100 to 5700 lines of code (2705 on average), 33 to 136 procedures (67 on average), and four to ten loops (5.5 on average), thus demonstrating the scalability and applicability of the approach to programs with thousands of lines of code.

This article is a thoroughly revised version of the paper [Chen et al. 2015], extended with the lexicographic ranking function synthesis algorithm, more detailed examples, proofs, illustrations (cf. [Schrammel 2016]), and an experimental comparison with the most recent version of the available termination analysers.

## 2. PRELIMINARIES

In this section, we introduce basic notions of interprocedural termination analysis, template abstract domains and their usage to reduce second-order problems to first-order ones.

### 2.1. Program model and notation

We consider non-recursive programs with multiple procedures. We assume that programs are given in terms of acyclic call graphs, where individual procedures $f$ are given in terms of symbolic input/output transition systems. Formally, the input/output transition system of a procedure $f$ is a triple $(Init_f, Trans_f, Out_f)$, where $Trans_f(\boldsymbol{x}, \boldsymbol{x}')$ is the transition relation; the input relation $Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s)$ defines the initial states $\boldsymbol{x}^s$ of the transition system and relates them to the inputs $\boldsymbol{x}^{in}$; the output relation $Out_f(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}^{out})$ connects the transition system to the outputs $\boldsymbol{x}^{out}$ of the procedure. Inputs are procedure parameters, global variables, and memory objects that are read by $f$. Outputs are return values, and potential side effects such as global variables and memory objects written by $f$. Internal states $\boldsymbol{x}$ are commonly the values of variables at the loop heads in $f$.

These relations are given *as first-order logic formulae* resulting from the logical encoding of the program semantics (using e.g. bitvector and array theories). For illustration, Figure 2 gives the encoding of the two procedures in Figure 1 into such formulae.

When mapping back to the notation above, the input $\boldsymbol{x}^{in}$ of $f$ is $z^{in}$ and the output $\boldsymbol{x}^{out}$ consists of the return value denoted $w^{out}$. The transition relation of $h$ encodes the loop over the internal state variables $(x, y)$. We may need to introduce Boolean variables $g_i$ to model the control flow. Multiple and nested loops can be similarly encoded in the relation $Trans$.

Note that we view these formulae as predicates, e.g., $Trans(\boldsymbol{x}, \boldsymbol{x}')$, with given parameters $\boldsymbol{x}, \boldsymbol{x}'$, and mean the substitution $Trans[\boldsymbol{a}/\boldsymbol{x}, \boldsymbol{b}/\boldsymbol{x}']$ when we write $Trans(\boldsymbol{a}, \boldsymbol{b})$. We omit the parameters and simply write $Trans$ when these are not relevant or clear

```
unsigned h(unsigned y) {
  unsigned x;
  for(x=0; x<10; x+=y);
  return x;
}
```

```
unsigned f(unsigned z) {
  unsigned w;
  w=h(z/2+1);
  return w;
}
```

Fig. 1: Example for conditional termination

$$\begin{array}{rcl} \mathit{Init}_h((y^{in}),(x^s,y^s)) & \equiv & (x^s{=}0 \wedge y^s{=}y^{in}) \\ \mathit{Trans}_h((x,y),(x',y')) & \equiv & (x'{=}x{+}y \wedge\ x{<}10 \wedge y'{=}y) \\ \mathit{Out}_h((x^s,y^s),(x,y),(x^{out})) & \equiv & (x^{out}{=}x \wedge \neg(x{<}10)) \end{array}$$

$$\begin{array}{rcl} \mathit{Init}_f((z^{in}),(z^s)) & \equiv & (z^s{=}z^{in}) \\ \mathit{Trans}_f((z),(z')) & \equiv & \mathit{true} \\ \mathit{Out}_f((z^s),(z),(w^{out})) & \equiv & (p^{in}{=}z^s/2{+}1 \wedge h_0((p_{in}),(p_{out})) \wedge w^{out}{=}p_{out}) \end{array}$$

Fig. 2: Encoding of the functions h and f from Example 1

from the context. Moreover, we write $\boldsymbol{x}$ and $x$ with the understanding that the former is a vector, whereas the latter is a scalar.

Each call to a procedure $h$ at call site $i$ in a procedure $f$ is modelled by a *placeholder predicate* $h_i(\boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}})$ occurring in the formulae $\mathit{Init}_f$, $\mathit{Trans}_f$, and $\mathit{Out}_f$ for $f$. The placeholder predicate ranges over intermediate variables representing its actual input and output parameters $\boldsymbol{x}_i^{p_{in}}$ and $\boldsymbol{x}_i^{p_{out}}$, respectively. Placeholder predicates evaluate to $true$, which corresponds to havocking procedure calls. In procedure $f$ in Figure 2, the placeholder for the procedure call to $h$ is $h_0((p_{in}),(p_{out}))$ with the actual input and output parameters $p_{in}$ and $p_{out}$, respectively.

To give a further example, the procedures in Figure 3 can be encoded as follows: Procedure inc:

$$\begin{array}{rcl} \mathit{Init}((i^{in}),(i^s)) & = & (i^{in}{=}i^s) \\ \mathit{Trans}((i),(i')) & = & \mathit{true} \\ \mathit{Out}((i^s),(i),(i^{out})) & = & (i^{out}{=}i^s{+}1) \end{array}$$

Procedure inc_if_pos:

$$\begin{array}{rcl} \mathit{Init}((x^{in},y^{in}),(x^s,y^s)) & = & (x^s{=}x^{in} \wedge y^s{=}y^{in}) \\ \mathit{Trans}((x,y),(x',y')) & = & \mathit{true} \\ \mathit{Out}((x^s,y^s),(x,y),(x^{out})) & = & (g_0{=}(y^s{>}0.0) \wedge \mathit{inc}_0((x^s),(x_0)) \wedge x^{out}{=}(g_0?x_0{:}x^s)) \end{array}$$

Note that we encode the control flow join after the **if** statement using the conditional operator[1] on the **if**'s condition $g_0$. Since the procedure call to inc is inside the **if** with condition $g_0$, we call $g_0$ the *guard* of procedure inc. More details on the program representation can be found in the elaborate example containing nested loops in Section 4.3, as well as in [Brain et al. 2015].

## 2.2. Interprocedural Analyses

We introduce the notation for the basic concepts of interprocedural analyses.

*Definition* 2.1 (*Invariants, Summaries, Calling Contexts*). For a procedure given by the triplet $(\mathit{Init}, \mathit{Trans}, \mathit{Out})$ we define:

[1]The conditional operator $c?a{:}b$ returns $a$ if $c$ evaluates to $true$, and $b$ otherwise.

```
int inc(int i) {
  return i+1;
}
```

```
int inc_if_pos(int x, float y) {
  if(y>0.0f) {
    x=inc(x);
  }
  return x;
}
```

Fig. 3: Example for encoding procedures

— An *invariant* is a predicate $\mathit{Inv}$ such that:

$$\forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : \big(\mathit{Init}(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow \mathit{Inv}(\boldsymbol{x}^s, \boldsymbol{x}^s)\big) \wedge \big(\mathit{Inv}(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathit{Inv}(\boldsymbol{x}^s, \boldsymbol{x}')\big)$$

— Given an invariant $\mathit{Inv}$, a *summary* is a predicate $\mathit{Sum}$ such that:

$$\forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}^{out} : \mathit{Init}(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \wedge \mathit{Inv}(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \mathit{Out}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}^{out}) \Longrightarrow \mathit{Sum}(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})$$

— Given an invariant $\mathit{Inv}$, the *calling context* for a procedure call $h$ at call site $i$ in the given procedure is a predicate $\mathit{CallCtx}_{h_i}$ such that

$$\forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}} : \mathit{Init}(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \wedge \mathit{Inv}(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \wedge \mathit{Out}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}^{out}) \Longrightarrow \mathit{CallCtx}_{h_i}(\boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}})$$

These concepts have the following roles: Invariants abstract the effect of loops on variables. Summaries abstract the behaviour of called procedures; they are used to strengthen the placeholder predicates. Calling contexts abstract the caller's behaviour w.r.t. the procedure being called, i.e., they are an assumption about the callee's execution environment in the sense of assume-guarantee reasoning [Grumberg and Long 1994]. When analysing the callee, these calling contexts are used to constrain its inputs and outputs.

We give a short example to illustrate the $\mathit{Inv}$ and $\mathit{Sum}$ notions.

```
int add(int x, int y)
{
  for(int i=0; i<y; i++)
    x++;
  return x;
}
```

In terms of our program model, we can encode this procedure as follows.

$$\mathit{Init}((x^{in}, y^{in}), (x^s, y^s, i^s)) = (x^s = x^{in} \wedge y^s = y^{in} \wedge i^s = 0)$$
$$\mathit{Trans}((x, y, i), (x', y', i')) = (i < y \wedge i' = i + 1 \wedge x' = x + 1 \wedge y' = y)$$
$$\mathit{Out}((x^s, y^s, i^s), (x, y, i), (x^{out})) = (x^{out} = x \wedge \neg(i < y))$$

Our definition of $\mathit{Inv}$ allows us to relate the current state $(x, y, i)$ inside the loop with the state $(x^s, y^s, i^s)$ at loop entry. $\mathit{Inv}((x^s, y^s, i^s), (x, y, i)) = (0 \leq i \leq y \wedge x = x^s + i)$ satisfies the definition above and we can use it to derive the summary $\mathit{Sum}((x^{in}, y^{in}), (x^{out})) = (x^{out} = x^{in} + y^{in})$. In Section 3 we will discuss these notions in more details on the program in Figure 1.

Since we want to reason about termination, we need the notions of ranking functions and preconditions for termination.

*Definition* 2.2 (*Ranking function*). A *ranking function* for a procedure $(Init,\ Trans,\ Out)$ is a function $r$ such that

$$\begin{array}{l}\exists \Delta > 0, Inv : \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : \\ \quad \big(Init(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow Inv(\boldsymbol{x}^s, \boldsymbol{x}^s)\big) \\ \wedge\ \ (Inv(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv(\boldsymbol{x}^s, \boldsymbol{x}') \wedge r(\boldsymbol{x}) - r(\boldsymbol{x}') > \Delta \wedge r(\boldsymbol{x}) > 0)\end{array}$$

Thus, $r$ is a function from the set of program states to a well-founded domain, e.g. $\mathbb{R}^{\geq 0}$.

We denote by $RR(\boldsymbol{x}, \boldsymbol{x}')$ the constraints on $r$ that form the *termination argument*, i.e., $r(\boldsymbol{x}) - r(\boldsymbol{x}') > \Delta \wedge r(\boldsymbol{x}) > 0$ for monolithic ranking functions. The existence of a ranking function for a procedure guarantees its *universal* termination.

The weakest termination precondition for a procedure describes the inputs for which it terminates. If it is $true$, the procedure terminates universally; if it is $false$, then it will not terminate for any input. Since the weakest precondition is intractable to compute or even uncomputable, we under-approximate[2] the precondition. Thus, we compute a *sufficient precondition* denoting a subset of the terminating inputs. Consequently, a sufficient precondition for termination guarantees that the program terminates for all $\boldsymbol{x}^{in}$ that satisfy it.

*Definition* 2.3 (*Precondition for termination*). Given a procedure $(Init,\ Trans,\ Out)$, a sufficient *precondition for termination* is a predicate $Precond$ such that

$$\begin{array}{l}\exists RR, Inv : \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : \\ \quad \big(Precond(\boldsymbol{x}^{in}) \wedge Init(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow Inv(\boldsymbol{x}^s, \boldsymbol{x}^s)\big) \\ \wedge\ \ (Inv(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv(\boldsymbol{x}^s, \boldsymbol{x}') \wedge RR(\boldsymbol{x}, \boldsymbol{x}'))\ .\end{array}$$

Note that $false$ is always a trivial model for $Precond$, but not a very useful one.

A sufficient precondition for termination *under-approximates* the weakest precondition, i.e., it contains only inputs for which the program terminates, but not all of them. The complement of the sufficient precondition *over-approximates* the inputs for which the program does not terminate, i.e., it contains all inputs for which the program does not terminate, but it also contains some inputs for which the program terminates.

## 2.3. Reduction from Second-order to First-order Problems

Our approach requires us to effectively solve second-order problems. We achieve this by reducing them to first-order by restricting the space of solutions to expressions of the form $\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$ where

— $\boldsymbol{d}$ are parameters to be instantiated with concrete values and $\boldsymbol{x}^s, \boldsymbol{x}$ are vectors of program variables.
— $\mathcal{T}$ is a template that gives a blueprint for the shape of the formulas to be computed.

Choosing a template is analogous to choosing an abstract domain in abstract interpretation [Cousot and Cousot 1977]. To allow for a flexible choice, we consider *template polyhedra* [Sankaranarayanan et al. 2005]. Polyhedral templates subsume intervals, zones and octagons [Miné 2006].

We now state a soundness result (where, given a formula $\sigma$ over existential variables $\boldsymbol{d}$, $\exists \boldsymbol{d}.\sigma(\boldsymbol{d})$, we call $\boldsymbol{d}$ a satisfying model of $\sigma$ if $\boldsymbol{d} \models \sigma(\boldsymbol{d})$):

[2]Given two formulae $A$ and $B$, we say that $A$ is an under-approximation of $B$ if $A \rightarrow B$. Conversely, we call $B$ an over-approximation of $A$.

THEOREM 2.4. *Any satisfying model* $\boldsymbol{d}$ *of the reduction of the second-order constraint for invariants in Definition 2.1 using template* $\mathcal{T}$

$$\begin{aligned}\exists \boldsymbol{d}, \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : \ & \big(Init(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow \mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})\big) \\ \wedge \ & \big(\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{d})\big)\end{aligned}$$

*satisfies* $\exists Inv : \forall \boldsymbol{x}^s, \boldsymbol{x} : Inv(\boldsymbol{x}^s, \boldsymbol{x}) \Longrightarrow \mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$*, i.e.,* $\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$ *is a sound over-approximating invariant. Similar soundness results hold true for summaries and calling contexts.*

PROOF SKETCH. Let $P_{Inv}$ be the set of all first-order formulae over variables $\boldsymbol{x}^s, \boldsymbol{x}$. Let $D$ be the domain of $\boldsymbol{d}$. Then $P_{\mathcal{T}} =_{def} \{\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d}) | \boldsymbol{d} \in D\}$ is the set of first-order formulae that $\mathcal{T}$ can describe. Obviously, we have $P_{\mathcal{T}} \subseteq P_{Inv}$. Assume that $\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$ is *not* a sound over-approximating invariant, i.e., $\neg\exists Inv : \forall \boldsymbol{x}^s, \boldsymbol{x} : Inv(\boldsymbol{x}^s, \boldsymbol{x}) \Longrightarrow \mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$, then $\forall Inv : \exists \boldsymbol{x}^s, \boldsymbol{x} : Inv(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \neg\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$. However, since $P_{\mathcal{T}} \subseteq P_{Inv}$ there must be a $Inv \in P_{Inv}$ such that $Inv$ equals $\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})$, which contradicts the assumption. □

In the above, we call $\boldsymbol{d}$ a satisfying model to mean that $\boldsymbol{d} \models (\forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : Init(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow \mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d})) \wedge (\mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{d}) \wedge Trans(\boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathcal{T}(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{d}))$. We will consider the same meaning whenever using the term "satisfying model" in the rest of the paper (for both first-order and second-order existentially quantified entities).

Similar approaches for template-based synthesis have been described, for instance, by Gawlitza and Seidl [2007], Gulwani et al. [2008], and Li et al. [2014]. However, these methods consider programs over mathematical integers.

Note that the second-order logic formulae are independent of specific theories. In the implementation, they are solved by a series of first-order formulae using quantifier-free bitvectors and array theory. The corresponding synthesis algorithms have been presented by Brain et al. [2015].

## 3. OVERVIEW OF THE APPROACH

In this section, we introduce the architecture of our interprocedural termination analysis. Our analysis combines, in a non-trivial synergistic way, the inference of invariants, summaries, calling contexts, termination arguments, and preconditions, which have a concise characterisation in second-order logic (see Definitions 2.1, and 2.3). At the lowest level our approach relies on the solver back-end for second-order problems described in Section 2.3. Details about the exact constraints solved by the back-end will be given in Section 5.

To see how the different analysis components fit together, we now go through the pseudo-code of our termination analyser (Algorithm 1). We use three global maps $Sums^o$, $Invs^o$, and $Preconds^u$ to store the summaries, invariants and preconditions that we compute for procedures $f$ of the program. Function $analyze$ is given the entry procedure $f_{entry}$ of the program as argument and proceeds in two analysis phases.

Phase one is an *over-approximate* forward analysis, given in subroutine $analyzeForward$, which recursively descends into the call graph from the entry point $f_{entry}$. Subroutine $analyzeForward$ infers for each procedure call in $f$ an over-approximating calling context $CallCtx^o$, using procedure summaries and other previously-computed information. Before analyzing a callee, the analysis checks if the callee has already been analysed and whether the stored summary can be re-used, i.e., if it is compatible with the new calling context $CallCtx^o$. Finally, once summaries for all callees are available, the analysis infers loop invariants and a summary for $f$ itself, which are stored for later re-use.

The second phase is an *under-approximate* backward analysis, subroutine $analyzeBackward$, which infers termination preconditions. Again, we recursively descend into the call graph. Analogous to the forward analysis, we infer for each procedure

**Algorithm 1:** *analyze*

```
1  global $Sums^o, Invs^o, Preconds^u$;
2  function $analyzeForward(f, CallCtx^o_f)$
3      foreach procedure call $h$ in $f$ do
4          $CallCtx^o_h = compCallCtx^o(f, CallCtx^o_f, h)$;
5          if $needToReAnalyze^o(h, CallCtx^o_h)$ then
6              $analyzeForward(h, CallCtx^o_h)$;
7      $join^o((Sums^o[f], Invs^o[f]), compInvSum^o(f, CallCtx^o_f))$
8  function $analyzeBackward(f, CallCtx^u_f)$
9      $termConds = CallCtx^u_f$;
10     foreach procedure call $h$ in $f$ do
11         $CallCtx^u_h = compCallCtx^u(f, CallCtx^u_f, h)$;
12         if $needToReAnalyze^u(h, CallCtx^u_h)$ then
13             $analyzeBackward(h, CallCtx^u_h)$;
14         $termConds \leftarrow termConds \wedge Preconds^u[h]$;
15     $join^u(Preconds^u[f],$
               $compPrecondTerm(f, Invs^o[f], termConds)$;
16 function $analyze(f_{entry})$
17     $analyzeForward(f_{entry}, true)$;
18     $analyzeBackward(f_{entry}, true)$;
19     return $Preconds^u[f_{entry}]$;
```

call in $f$ an under-approximating calling context $CallCtx^u$ and recur only if necessary (Line 12). Finally, we compute the under-approximating precondition for termination (Line 15). This precondition is inferred w.r.t. the termination conditions that have been collected: the backward calling context (Line 9), the preconditions for termination of the callees (Line 14), and the termination arguments for $f$ itself. Note that superscripts $o$ and $u$ in predicate symbols indicate over- and under-approximation, respectively.

We explain the subroutines $join^o$, $join^u$, $needToReAnalyze^o$, and $needToReAnalyze^u$ in Section 5.3.

**Challenges.** Our algorithm uses over- and under-approximation in a systematic way in order to address the challenging problem of finding meaningful preconditions by a context-sensitive interprocedural termination analysis.

— The precondition in Definition 2.3 admits the trivial solution *false* for $Precond$. How do we find a good candidate? To this end, we "bootstrap" the process with a candidate precondition: a single value of $\boldsymbol{x}^{in}$, for which we compute a termination argument. The key observation is that the resulting termination argument is typically more general, i.e., it shows termination for many further entry states. The more general precondition is then computed by precondition inference w.r.t. the termination argument.

— A second challenge is to compute under-approximations. Obviously, the predicates in the definitions in Section 2 can be over-approximated using abstract domains such as intervals. However, there are only few methods for under-approximating analysis. In this work, we use a method similar to Cook et al. [2008] to obtain under-approximating preconditions w.r.t. property $p$: we infer an over-approximating precondition w.r.t. $\neg p$ and negate the result. In our case, $p$ is the termination condition $termConds$.

*Example* 3.1. We illustrate the algorithm on the simple example given in Figure 1 with the encoding given in Figure 2. The entry procedure `f` calls procedure `h`. Procedure `h` terminates if and only if its argument `y` is non-zero, i.e., procedure `h` only terminates conditionally. The termination of procedure `f` depends on the argument passed in the call to `h`. For instance, passing an argument strictly greater than 0 guarantees universal termination of procedure `f`.

Let us assume that unsigned integers are 32 bits wide, i.e., they range from 0 to $M$ (with $M = 2^{32}-1$). We use the interval abstract domain for invariant, summary and precondition inference, but the abstract domain with the elements $\{true, false\}$ that can only express reachability of a procedure call for calling contexts. Thus, $true$ means that the call is reachable, and $false$ means that it is unreachable.

Our algorithm proceeds as follows. The first phase is $analyzeForward$, which starts from the entry procedure `f`. By descending into the call graph, we must compute an over-approximating calling context $CallCtx^o_h((p_{in}), (p_{out}))$ for procedure `h` for which no calling context has been computed before. This calling context is $true$ because the call is reachable. Hence, we recursively analyse `h`. Given that `h` does not contain any procedure calls, we compute the over-approximating summary $Sum^o_h((y^{in}), (x^{out})) = (0 \le y^{in} \le M \wedge 0 \le x^{out} \le M)$ and invariant $Inv^o_h((x^s, y^s), (x, y))) = (0 \le x \le M \wedge 0 \le y \le M)$. Now, this information can be used in order to compute $Sum^o_f((z^{in}), (w^{out})) = (0 \le z^{in} \le M \wedge 0 \le w^{out} \le M)$ and invariant $Inv^o_f((z^s), (z)) = true$ for the entry procedure `f`. As explained in Section 2.3, we use synthesis techniques that iteratively call an Satisfiability Modulo Theories (SMT) solver to optimise values of template parameters in order to compute these predicates.

The backwards analysis starts again from the entry procedure `f`. It computes an under-approximating calling context $CallCtx^u_h((p_{in}), (p_{out}))$ for procedure `h`, which is $true$ (because `h` is always backwards reachable from `f`'s exit point), before descending into the call graph. It then computes an under-approximating precondition for termination $Precond^u_h((y^{in})) = (1 \le y^{in} \le M)$ or, more precisely, an under-approximating summary whose projection onto the input variables of `h` is the precondition $Precond^u_h$. By applying this summary at the call site of `h` in `f`, we can now compute the precondition for termination $Precond^u_f((z^{in})) = (0 \le z^{in} \le M)$ of `f`, which proves universal termination of `f`.

We illustrate the effect of the choice of the abstract domain on the analysis of the example program. Assume we replace the $\{true, false\}$ domain by the interval domain. In this case, $analyzeForward$ computes[3]

$$CallCtx^o_h((p_{in}), (p_{out})) = (1 \le p_{in} \le 2^{31} \wedge 0 \le p_{out} \le M)$$

The calling context is computed over the actual parameters $p_{in}$ and $p_{out}$. It is renamed to the formal parameters $y^{in}$ and $x^{out}$ (the return value) when $CallCtx^o_h$ is used for constraining the pre/postconditions in the analysis of `h`. Subsequently, $analyzeBackward$ computes the precondition for termination of `h` using the union of all calling contexts in the program. Since `h` terminates unconditionally in these calling contexts, we trivially obtain $Precond^u_h((y^{in})) = (1 \le y^{in} \le 2^{31})$, which in turn proves universal termination of `f`.

## 4. INTRAPROCEDURAL TERMINATION ANALYSIS

In this section, we explain the termination analysis for one procedure. The results in this section will be used in Section 5 for the interprocedural termination analysis.

[3]Note that $\lfloor \frac{M}{2} \rfloor + 1 = 2^{31}$.

### 4.1. Lexicographic Ranking Functions

Monolithic ranking functions are complete, i.e., termination can always be proven monolithically if a program terminates. However, in practice, combinations of linear ranking functions, e.g., linear lexicographic functions [Bradley et al. 2005a; Cook et al. 2013] are preferred. This is driven by the fact that monolithic *linear* ranking functions are not expressive enough, and that *non-linear* theories are challenging for the existing SMT solvers, which handle the linear case much more efficiently.

*Definition* 4.1 (*Lexicographic ranking function*). A lexicographic ranking function $R$ for a transition relation $\mathit{Trans}(\boldsymbol{x},\boldsymbol{x}')$ is an $n$-tuple of expressions $(R_n, R_{n-1}, \ldots, R_1)$ such that

$$\begin{array}{lr}
\exists\Delta > 0, \mathit{Inv} : \forall \boldsymbol{x}^{in}, \boldsymbol{x}^{s}, \boldsymbol{x}, \boldsymbol{x}' : & \\
\quad \big(\mathit{Init}(\boldsymbol{x}^{in}, \boldsymbol{x}^{s}) \Longrightarrow \mathit{Inv}(\boldsymbol{x}^{s}, \boldsymbol{x}^{s})\big) & \\
\wedge \ \big(\mathit{Inv}(\boldsymbol{x}^{s}, \boldsymbol{x}) \wedge \mathit{Trans}(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathit{Inv}(\boldsymbol{x}^{s}, \boldsymbol{x}') \wedge \exists i \in [1,n] : & \\
\qquad\quad R_i(\boldsymbol{x}) > 0 & \text{(Bounded)} \\
\qquad \wedge \ R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > \Delta & \text{(Decreasing)} \\
\qquad \wedge \ \forall j > i : R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0 \,\big) & \text{(Non-increasing)}
\end{array}$$

Notice that this is a special case of Definition 2.2. In particular, the existence of $\Delta > 0$ and the *Bounded* condition guarantee that $>$ is a well-founded relation.

Before we encode the requirements for lexicographic ranking functions into constraints, we need to optimise them to take advantage of bit-vector semantics. Since bit-vectors are bounded, it follows that the *Bounded* condition is trivially satisfied and therefore can be omitted. Moreover, bit-vectors are discrete, hence we can replace the *Decreasing* condition with $R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > 0$. The following formula, $LR^n$, holds if and only if $(R_n, R_{n-1}, \ldots, R_1)$ is a lexicographic ranking function with $n$ components over bit-vectors.

$$LR^n(\boldsymbol{x},\boldsymbol{x}') = \bigvee_{i=1}^{n} \left( R_i(\boldsymbol{x}) - R_i(\boldsymbol{x}') > 0 \ \wedge \ \bigwedge_{j=i+1}^{n} (R_j(\boldsymbol{x}) - R_j(\boldsymbol{x}') \geq 0) \right)$$

Assume we are given the transition relation $\mathit{Trans}(\boldsymbol{x},\boldsymbol{x}')$ of a procedure $f$. The procedure $f$ may be composed of several loops, and each of the loops is associated with guards $g$ (and $g'$) that express the reachability of the loop head (and the end of the loop body, respectively; see Section 2.3). That is, suppose $f$ has $k$ loops and $n_i$ denotes the number of lexicographic components for loop $i$, then the termination argument to prove termination of $f$ takes the form:

$$RR^{\boldsymbol{n}}(\boldsymbol{x},\boldsymbol{x}') = \bigwedge_{i=1}^{k} g_i \wedge g_i' \Longrightarrow LR_i^{n_i}(\boldsymbol{x},\boldsymbol{x}')$$

### 4.2. Synthesising Lexicographic Ranking Functions

In this section, we show how to compute lexicographic ranking functions. While ranking techniques for mathematical integers use e.g., Farkas' Lemma, this is not applicable to bitvector operations. Thus, we use a synthesis approach (like the TAN tool by Kroening et al. [2010]) and extend it from monolithic to lexicographic ranking functions.

We consider the class of lexicographic ranking functions generated by the template where $R_i(\boldsymbol{x})$ is the product $\boldsymbol{\ell}_i\boldsymbol{x}$ with the row vector $\boldsymbol{\ell}_i$ of template parameters. We denote the resulting constraints for loop $i$ as $\mathcal{LR}_i^{n_i}(\boldsymbol{x},\boldsymbol{x}',L_i^{n_i})$, where $L_i^{n_i}$ is the vector $(\boldsymbol{\ell}_i^1, \ldots, \boldsymbol{\ell}_i^{n_i})$. The constraints for the ranking functions of a whole procedure are $\mathcal{RR}(\boldsymbol{x},\boldsymbol{x}',\boldsymbol{L^n})$, where $\boldsymbol{L^n}$ is the vector $(L_1^{n_1}, \ldots, L_k^{n_k})$.

Putting all this together, we obtain the following reduction of ranking function synthesis to a first-order quantifier elimination problem over templates:

$$\exists \boldsymbol{L^n} : \forall \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : Inv(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathcal{RR}(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{L^n})$$

The parameters $L_i^{n_i}$ are signed bitvectors extended by the special value $\top$ in order to complete the lattice of ranking constraints $\mathcal{LR}_i^{n_i}$. We define $\mathcal{LR}_i^{n_i}(\boldsymbol{x}, \boldsymbol{x}', \top) \equiv true$ indicating that no ranking function has been found for the given template ("don't know"). We write $\bot$ for the equivalence class of bottom elements for which $\mathcal{LR}_i^{n_i}(\boldsymbol{x}, \boldsymbol{x}', L_i^{n_i})$ evaluates to *false*, meaning that the ranking function has not yet been computed. For example, $\boldsymbol{0}$ is a bottom element. Note that this intuitively corresponds to the meaning of $\bot$ and $\top$ as known from invariant inference by abstract interpretation (see Section 2.3).

```
Algorithm 2: compTermArg
Input: procedure $f$ with invariant $Inv$, a bound on the number of lexicographic
       components $N$
Output: ranking constraint $\mathcal{RR}$
1  $\boldsymbol{n} \leftarrow \boldsymbol{1}^k$; $\boldsymbol{\Lambda^n} \leftarrow \bot^k$; $\boldsymbol{M} \leftarrow \emptyset^k$;
2  let $\varphi = Inv(\boldsymbol{x}) \wedge Trans(\boldsymbol{x}, \boldsymbol{x}')$;
3  while $true$ do
4      let $\psi = \varphi \wedge \neg\mathcal{RR}(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{\Lambda^n})$;
5      solve $\psi$ for $\boldsymbol{x}, \boldsymbol{x}'$;
6      if UNSAT then return $\mathcal{RR}(\boldsymbol{x}, \boldsymbol{x}', \boldsymbol{\Lambda^n})$;
7      let $(\boldsymbol{\chi}, \boldsymbol{\chi}')$ be a model of $\psi$;
8      let $i \in \{i \mid \neg(g_i \wedge g_i' \Rightarrow \mathcal{LR}_i^{n_i}(\boldsymbol{\chi}, \boldsymbol{\chi}', \Lambda_i^{n_i}))\}$;
9      $M_i \leftarrow M_i \cup \{(\boldsymbol{\chi}, \boldsymbol{\chi}')\}$;
10     let $\theta = \bigwedge_{(\boldsymbol{\chi}, \boldsymbol{\chi}') \in M_i} \mathcal{LR}_i^{n_i}(\boldsymbol{\chi}, \boldsymbol{\chi}', L_i^{n_i})$;
11     solve $\theta$ for $L_i^{n_i}$;
12     if UNSAT then
13         if $n_i < N$ then $n_i \leftarrow n_i + 1$; $\Lambda_i^{n_i} = \bot$; $M_i = \emptyset$ ;
14         else return $\mathcal{RR}(\boldsymbol{x}, \boldsymbol{x}', \top^k)$;
15     else
16         let $m$ be a model of $\theta$;
17         $\Lambda_i^{n_i} \leftarrow m$;
```

We now use the example in Figure 4 to walk through Algorithm 2. The left-hand side of Figure 4 is the C code and the right-hand side is its transition relation.

The ranking template with a single component is of the form

$$x > 0 \Longrightarrow (\ell_x^1 x + \ell_y^1 y) - (\ell_x^1 x' + \ell_y^1 y') > 0$$

with $\boldsymbol{L}^{(1)} = ((\ell_x^1, \ell_y^1))$. The ranking template with two components is of the form

$$\begin{aligned} x > 0 \Longrightarrow \big( & (\ell_x^1 x + \ell_y^1 y) - (\ell_x^1 x' + \ell_y^1 y') > 0 \vee \\ & (\ell_x^2 x + \ell_y^2 y) - (\ell_x^2 x' + \ell_y^2 y') > 0 \wedge (\ell_x^1 x + \ell_y^1 y) - (\ell_x^1 x' + \ell_y^1 y') \geq 0 \big) \end{aligned}$$

with $\boldsymbol{L}^{(2)} = ((\ell_x^1, \ell_y^1), (\ell_x^2, \ell_y^2))$. Since the procedure only has a single loop, we will omit the guard ($g = true$). Also, we assume that we have obtained the invariant $Inv((x^s, y^s), (x, y)) = true$. We use Latin letters such as $x$ to denote variables and Greek letters such as $\chi$ to denote the values of these variables.

```
int x=1, y=1;
while(x>0) {
  if(y<10) x=nondet();
  else x--;
  if(y<100) y++;
}
```

$$\begin{array}{l} \mathit{Trans}((x,y),(x',y')) = \\ x{>}0 \Rightarrow \big(\ (y{\geq}10 \ \Rightarrow x'{=}x{-}1) \\ \qquad \wedge (y{<}100 \Rightarrow y'{=}y{+}1) \\ \qquad \wedge (y{\geq}100 \Rightarrow y' = y)\ \ \big) \wedge \\ x{\leq}0 \Rightarrow \big(\ \ x' = x \ \wedge y' = y \ \ \big) \end{array}$$

Fig. 4: Example for Algorithm 2 (with simplified $\mathit{Trans}$)

In each iteration, our algorithm checks the validity of the current ranking function candidate. If it is not yet a valid ranking function, the SMT solver returns a counterexample transition. Then, a new ranking function candidate that satisfies all previously observed counterexample transitions is computed. This process is guaranteed to terminate because of the finiteness of the state space. However, with a few heuristic optimisations (see Section 6), it terminates after a few iterations in practice.

We start from the bottom element for ranking functions with a single component (Line 1) and solve the corresponding formula $\psi$, which is $x > 0 \wedge \mathit{Trans}((x, y), (x', y')) \wedge \neg \mathit{false}$ (Line 5). The formula $\psi$ is satisfiable with the model $(1, 99, 0, 100)$ for $(x, y, x', y')$, for instance. This model entails the constraint $(1\ell_x^1 + 99\ell_y^1) - (0\ell_x^1 + 100\ell_y^1) > 0$, i.e., we have $\ell_x^1 - \ell_y^1 > 0$ in Line 10, from which we compute values for the template coefficients $\ell_x$ and $\ell_y$. This formula is given to the solver (Line 11) which reports SAT with the model $(1, 0)$ for $(\ell_x, \ell_y)$, for example. We use this model to update the vector of template parameter values $\Lambda_1^1$ to $(1, 0)$ (Line 17), which corresponds to the ranking function $x$.

We then continue with the next loop iteration, and check the current ranking function (Line 5). The formula $x > 0 \wedge \mathit{Trans} \wedge \neg(x - x' > 0)$ is satisfiable by the model $(1, 1, 1001, 2)$ for $(x, y, x', y')$, for instance. This model entails the constraint $(\ell_x^1 + \ell_y^1) - (1001\ell_x^1 + 2\ell_y^1) > 0$, i.e., $-1000\ell_x^1 - \ell_y^1 > 0$ in Line 10, which is conjoined with the constraint $\ell_x^1 - \ell_y^1 > 0$ from the previous iteration. The solver (Line 11) will tell us that this is UNSAT.

Since we did not find a ranking function we add another component to the lexicographic ranking function template (Line 13), and try to solve again (Line 5) and the solver might again return the model $(1, 1, 1001, 2)$ for $(x, y, x', y')$, for instance. Then in Line 11, we have to check the satisfiability of the constraint $(\ell_x^1 + \ell_y^1) - (1001\ell_x^1 + 2\ell_y^1) > 0 \vee (\ell_x^2 + \ell_y^2) - (1001\ell_x^2 + 2\ell_y^2) > 0 \wedge (\ell_x^1 + \ell_y^1) - (1001\ell_x^1 + 2\ell_y^1) \geq 0$, which simplifies to $-1000\ell_x^1 - \ell_y^1 > 0 \vee -1000\ell_x^2 - \ell_y^2 > 0 \wedge -1000\ell_x^1 - \ell_y^1 \geq 0$. The solver might report that the model $(0, -1, 0, -1)$ for $(\ell_x^2, \ell_y^2, \ell_x^1, \ell_y^1)$ satisfies the constraints. We use this model to update the ranking function to $(-y, -y)$.

We then continue with the next loop iteration, and check the current ranking function (Line 5). The formula $x > 0 \wedge \mathit{Trans} \wedge \neg(-y + y' > 0)$ is satisfiable by the model $(1, 100, 0, 100)$ for $(x, y, x', y')$, for instance. We conjoin the constraint $(\ell_x^1 + 100\ell_y^1) - (0\ell_x^1 + 100\ell_y^1) > 0 \vee (\ell_x^2 + 100\ell_y^2) - (0\ell_x^2 + 100\ell_y^2) > 0 \wedge (\ell_x^1 + 100\ell_y^1) - (0\ell_x^1 + 100\ell_y^1) \geq 0$, simplified $\ell_x^1 > 0 \vee \ell_x^2 > 0 \wedge \ell_x^1 \geq 0$, with the constraint from the previous iteration $-1000\ell_x^1 - \ell_y^1 > 0 \vee -1000\ell_x^2 - \ell_y^2 > 0 \wedge -1000\ell_x^1 - \ell_y^1 \geq 0$, and solve it. The solver might report that the model $(0, -1, 1, 0)$ for $(\ell_x^2, \ell_y^2, \ell_x^1, \ell_y^1)$ satisfies the constraints. We use this model to update the ranking function to $(-y, x)$.

Finally, we check whether there is another model for $(-y, x)$ not being a ranking function (Line 5), but this time the solver reports the formula to be UNSAT and the algorithm terminates, returning the ranking function $(-y, x)$.

### 4.3. A Worked Example: Bubble Sort

In this section, we illustrate the intraprocedural analysis on an example with nested loops by using the Bubble Sort procedure. The procedure below takes an array a and the size n of the array. Note that memory safety is an orthogonal issue. Therefore we can assume that n indeed corresponds to the actual size of a. Moreover, we make use of `__CPROVER_assume(n>=0)` in order to restrict the possible values of n to non-negative array sizes and thus prevent the underflow in the loop condition `x<n-1` of the outer loop. The procedure `swap` is assumed to be terminating.

```
void sort(int n, int a[])
{
  __CPROVER_assume(n>=0);

  for(int x=0; x<n-1; x++)
    for(int y=0; y<n-x-1; y++)
      if(a[y]>a[y+1])
        swap(a[y], a[y+1]);
}
```

We give the encoding of this procedure into our program model below. The guard $g$ expresses reachability of the head of the inner loop. The outer loop has guard $true$.

$$Init((n^{in}, a^{in}), (n^s, a^s, x^s, y^s, g^s)) \;=\; \big((n^s = n^{in}) \wedge (a^s = a^{in}) \wedge (x^s = 0) \wedge \neg g^s\big)$$

$$Assumptions((n^s, a^s, x^s, y^s, g^s), (n, a, x, y, g)) \;=\; (n^s \geq 0)$$

$$\begin{aligned}
&Trans((n,a,x,y,g),(n',a',x',y',g')) \;= \\
&\quad \big(\neg g \wedge (x < n-1) \Longrightarrow \\
&\qquad (n' = n) \wedge (a' = a) \wedge (x' = x) \wedge (y' = 0) \wedge g'\big) && (A)\\
&\quad \wedge\ \big(g \wedge \neg(y < n-x-1) \Longrightarrow \\
&\qquad (n' = n) \wedge (a' = a) \wedge (x' = x+1) \wedge (y' = y) \wedge \neg g'\big) && (B)\\
&\quad \wedge\ \big(g \wedge (y < n-x-1) \wedge (a[y] > a[y+1] \Longrightarrow \\
&\qquad (n' = n) \wedge (a' = swap(a[y], a[y+1])) \wedge (x' = x) \wedge (y' = y+1) \wedge g'\big) && (C)\\
&\quad \wedge\ \big(g \wedge (y < n-x-1) \wedge \neg(a[y] > a[y+1]) \Longrightarrow \\
&\qquad (n' = n) \wedge (a' = a) \wedge (x' = x) \wedge (y' = y+1) \wedge g'\big) && (D)\\
&\quad \wedge\ \big(\neg(x < n-1) \Longrightarrow \\
&\qquad (n' = n) \wedge (a' = a) \wedge (x' = x) \wedge (y' = y) \wedge (g' = g)\big) && (E)
\end{aligned}$$

$$Out((n^s, a^s, x^s, y^s, g^s), (n, a, x, y, g), (a^{out})) \;=\; (a^{out} = a)$$

The five cases (A)–(E) in $Trans$ describe the following path segments of the procedure: (A) from the loop head of the outer loop to the loop head of the inner loop; (B) from the loop head of the inner loop to the loop head of the outer loop, i.e., exiting the inner loop; (C) an iteration of the inner loop entering the if statement; (D) an iteration of the inner loop not entering the if statement; and (E) exiting the outer loop.

We obtain the following invariant ($true$ is the guard of the loop head of the outer loop and $g$ is the guard of the loop head of the inner loop):

$$(true \Longrightarrow 0 \leq x \leq 2^{31} - 2) \wedge (g \Longrightarrow 0 \leq y \leq 2^{31} - 2)$$

and the following termination argument:

$$\big(\neg g \wedge (x < n-1) \wedge \neg g' \Longrightarrow (-1 \cdot (x - x') + 0 \cdot (y - y')) > 0\big) \wedge \big(g \wedge g' \Longrightarrow (-1 \cdot (y - y') > 0)\big).$$

**Algorithm 3:** *compPrecondTerm*

```
Input: procedure $f$ with invariant $Inv$, additional termination conditions $termConds$
Output: precondition $Precond$
(Precond, blocked) ← (false, true);
let $\varphi = Init(\boldsymbol{x}^{in}, \boldsymbol{x}) \wedge Inv(\boldsymbol{x})$;
while $true$ do
    $\psi \leftarrow blocked \wedge \neg Precond(\boldsymbol{x}^{in}) \wedge \varphi$;
    solve $\psi$ for $\boldsymbol{x}^{in}, \boldsymbol{x}$;
    if UNSAT then return $Precond$;
    else
        let $\boldsymbol{\chi}^{in}$ be a model of $\psi$;
        let $Inv_p^o = compInv(f, \boldsymbol{x}^{in}{=}\boldsymbol{\chi}^{in})$;
        let $RR_f = compTermArg(f, Inv_p^o)$;
        if $RR_f = true$ then $blocked \leftarrow blocked \wedge (\boldsymbol{x}^{in} \neq \boldsymbol{\chi}^{in})$;
        else
            let $\theta = termConds \wedge RR_f$;
            let $Precond' = \neg compNecPrecond(f, \neg\theta)$;
            $Precond \leftarrow Precond \vee Precond'$;
```

This means that the outer loop terminates because of the increasing variable $x$ and the inner loop due to the increasing variable $y$. Both variables are bounded by the invariant. Note that, in this example, the ranking function requires only one component.

### 4.4. Preconditions for Termination

If a procedure terminates conditionally like procedure $h$ in Figure 1, then *compTermArg* will not be able to find a satisfying predicate $RR$. Algorithm 2 returns *true* ("don't know"). However, we would like to know under which preconditions, i.e., values of y in the case of $h$, the procedure terminates.

We can state this problem as defined in Definition 2.3. In Algorithm 3 we search for *Precond*, *Inv*, and $RR$ in an interleaved manner. The termination conditions *termConds* that must be satisfied in addition to the termination argument come from the propagation of preconditions for termination from the caller (through the calling context) and the callees (through summaries that are used). Note that *false* is a trivial solution for *Precond*; we thus have to aim at finding a good under-approximation of the maximal solution (weakest precondition) for *Precond*.

We bootstrap the process by assuming $Precond = false$ and search for values of $\boldsymbol{x}^{in}$ (Line 5). If such a value $\boldsymbol{\chi}^{in}$ exists, we can compute an invariant under the precondition candidate $\boldsymbol{x}^{in} = \boldsymbol{\chi}^{in}$ ($Inv_p^o$, Line 9) and search for the corresponding termination argument (Line 10).

If we fail to find a termination argument ($RR_f = true$), we block the precondition candidate (Line 11) and restart the bootstrapping process. Otherwise, the algorithm returns a termination argument $RR_f$ that is valid for the concrete value $\boldsymbol{\chi}^{in}$ of $\boldsymbol{x}^{in}$. Now we need to find a sufficiently weak *Precond* for which $RR_f$ guarantees termination. To this end, we compute an over-approximating precondition for those inputs for which we cannot guarantee termination. This is represented by $\neg\theta$ in Line 14, which includes additional termination conditions coming from the backward calling context and preconditions of procedure calls (see Section 5.2). The negation of this precondition is an under-approximation of those inputs for which $f$ terminates. Finally, we add this

| Notation | Explanation |
|---|---|
| $Init$ | initial states of the transition system |
| $Trans$ | transition relation of the transition system |
| $Out$ | output relation of the transition system |
| $Inv$ | invariant of a procedure |
| $Invs[]$ | map for storing the computed invariants (indexed by procedure) |
| $Sum$ | summary of a procedure |
| $Sums[]$ | map for storing the computed summaries (indexed by procedure) |
| $Summaries$ | summaries for all procedure calls in a procedure instantiated at the call site |
| $CallCtx$ | calling context of a procedure (the subscript indicates the call site) |
| $RR$ | termination argument |
| $\mathcal{RR}$ | template for termination argument |
| $Precond$ | precondition of a procedure |
| $Preconds[]$ | map for storing the computed preconditions (indexed by procedure) |
| $Preconditions$ | preconditions for all procedure calls in a procedure instantiated at the call site |
| $\mathcal{P}$ | template for preconditions |
| $Assumptions$ | correspond to assume statements in the code |
| Superscript $^o$ | indicates over-approximations |
| Superscript $^u$ | indicates under-approximations |
| Superscript $^{\widetilde{u}}$ | indicates complement of under-approximations |
| Subscript $_f$ | indicates the associated procedure |

Table I: Notation

negated precondition to our $Precond$ (Line 15) before we start over the bootstrapping process. Our aim is to find precondition candidates outside the current precondition ($\neg Precond$) for which we might be able to guarantee termination.

*Example* 4.2. Let us consider again function `h` in Figure 1. This time, we will assume we have the invariant $0 \leq x \leq M$ (with $M := 2^{32} - 1$). We bootstrap by assuming $Precond = false$ and searching for values of $y$ satisfying $true \wedge \neg false \wedge x{=}0 \wedge 0 \leq x \leq M$. The variable $y$ is unconstrained in this formula, hence it can take any value; one possibility is $y = 0$. We then compute the invariant under the precondition $y = 0$ and get $x = 0$. Obviously, we cannot find a termination argument in this case. Hence, we start over and search for values of $y$ satisfying $y \neq 0 \wedge \neg false \wedge x{=}0 \wedge 0{\leq}x{\leq}M$. This formula is for instance satisfied by $y = 1$. This time we get the invariant $0{\leq}x{\leq}10$ and the ranking function $-x$. Thus, we have to solve

$$\exists \boldsymbol{e} : \mathcal{P}(y, \boldsymbol{e}) \wedge 0{\leq}x{\leq}M \wedge x'{=}x{+}y \wedge x{<}10 \Rightarrow \neg(-x{>} - x')$$

to compute an over-approximating precondition over the template $\mathcal{P}$ (for details on templates see Section 2.3). The aboe formula means that we are looking for parameter values $\boldsymbol{e}$ such that $\mathcal{P}$ instantiated with these values entails that the termination argument candidate $-x > -x'$ does not hold. In this case, $\mathcal{P}(y, e)$ turns out to be $y = 0$, therefore its negation $y \neq 0$ is the $Precond$ that we get. Finally, we have to check for further precondition candidates, but $y \neq 0 \wedge \neg(y \neq 0) \wedge x{=}0 \wedge 0{\leq}x{\leq}M$ is obviously UNSAT. Hence, we return the sufficient precondition for termination $y \neq 0$.

## 5. INTERPROCEDURAL TERMINATION ANALYSIS

Now that we described how the intraprocedural termination analysis works, we can get back to the interprocedural analysis described in Algorithm 1. Essentially, Algorithm 1 solves a series of formulae in second-order predicate logic with existentially quantified

**Algorithm 4:** *analyze* for universal termination

```
global $Sums^o, Invs^o, termStatus$;
function $analyzeForward(f, CallCtx^o_f)$
    foreach procedure call $h$ in $f$ do
        $CallCtx^o_h = \underline{compCallCtx^o}(f, CallCtx^o_f, h)$;
        if $needToReAnalyze^o(h, CallCtx^o_h)$ then
            $analyzeForward(h, CallCtx^o_h)$;
    $join^o((Sums^o[f], Invs^o[f]), \underline{compInvSum^o}(f, CallCtx^o_f))$
function $analyzeBackward'(f)$
    $join(termStatus[f], \underline{compTermArg}(f))$;
    foreach procedure call $h$ in $f$ do
        if $needToReAnalyze^u(h, CallCtx^o_h)$ then
            $analyzeBackward'(h)$;
            $join(termStatus[f], termStatus[h])$;
function $analyze(f_{entry})$
    $analyzeForward(f_{entry}, true)$;
    $analyzeBackward'(f_{entry})$;
    return $termStatus[f_{entry}]$;
```

predicates, for which we are seeking a satisfying model.[4] In this section, we state the constraints we solve, including all the side constraints arising from the interprocedural analysis. Note that this is not a formalisation exercise, but these are precisely the formulae solved by our synthesis back-end, which is described in Section 2.3. To facilitate readability, we summarise the notations in Table I. In particular, mind the difference between *Sum*, *Sums*, and *Summaries*.

### 5.1. Universal Termination

We first explain a simpler variant of Algorithm 1. The variant is able to show universal termination (see Algorithm 4). This variant reduces the backward analysis to a call to *compTermArg* and propagating back the qualitative result (*termStatus*) obtained: *terminating*, *potentially non-terminating* or *non-terminating*.

This section states the constraints that are solved to compute the outcome of the functions underlined in Algorithm 4 and establishes its soundness:

— $compCallCtx^o$ (Definition 5.1)
— $compInvSum^o$ (Definition 5.4)
— *compTermArg* (Lemma 5.8)

To provide further intuition behind the algorithm, Figure 5 illustrates the dependencies between the predicates (*Inv*, *CallCtx*, *RR*, *Sum*) and *termStatus*, where $h$ is a procedure called in $f$. The element at the arrow tail is required to compute the element at the arrow head. For instance, in order to compute $CallCtx_h$ for a procedure call $h$ in $f$ we have to compute $Inv_f$, which in turn has dependencies on the calling context $CallCtx_f$ of $f$ and the summaries $Sum_h$ for each procedure $h$ called in $f$. The dashed arrows indicate indirect dependencies. For example, we need the $CallCtx_h$ of $h$ to com-

[4] To be precise, we are not only looking for model predicates but (good approximations of) weakest or strongest predicates. Finding such biased models is a feature of our synthesis algorithms.

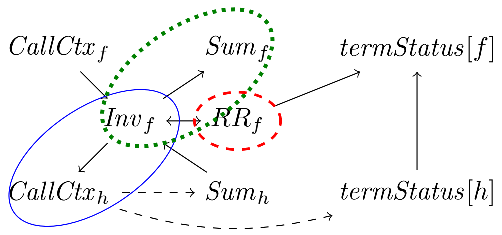

Fig. 5: Dependencies in universal termination and decomposition

pute its summary $Sum_h$. These dependencies become apparent when we consider the dependencies of each procedure recursively over the call graph, building the predicate dependency graph for the entire universal termination problem of a procedural program. This would allow us to construct a single second-order formulae characterising that problem [Schrammel 2016]. However, this formula might be large and hard to solve. Algorithm 4 breaks down this formula into smaller parts that can be easily solved. These parts are visualised as ellipses in Figure 5: $compCallCtx^o$ in solid blue, $compInvSum^o$ in dotted green, and $compTermArg$ in dashed red.

### *5.1.1. Over-approximating Calling Contexts*

*Definition* 5.1 ($compCallCtx^o$). A forward calling context $CallCtx^o_{h_i}$ for $h_i$ in procedure $f$ in calling context $CallCtx^o_f$ is a satisfying model of the following formula:

$$\begin{array}{l}\exists CallCtx^o_{h_i}, Inv^o_f : \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}^{out}, \boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}} :\\ \quad CallCtx^o_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Summaries^o_f \wedge Assumptions_f(\boldsymbol{x}^s, \boldsymbol{x})\\ \quad\Longrightarrow\big(Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x}^s)\big) \;\wedge\\ \qquad\big(Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \wedge Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Trans_f(\boldsymbol{x}, \boldsymbol{x}') \wedge Out_f(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{x}^{out})\\ \qquad\;\Longrightarrow Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x}') \wedge (g_{h_i} \Rightarrow CallCtx^o_{h_i}(\boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}})\big)\end{array}$$

with

$$Summaries^o_f = \bigwedge_{\text{calls } h_j \text{ in } f} g_{h_j} \Longrightarrow Sums^o[h](\boldsymbol{x}_j^{p_{in}}, \boldsymbol{x}_j^{p_{out}})$$

where $g_{h_j}$ is the guard condition of procedure call $h_j$ in $f$ capturing the branch conditions from conditionals.

For example, $g_{h_0}$ of the procedure call to `h` in `f` in Figure 1 is $z > 0$. $Sums^o[h]$ is the currently available summary for `h` (cf. global variables in Algorithm 1).

Assumptions correspond to assume statements in the code that most program analyzers define in order to restrict non-determinism. In our tool this is done using `__CPROVER_assume(condition)`.

Similar to Theorem 2.4, we say that the calling context $CallCtx^o_{h_i}$ (together with $Inv^o_f$) is a satisfying model to mean that:

$$\begin{array}{l}(CallCtx^o_{h_i}, Inv^o_f) \models \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}^{out}, \boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}} :\\ \quad CallCtx^o_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Summaries^o_f \wedge Assumptions_f(\boldsymbol{x}^s, \boldsymbol{x})\\ \quad\Longrightarrow\big(Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \Longrightarrow Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x})\big) \;\wedge\\ \qquad\big(Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \wedge Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Trans_f(\boldsymbol{x}, \boldsymbol{x}') \wedge Out_f(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{x}^{out})\\ \qquad\;\Longrightarrow Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x}') \wedge (g_{h_i} \Rightarrow CallCtx^o_{h_i}(\boldsymbol{x}_i^{p_{in}}, \boldsymbol{x}_i^{p_{out}})\big)\end{array}$$

Note that Definition 5.1 states the inductiveness and the initialization conditions for $Inv^o_f$ and establishes the calling context $CallCtx^o_{h_i}$ for $h_i$ according to Definition 2.1, under the calling context $CallCtx^o_f$, assumptions $Assumptions_f$ and the currently available

| Definition | Example |
|---|---|
| $\boldsymbol{x}^{in}$ | $(z^{in})$ |
| $\boldsymbol{x}^{s}$ | $(z^{s})$ |
| $\boldsymbol{x}$ | $(z)$ |
| $\boldsymbol{x}'$ | $(z')$ |
| $\boldsymbol{x}^{p_{in}}$ | $(p_{in})$ |
| $\boldsymbol{x}^{p_{out}}$ | $(p_{out})$ |
| $\boldsymbol{x}^{out}$ | $(w^{out})$ |

Table II: Variable mapping for procedure `f`

summary for `h`, $\mathit{Sums}^o[h]$. Using Definition 5.1, we can compute $\mathit{Inv}_f^o$ and $\mathit{CallCtx}_{h_i}^o$ at the same time.

LEMMA 5.2. *$\mathit{CallCtx}_{h_i}^o$ is over-approximating.*

PROOF SKETCH. By induction over the acyclic call graph:
— Base case: $\mathit{CallCtx}_f^o$ is $\mathit{true}$ when $f$ is the entry-point procedure; also, the summary $\mathit{Sums}^o[h]$ is initially assumed to be $\mathit{true}$, i.e., over-approximating.
— Step case: Given that $\mathit{CallCtx}_f^o$ and $\mathit{Summaries}_f^o$ (which is built from $\mathit{Sums}^o[h]$) are over-approximating, $\mathit{CallCtx}_{h_i}^o$ is over-approximating by the soundness of the synthesis (see Theorem 2.4 in Section 2.3). □

*Example* 5.3. Let us consider procedure `f` in Figure 1. For ease of understanding, we provide in Table II the mapping between the variables in Definition 5.1 and those in the encoding of `f`.

Procedure `f` is the entry procedure, hence we have $\mathit{CallCtx}_f^o((z^{in}),(w^{out})) = \mathit{true}$, which is $0 \le z^{in} \le M \wedge 0 \le w^{out} \le M$ where $M := 2^{32}-1$ when using the interval abstract domain for 32 bit integers. Then, we instantiate Definition 5.1 (for procedure `f`) to compute $\mathit{CallCtx}_{h_0}^o$. We assume that we have not yet computed a summary for `h`, thus, $\mathit{Sums}^o[h]$ is $\mathit{true}$. Remember that the placeholder $h_0((p_{in}),(p_{out}))$ evaluates to $\mathit{true}$. Notably, there are no assumptions in the code, meaning that $\mathit{Assumptions}_f((z^s),(z)) = \mathit{true}$.

$$\begin{array}{l}
\exists \mathit{CallCtx}_{h_0}^o, \mathit{Inv}_f^o : \forall z^{in}, z^s, z, z', w^{out} : \\
\quad 0 \le z^{in} \le M \wedge 0 \le w^{out} \le M \wedge (\mathit{true} \Longrightarrow \mathit{true}) \wedge \mathit{true} \\
\quad \Longrightarrow \big(z^s{=}z^{in} \Longrightarrow \mathit{Inv}_f^o((z^s),(z^s))\big) \wedge \\
\qquad \big(z^s{=}z^{in} \wedge \mathit{Inv}_f^o((z^s),(z)) \wedge \mathit{true} \wedge p_{in}{=}z^s/2{+}1 \wedge h_0((p_{in}),(p_{out})) \wedge w^{out}{=}p_{out} \\
\qquad\quad \Longrightarrow \mathit{Inv}_f^o((z^s),(z')) \wedge (\mathit{true} \Rightarrow \mathit{CallCtx}_{h_0}^o((p_{in}),(p_{out})))\big)
\end{array}$$

A solution is $\mathit{Inv}_f^o = \mathit{true}$, and $\mathit{CallCtx}_{h_0}^o((p_{in}),(p_{out})) = (1 \le p_{in} \le 2^{31} \wedge 0 \le p_{out} \le M)$. Note that $\lfloor \frac{M}{2} \rfloor + 1 = 2^{31}$.

### 5.1.2. Over-approximating Invariants and Summaries

*Definition* 5.4 ($\mathit{compInvSum}^o$). A forward summary $\mathit{Sum}_f^o$ and invariants $\mathit{Inv}_f^o$ for procedure $f$ in calling context $\mathit{CallCtx}_f^o$ are a satisfying model of the following formula:

$$\begin{array}{l}
\exists \mathit{Sum}_f^o, \mathit{Inv}_f^o : \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}^{out} : \\
\quad \mathit{CallCtx}_f^o(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge \mathit{Summaries}_f^o \wedge \mathit{Assumptions}_f(\boldsymbol{x}^s, \boldsymbol{x}) \\
\quad \Longrightarrow \big(\mathit{Init}_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \wedge \mathit{Inv}_f^o(\boldsymbol{x}^s, \boldsymbol{x}') \wedge \mathit{Out}_f(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{x}^{out}) \\
\qquad\quad \Longrightarrow \mathit{Inv}_f^o(\boldsymbol{x}^s, \boldsymbol{x}^s) \wedge \mathit{Sum}_f^o(\boldsymbol{x}^{in}, \boldsymbol{x}^{out})\big) \wedge \\
\qquad \big(\mathit{Inv}_f^o(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \mathit{Trans}_f(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow \mathit{Inv}_f^o(\boldsymbol{x}^s, \boldsymbol{x}')\big)
\end{array}$$

| Definition | Example |
|---|---|
| $\boldsymbol{x}^{in}$ | $(y^{in})$ |
| $\boldsymbol{x}^{s}$ | $(x^s, y^s)$ |
| $\boldsymbol{x}$ | $(x, y)$ |
| $\boldsymbol{x}'$ | $(x', y')$ |
| $\boldsymbol{x}^{out}$ | $(x^{out})$ |

Table III: Variable mapping for procedure `h`

Using Definition 5.4, we can compute $Inv_f^o$ and $Sum_f^o$ at the same time. This formula is the result of merging the definitions for invariant and summary from Definition 2.1 (and adding assumptions, callee summaries, and calling context information). The first implication of the invariant in Definition 2.1 is merged with the implication of the summary definition. $Inv_f^o(\boldsymbol{x}^s, \boldsymbol{x})$ appears on the right-hand side now because it is inferred, whereas it appears on the left-hand side in Definition 2.1 because it is assumed given.

Since at this point all callee summaries $Summaries_f^o$ are available, the invariant $Inv_f^o$ will be more precise than the one obtained by solving the formula in Definition 5.1. Similarly to Definition 5.1, $Assumptions$ in Definition 5.4 corresponds to assume statements in the code that most program analyzers define in order to restrict non-determinism.

LEMMA 5.5. *$Sum_f^o$ and $Inv_f^o$ are over-approximating.*

PROOF SKETCH. By induction over the acyclic call graph:
— Base case: By Lemma 5.2, $CallCtx_f^o$ is over-approximating. Also, the summaries in $Summaries_f^o$ are initially assumed to be $true$, i.e., over-approximating.
— Step case: Given that $CallCtx_f^o$ and $Summaries_f^o$ are over-approximating, $Sum_f^o$ and $Inv_f^o$ are over-approximating by the soundness of the synthesis (Theorem 2.4). □

*Example* 5.6. Let us consider procedure `h` in Figure 1. The mapping between the variables in Definition 5.4 and those in the encoding of `h` is given in Table III.

We have computed $CallCtx_{h_0}^o((y^{in}), (x^{out})) = (1{\le}y^{in}{\le}2^{31} \wedge 0{\le}x^{out}{\le}M)$ (with actual parameters renamed to formal ones) in Example 5.3. Then, we need to obtain models $Inv_{h_0}^o$ and $Sum_{h_0}^o$ to satisfy of the instantiation of Definition 5.4 (for procedure `h`) as given below.

$$\begin{array}{l}
\exists Inv_{h_0}^o, Sum_{h_0}^o : \forall y^{in}, x^s, y^s, x, y, x', y', x^{out} : \\
\quad 1{\le}y^{in}{\le}2^{31} \wedge 0{\le}x^{out}{\le}M \wedge true \wedge true \\
\quad\Longrightarrow \big((x^s{=}0 \wedge y^s{=}y^{in}) \wedge Inv_{h_0}^o((x^s, y^s), (x', y')) \wedge (x^{out}{=}x' \wedge \neg(x'{<}10)) \\
\qquad\qquad \Longrightarrow Inv_{h_0}^o((x^s, y^s), (x^s, y^s)) \wedge Sum_{h_0}^o((y^{in}), (x^{out}))\big) \wedge \\
\qquad \big(Inv_{h_0}^o((x^s, y^s), (x, y)) \wedge x'{=}x{+}y \wedge x{<}10 \wedge y{=}y') \Longrightarrow Inv_{h_0}^o((x^s, y^s), (x', y'))\big)
\end{array}$$

A solution is $Inv_{h_0}^o = (0{\le}x{\le}2^{31}{+}9 \wedge 1{\le}y{\le}2^{31})$ and $Sum_{h_0}^o = (1{\le}y^{in}{\le}2^{31} \wedge 10{\le}x^{out}{\le}2^{31}{+}9)$, for instance.

*Remark* 5.7. Since Definition 5.1 and Definition 5.4 are interdependent, we would have to compute them iteratively until a fixed point is reached in order to improve the precision of calling contexts, invariants and summaries. However, for efficiency reasons, we perform only the first iteration of this (greatest) fixed point computation. This is a design choice made in order to break the cycles in Figure 5.

### 5.1.3. Computing Termination Arguments

LEMMA 5.8 ($compTermArg$). *A procedure $f$ with forward invariants $Inv_f^o$ terminates if there is a termination argument $RR_f$:*

$$\begin{array}{l}\exists RR_f : \forall \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}' : \\ \quad Summaries_f^o \wedge Assumptions_f(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \\ \quad Inv_f^o(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Trans_f(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow RR_f(\boldsymbol{x}, \boldsymbol{x}')\end{array}$$

Over-approximating forward information may lead to inclusion of spurious non-terminating traces. For that reason, we might not find a termination argument although the procedure is terminating. As we essentially under-approximate the set of terminating procedures, we will not give false positives. Regarding the solving algorithm for this formula, we present it in the next section.

*Example* 5.9. Let us consider function `h` in Figure 1. The mapping between the variables in Lemma 5.8 and those in the encoding of `h` is given in Table III. Assume we have the invariant $(0{\le}x{\le}2^{31}{+}9) \wedge (1{\le}y{\le}2^{31})$, as computed in Example 5.6. Thus, we have to solve

$$\begin{array}{l}\exists RR_h : \forall x^s, y^s, x, y, x', y' : \\ \quad true \wedge true \wedge \\ \quad (0{\le}x{\le}2^{31}{+}9) \wedge (1{\le}y{\le}2^{31}) \wedge x'{=}x{+}y \wedge x{<}10 \wedge y'{=}y \Longrightarrow RR_h((x,y),(x',y'))\end{array}$$

When using a linear ranking function template $\ell_x \cdot x + \ell_y \cdot y$, we obtain as solution, for example, $RR_h = (-x > -x')$, i.e., $\ell_x{=}{-1}$ and $\ell_y{=}0$.

### 5.1.4. Proving Never-Termination.

If there is no trace from procedure entry to exit, then we can prove non-termination, even when using over-approximations:

LEMMA 5.10. *A procedure $f$ in forward calling context $CallCtx_f^o \neq false$ and forward invariants $Inv_f^o$ never terminates if its summary $Sum_f^o$ is $false$.*

PROOF SKETCH. If the over-approximating summary of $f$ is $false$ then there is no execution path from the entry to the exit. This entails that $f$ does not terminate. □

### 5.1.5. Propagating the Termination Information.

The summary $Sum_f^o$ is computed in Line 7 of Algorithm 4.

Termination information is then propagated in the (acyclic) call graph ($join$ in Line 13 in Algorithm 4):

PROPOSITION 5.11. *A procedure $f$ is declared*

*(1)* non-terminating *if it is non-terminating by Lemma 5.10.*
*(2)* terminating *if*
  *(a) all its procedure calls $h_i$ that are potentially reachable (i.e., with $CallCtx_{h_i}^o \neq false$) are declared terminating, and*
  *(b) $f$ itself is terminating according to Lemma 5.8;*
*(3)* potentially non-terminating *otherwise.*

Our implementation is more efficient than Algorithm 4 because it avoids computing a termination argument for $f$ if one of its callees is potentially non-terminating.

THEOREM 5.12. *(a) If the entry procedure of a program is declared terminating, then the program terminates universally.*

*(b) If the entry procedure of a program is declared non-terminating, then the program never terminates.*

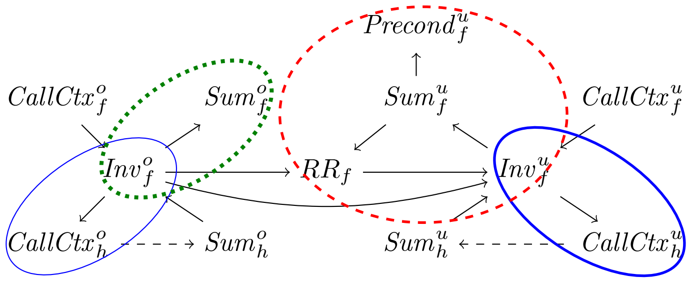

Fig. 6: Dependent predicates in conditional termination and decomposition

PROOF SKETCH. (a) Assume that a program does not terminate, but its entry procedure is declared *terminating*. Then there must be a reachable procedure $h$ in the call graph that does not terminate. Now, we proceed by induction following the call graph from $h$ to the entry procedure.

— Base case: If $h$ does not terminate then it is either declared *non-terminating* by Lemma 5.10 or *potentially non-terminating* by a failure to prove its termination by Lemma 5.8.

— Step case: By Proposition 5.11 procedure $f$ is only declared *terminating* if all of its reachable callees have been declared terminating. Hence, if $f$ has a reachable callee $h$, $f$ is either declared *non-terminating* or *potentially non-terminating*.

It follows that the entry procedure cannot be declared *terminating*, which contradicts the assumption.

(b) Assume that a program terminates, but its entry procedure $f$ is declared *non-terminating*. Then, by Lemma 5.10, the over-approximating summary of $f$ must be $false$, which contradicts the assumption that the program terminates. □

## 5.2. Conditional Termination

We now extend the formalisation to Algorithm 1, which additionally requires the computation of under-approximating calling contexts and sufficient preconditions for termination (procedure $compPrecondTerm$, see Algorithm 3).

We will start by reminding the reader the details of procedure $compPrecondTerm$. First, it computes in Line 9 an over-approximating invariant $Inv_p^o$ entailed by the candidate precondition. $Inv_p^o$ is computed through Definition 5.4 by conjoining the candidate precondition to the antecedent. Then, Line 10 computes the corresponding termination argument $RR_f$ by applying Lemma 5.8 using $Inv_p^o$ instead of $Inv_f^o$. A termination argument $RR_f \neq true$ proves that $f$ terminates for this candidate precondition.

Then, in Line 14 of $compPrecondTerm$, we compute under-approximating (sufficient) preconditions for traces satisfying the termination argument $RR_f$ via over-approximating the traces violating $RR_f$.

Now, we are left to specify the formulae corresponding to the following functions:

— $compCallCtx^u$ (Definition 5.13)

— $compNecPrecond$ (Definition 5.16)

Figure 6 illustrates the dependencies between the predicates and the decomposition [Schrammel 2016] that we have chosen. $compCallCtx^u$ is the thick blue ellipse and $compNecPrecond$ the dashed red ellipse.

In the sequel, we use the superscript $\widetilde{u}$ to indicate negations of under-approximating information.

### 5.2.1. Under-approximating Calling Contexts.

We compute under-approximating calling contexts as follows:

*Definition* 5.13 ($compCallCtx^u$). The backward calling context $CallCtx^u_{h_i}$ for procedure call $h_i$ in procedure $f$ in backward calling context $CallCtx^u_f$ and forward invariants $Inv^o_f$ is $CallCtx^u_{h_i} \equiv \neg\widetilde{CallCtx}^{u}_{h_i}$, the negation of a satisfying model for:

$$\begin{array}{l}
\exists \widetilde{CallCtx}^{u}_{h_i}, \widetilde{Inv}^{u}_{f} : \forall \boldsymbol{x}^{in}, \boldsymbol{x}^s, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}'', \boldsymbol{x}^{p_{in}}_i, \boldsymbol{x}^{p_{out}}_i, \boldsymbol{x}^{out} : \\
\quad \neg CallCtx^u_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge \widetilde{Summaries}^{u}_{f} \wedge Assumptions_f(\boldsymbol{x}^s, \boldsymbol{x}) \wedge \\
\quad CallCtx^o_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Inv^o_f(\boldsymbol{x}^s, \boldsymbol{x}) \wedge Summaries^o_f \\
\quad\quad \Longrightarrow \big( Out_f(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{x}^{out}) \Longrightarrow \widetilde{Inv}^{u}_{f}(\boldsymbol{x}^s, \boldsymbol{x}')\big) \;\; \wedge \\
\quad\quad\quad \big( Out_f(\boldsymbol{x}^s, \boldsymbol{x}'', \boldsymbol{x}^{out}) \wedge \widetilde{Inv}^{u}_{f}(\boldsymbol{x}^s, \boldsymbol{x}') \wedge Trans_f(\boldsymbol{x}, \boldsymbol{x}') \wedge Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \\
\quad\quad\quad\quad \Longrightarrow \widetilde{Inv}^{u}_{f}(\boldsymbol{x}^s, \boldsymbol{x}) \wedge (g_{h_i} \Longrightarrow \widetilde{CallCtx}^{u}_{h_i}(\boldsymbol{x}^{p_{in}}, \boldsymbol{x}^{p_{out}}))\big)
\end{array}$$

with $\widetilde{Summaries}^{u}_{f} = \bigwedge_{\text{calls } h_j \text{ in } f} g_{h_j} \Longrightarrow \neg Sums^u[h](\boldsymbol{x}^{p_{in}}_j, \boldsymbol{x}^{p_{out}}_j)$.

This definition is structurally similar to the definition of $CallCtx^o$ (Definition 5.1). However, here we perform a backwards computation: note that the base case (Line 4) of the inductive invariant definition starts from $Out$ instead from $Init$, and that we reverse $Trans$ in the step case (Line 5). Since $CallCtx^u$ must be contained in $CallCtx^o$ (and similarly invariants and callee summaries), we can constrain the formula using the information obtained from the forward analysis in Line 3. These are not required for soundness, but they might help solving the formula by restricting the search space.

LEMMA 5.14. *$CallCtx^u_{h_i}$ is under-approximating.*

PROOF SKETCH. The computation is based on the negation of the under-approximating calling context of $f$ and the negated under-approximating summaries for the function calls in $f$. By Theorem 2.4, this leads to an over-approximation of the negation of the calling context for $h_i$. □

*Example* 5.15. Let us consider procedure `f` in Figure 1. For the mapping between the variables in Definition 5.13 and those in the encoding of `f` we refer to Table II. Let us assume that in `f`, we have $CallCtx^u_f((z^{in}), (w^{out})) = (11 \leq w^{out} \leq M)$, i.e., `f` is called in a context where a return value of less than 11 would cause non-termination of the caller of `f`. Then, we instantiate Definition 5.13 (for procedure `f`) to compute $CallCtx^u_{h_0}$. We assume that we have already computed an over-approximating summary $Sum^o_{h_0} = (1 \leq p_{in} \leq M \wedge 0 \leq p_{out} \leq M)$, but not yet computed an under-approximating summary for `h`, thus, $Sum^u_h$ is $false$. Notably, there are no assumptions in the code, meaning that $Assumptions_f((z^s), (z)) = true$.

$$\begin{array}{l}
\exists \widetilde{CallCtx}^{u}_{h_0}, \widetilde{Inv}^{u}_{f} : \forall z^{in}, z^s, z, z', w^{out} : \\
\quad 0 \leq w^{out} \leq 10 \wedge (true \Rightarrow \neg false) \wedge true \wedge \\
\quad true \wedge true \wedge (true \Rightarrow 1 \leq p_{in} \leq M \wedge 0 \leq p_{out} \leq M) \\
\quad\quad \Longrightarrow \big(p_{in} = z^s/2{+}1 \wedge h_0((p_{in}), (p_{out})) \wedge w^{out} = p_{out} \Longrightarrow \widetilde{Inv}^{u}_{f}((z^s), (z'))\big) \wedge \\
\quad\quad\quad \big(p_{in} = z^s/2{+}1 \wedge h_0((p_{in}), (p_{out})) \wedge w^{out} = p_{out} \wedge \widetilde{Inv}^{u}_{f}((z^s), (z')) \wedge true \wedge z^s = z^{in} \\
\quad\quad\quad\quad \Longrightarrow \widetilde{Inv}^{u}_{f}((z^s), (z)) \wedge (true \Rightarrow \widetilde{CallCtx}^{u}_{h_0}((p_{in}), (p_{out})))\big)
\end{array}$$

A solution is $Inv^{\widetilde{u}}_f = true$, and $CallCtx^{\widetilde{u}}_{h_0} = (1{\le}p_{in}{\le}M \wedge 0{\le}p_{out}{\le}10)$, i.e., $CallCtx^{u}_{h_0} = (p_{in}{=}0 \vee 11{\le}p_{out}{\le}M)$.

### 5.2.2. Under-approximating Preconditions for Termination

*Definition* 5.16 (*Line 14 of* $compPrecondTerm$). A precondition for termination $Precond^u_f$ in backward calling context $CallCtx^u_f$ and with forward invariants $Inv^o_f$ is $Precond^u_f \equiv \neg Precond^{\widetilde{u}}_f$, i.e., the negation of a satisfying model $Precond^{\widetilde{u}}_f$ for:

$$\begin{array}{l}
\exists Precond^{\widetilde{u}}_f, Inv^{\widetilde{u}}_f, Sum^{\widetilde{u}}_f : \forall \boldsymbol{x}^{in}, \boldsymbol{x}, \boldsymbol{x}', \boldsymbol{x}^{out} : \\
\quad \neg CallCtx^u_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Summaries^{\widetilde{u}}_f \wedge Assumptions_f(\boldsymbol{x}', \boldsymbol{x}) \wedge \\
\quad CallCtx^o_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Inv^o_f(\boldsymbol{x}', \boldsymbol{x}) \wedge Summaries^o_f \\
\qquad \Longrightarrow \big(Out_f(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{x}^{out}) \wedge Inv^{\widetilde{u}}_f(\boldsymbol{x}^s, \boldsymbol{x}') \wedge Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \\
\qquad\qquad \Longrightarrow Inv^{\widetilde{u}}_f(\boldsymbol{x}^s, \boldsymbol{x}') \wedge Sum^{\widetilde{u}}_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Precond^{\widetilde{u}}_f(\boldsymbol{x}^{in})\big) \ \wedge \\
\qquad \Big(\big(\neg RR_f(\boldsymbol{x}, \boldsymbol{x}') \vee Preconditions^{\widetilde{u}}_f \vee \\
\qquad\quad \neg CallCtx^u_f(\boldsymbol{x}^{in}, \boldsymbol{x}^{out}) \wedge Init_f(\boldsymbol{x}^{in}, \boldsymbol{x}^s) \wedge Out_f(\boldsymbol{x}^s, \boldsymbol{x}', \boldsymbol{x}^{out})\big) \ \wedge \\
\qquad Inv^{\widetilde{u}}_f(\boldsymbol{x}^s, \boldsymbol{x}') \wedge Trans_f(\boldsymbol{x}, \boldsymbol{x}') \Longrightarrow Inv^{\widetilde{u}}_f(\boldsymbol{x}^s, \boldsymbol{x})\Big)
\end{array}$$

with $Preconditions^{\widetilde{u}}_f = \bigvee_{\text{calls } h_j \text{ in } f} g_{h_j} \Longrightarrow \neg Preconds^u[h](\boldsymbol{x}^{p_{in}}_j, \boldsymbol{x}^{p_{out}}_j)$.

$Precond^{\widetilde{u}}_f$ describes the set of inputs for which $f$ might not terminate. Structurally, the formula is similar to Definition 5.4 regarding the simultaneous computation of invariants and summary of $f$. However, similar to Definition 5.13 it proceeds backwards ($\boldsymbol{x}^{out}$ is directly tied to $Sum^{\widetilde{u}}_f$ through $Out$, whereas $\boldsymbol{x}^{in}$ is a consequence of $Inv^{\widetilde{u}}_f$).

Essentially, this definition is saying that a value satisfying $Precond^{\widetilde{u}}_f$ could cause non-termination in several ways (disjuncts in Line 5):
— either in $f$ itself, by not satisfying the termination argument $RR_f$,
— in a callee $h$, by not satisfying its termination precondition ($Preconditions^{\widetilde{u}}_f$),
— or through the outputs of $f$ in the caller of $f$ ($\neg CallCtx^u_f$).
The subformula in Line 5 is the argument $\neg\theta$ in Line 14 of Algorithm 3. We denote the negation of the models found for the summary and the invariant by $Sum^u_f \equiv \neg Sum^{\widetilde{u}}_f$ and $Inv^u_f \equiv \neg Inv^{\widetilde{u}}_f$, respectively.

LEMMA 5.17. *$Precond^u_f$, $Sum^u_f$ and $Inv^u_f$ are under-approximating.*

PROOF SKETCH. We compute an over-approximation of the negation of the precondition w.r.t. the negation of the under-approximating termination argument and the negation of further under-approximating information (backward calling context, preconditions of procedure calls) — by the soundness of the synthesis (see Theorem 2.4 in Section 2.3), this over-approximates the non-terminating traces, and hence under-approximates the terminating ones. Hence, the precondition is a sufficient precondition for termination. □

*Example* 5.18. This example is a continuation of Example 5.15. We will instantiate Definition 5.16 for `h` in Figure 1. Regarding the mapping between the variables in Definition 5.16 and those in the encoding of `h` we refer to Table III. We assume that we have $CallCtx^u_h((y^{in}), (x^{out})) = (y^{in}{=}0 \vee 11{\le}x^{out}{\le}M)$, as computed in Example 5.15.

We assume $\mathit{CallCtx}_h^o = \mathit{true}$ and $\mathit{Inv}_h^o = \mathit{true}$. h does not have procedure calls, thus $\mathit{Summaries}_h^o = \mathit{true}$, $\mathit{Summaries}_h^{\tilde{u}} = \mathit{true}$ and $\mathit{Preconditions}_h^{\tilde{u}} = \mathit{false}$. Assume we have the termination argument candidate $RR_h = (-x > -x')$.

$$\begin{array}{l}
\exists \mathit{Precond}_h^{\tilde{u}}, \mathit{Inv}_h^{\tilde{u}}, \mathit{Sum}_h^{\tilde{u}} : \forall y^{in}, x^s, y^s, x, y, x', y', x^{out} : \\
\quad \neg(y^{in}{=}0 \vee 11{\le}x^{out}{\le}M) \wedge \mathit{true} \wedge \mathit{true} \wedge \\
\quad \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \\
\qquad \Longrightarrow \big((x^{out}{=}x' \wedge \neg(x'{<}10)) \wedge \mathit{Inv}_h^{\tilde{u}}((x^s, y^s), (x', y')) \wedge (x^s{=}0 \wedge y^s{=}y^{in}) \\
\qquad\qquad \Longrightarrow \mathit{Inv}_h^{\tilde{u}}((x^s, y^s), (x', y')) \wedge \mathit{Sum}_h^{\tilde{u}}(y^{in}, x^{out}) \wedge \mathit{Precond}_h^{\tilde{u}}(y^{in})\big) \;\; \wedge \\
\qquad\quad \Big(\big(\neg(-x > -x') \vee \mathit{false} \vee \\
\qquad\qquad \neg(y^{in}{=}0 \vee 11{\le}x^{out}{\le}M) \wedge (x^s{=}0 \wedge y^s{=}y^{in}) \wedge (x^{out}{=}x' \wedge \neg(x'{<}10))\big) \wedge \\
\qquad\quad \mathit{Inv}_h^{\tilde{u}}((x^s, y^s), (x', y')) \wedge (x'{=}x{+}y \wedge x{<}10 \wedge y{=}y') \Longrightarrow \mathit{Inv}_h^{\tilde{u}}((x^s, y^s), (x, y))\Big)
\end{array}$$

The second conjunct in the consequent of the top-level implication is satisfied either for $y^{in}{=}0$ making $\neg RR_h$ true or $x^{out}{\le}10$ violating the calling context (third disjunct).

Hence, a solution is

$$\begin{aligned}
\mathit{Precond}_h^{\tilde{u}} &= (0{\le}y^{in}{\le}1) \\
\mathit{Sum}_h^{\tilde{u}} &= (0{\le}y^{in}{\le}1 \wedge x^{out}{=}10) \\
\mathit{Inv}_h^{\tilde{u}} &= (x^s{=}0 \wedge 0{\le}y^s{\le}1 \wedge 0{\le}x{\le}10 \wedge 0{\le}y{\le}1)
\end{aligned}$$

Thus, a sufficient precondition for termination is $\mathit{Precond}_h^u = (2{\le}y^{in}{\le}M)$. Note that without the backwards propagation of the calling context from the caller of `f` through `f` into `h`, we would be unable to derive that we require $y^{in} \ge 2$ in order to prevent $x^{out} \le 10$ and consequently $w^{out} \le 10$, which would lead to non-termination in the caller of `f`.

We can therefore conclude:

THEOREM 5.19. *A program with entry procedure $f$ terminates for all values of $x^{in}$ satisfying $\mathit{Precond}_f^u$.*

PROOF SKETCH. Assume that there as a value $\chi^{in}$ that satisfies the precondition, but for which the program does not terminate. Following Definition 5.16, a value $\chi^{in}$ satisfying $\mathit{Precond}_f^u$ of a procedure $f$ must cause non-termination either in $f$ (by not satisfying the termination argument $RR_f$), in a callee $h$ in $f$, or through the outputs of $f$ in the caller of $f$. We proceed by induction over the call graph from the leaves up to the entry procedure:

— Base case: Assume that a procedure $f$ has no callees, and $\chi^{in}$ satisfies $RR_f$ (otherwise it would not have been included in $\mathit{Precond}_f^u$, Lemma 5.17). Hence, $f$ itself terminates, but $\chi^{in}$ could cause non-termination of the caller of $f$.

— Step case: Assume that the callees in a procedure $f$ terminate. By Lemma 5.17, $\chi^{in}$ satisfies $RR_f$; hence, $f$ itself terminates. Yet, $\chi^{in}$ could cause non-termination of the caller of $f$.

However, the entry procedure has no caller that would not terminate. Hence, no such $\chi^{in}$ can exist, which contradicts the assumption. □

## 5.3. Context-Sensitive Summaries

The key idea of interprocedural analysis is to avoid re-analysing procedures that are called multiple times. For that reason, Algorithm 1 first checks whether it can re-use already computed information. For that purpose, summaries are stored as implications

$CallCtx^o \Rightarrow Sum^o$. As the call graph is traversed, the possible calling contexts $CallCtx^o_{h_i}$ for a procedure $h$ are collected over the call sites $i$ and joined together.

— Subroutine $NeedToReAnalyze^o(h, CallCtx^o_{h_i})$ (Line 5 in Algorithm 1) checks whether the current calling context $CallCtx^o_{h_i}$ is subsumed by calling contexts $\bigvee_j CallCtx^o_{h_j}$ that we have already encountered, and if so, $Sums^o[h]$ is reused; otherwise $h$ needs to be recomputed in calling context $CallCtx^o_{h_i}$ and the obtained invariant and summaries are *join*ed conjunctively with previously inferred information as follows.

— Subroutine $join^o(h, (Sum^o_{h_i}, Inv^o_{h_i}), CallCtx^o_{h_i})$ in Algorithm 1 performs the following operation: $Sums^o[h] \leftarrow Sums^o[h] \wedge (CallCtx^o_{h_i} \Rightarrow Sum^o_{h_i})$, and $Invs^o[h] \leftarrow Invs^o[h] \wedge (CallCtx^o_{h_i} \Rightarrow Inv^o_{h_i})$.

The same considerations apply to termination arguments and preconditions. We store under-approximating information (e.g., $Precond^u_f$) in its complemented form $\widetilde{Precond^u_f}$ so that we can use $join^o$ instead of implementing the De-Morgan-dual $join^u$ operator.

The termination status in Algorithm 4 is joined as follows. $join(termStatus_1, termStatus_2)$ means that $termStatus_1$ is set to ***terminating*** if $termStatus_2$ is ***terminating*** and $termStatus_1$ is either ***terminating*** or has not been initialised (i.e., the corresponding function has not been analysed yet). $termStatus_1$ is set to *non-terminating* if $termStatus_2$ is *non-terminating*. Otherwise $termStatus_1$ is set to *potentially non-terminating*.

## 6. IMPLEMENTATION

We have implemented the algorithm in 2LS [2LS 2016],[5] a static analysis tool for C programs built on the CPROVER framework, using MiniSat 2.2.1 as back-end solver. Other SAT and SMT solvers with incremental solving support would also be applicable. Our approach enables us to use a single solver instance per procedure to solve a series of second-order queries as required by Algorithm 1. It is essential to use incremental solving as our synthesis algorithms make thousands of very similar solver calls. Architectural settings (e.g., bitwidths) can be provided on the command line.

**Bitvector Width Extension** As aforementioned, integers wrap around on most architectures when they over/underflow, which can result in non-terminating behaviour. Let us consider the following example, which we give as input to Algorithm 2:

```
void f() {
  signed char x;
  while(1) x++;
}
```

The ranking function synthesis aims to compute a value for template parameter $\ell$ such that $\ell(x{-}x') > 0$ holds for all $x, x'$ under the transition relation $x'{=}x{+}1$ and the computed invariant *true*.

Thus, assuming that the current value for $\ell$ is $-1$, the constraint to be solved (Algorithm 2 Line 5) is

$$true \wedge x'{=}x{+}1 \wedge \neg(-1 \cdot (x{-}x'){>}0),$$

which is equivalent to $\neg(-1(x{-}(x{+}1)){>}0)$. While for mathematical integers this is satisfiable, it is unsatisfiable for signed bit-vectors due to overflows. For $x{=}127$, the overflow happens such that $x{+}1{=}{-}128$. Thus, $\neg(-1 \cdot (127{-}({-}128)){>}0)$ becomes $\neg(1{>}0)$, which makes the constraint unsatisfiable, and we would incorrectly conclude that $-x$ is a ranking function, which does not hold for signed bitvector semantics. However, if

[5]The source code of the tool and instructions for its usage can be found on http://www.cprover.org/2LS.

we extend the bitvector width to $k{=}9$ such that the arithmetic in the template does not overflow, then $\neg(-1\cdot((\mathit{signed}_9)127-(\mathit{signed}_9)(-128))>0)$ evaluates to $\neg((-1\cdot 255)>0)$, where $\mathit{signed}_k$ is a cast to a $k$-bit signed integer. Now, $x{=}127$ is a model that shows that $-x$ is not a valid ranking function.

For these reasons to retain soundness, we extend the bit-width of signed and unsigned integer operands to integers that can hold the result of the operation without over- or underflow, e.g., one additional bit for additions and doubling the size plus one bit for multiplications. The maximum bit-width required depends on the shape of the template. Since our templates contain a finite number of operations, the maximum bit-width is finite.

Floating-point numbers do not require extensions as they overflow (resp. underflow) to infinity (resp. minus infinity), which does not impact soundness.

**Optimisations** Our ranking function synthesis algorithm searches for coefficients $\boldsymbol{\ell}$ such that a constraint is unsatisfiable. However, this may result in enumerating all the values for $\boldsymbol{\ell}$ in the range allowed by its type, which is inefficient. In many cases, a ranking function can be found for which $\ell_j \in \{-1,0,1\}$. In our implementation, we have embedded Algorithm 2 into an outer refinement loop which iteratively extends the range for $\boldsymbol{\ell}$ if a ranking function could not be found. We start with $\ell_j \in \{-1,0,1\}$, then we try $\ell_j \in [-10,10]$ before extending it to the whole range.

**Further Bounds** As explained in Algorithm 2, we bound the number of lexicographic components (default 3), because otherwise Algorithm 2 does not terminate if there is no number $n$ such that a lexicographic ranking function with $n$ components proves termination.

Since the domains of $\boldsymbol{x},\boldsymbol{x}'$ in Algorithm 2 and of $\boldsymbol{x}^{in}$ in Algorithm 3 might be large, we limit also the number of iterations (default 20) of the *while* loops in these algorithms. In the spirit of bounded model checking, these bounds only restrict completeness, i.e., there might exist ranking functions or preconditions which we could have found for larger bounds. The bounds can be given on the command line.

## 7. EXPERIMENTS

We performed experiments to support the following claims:

(1) Interprocedural termination analysis (IPTA) is faster than monolithic termination analysis (MTA).
(2) The precision of IPTA is comparable to MTA.
(3) 2LS is competitive with termination analysis tools on procedural programs.
(4) 2LS's analysis is bit-precise.
(5) 2LS computes valuable preconditions for termination.

IPTA is the approach presented in this article; MTA is IPTA applied to the program where all procedures have been inlined.

We used the *product line* benchmarks of the [SV-COMP 2016] benchmark repository. In contrast to other categories, this benchmark set contains programs with non-trivial procedural structure. This benchmark set contains 597 programs with 1100 to 5700 lines of code (2705 on average),[6] 33 to 136 procedures (67 on average), and four to ten loops (5.5 on average). Of these benchmarks, 264 terminate universally, whereas 333 never terminate when starting from `main`.

The experiments were run on a Xeon E3-1270 at 3.6 GHz running CentOS 7.2 with 64-bit binaries. Memory and CPU time were restricted to 16 GB and 1800 seconds per

[6]Measured using `cloc` 1.53.

Table IV: Tool comparison

| | expected | 2LS IPTA | 2LS MTA | TAN | Ultimate | llvm2kittel+ T2 | llvm2kittel+ KITTeL |
|---|---|---|---|---|---|---|---|
| terminating | 264 | 263 | 234 | 18 | 150 | 6 | 178 |
| non-terminating | 333 | 320 | 328 | 3 | 325 | 135 | – |
| potentially non-terminating | — | 14 | 35 | 425 | 0 | 0 | 4 |
| timed out | — | 0 | 0 | 151 | 118 | 61 | 233 |
| out of memory | — | 0 | 0 | 0 | 4 | 111 | 182 |
| errors | — | 0 | 0 | 0 | 0 | 284 | 0 |
| total run time (h) | — | 0.35 | 1.88 | 92.8 | 63.6 | 42.9 | 144.9 |

benchmark, respectively (following [Roussel 2011]). We have observed that interval templates are sufficient to obtain reasonable precision.[7]

**Modular termination analysis is fast** We compared IPTA with MTA. Table IV shows that the total analysis time of IPTA is 5.3 times lower than the time taken by MTA. The geometric mean speed-up of IPTA w.r.t. MTA on those benchmarks for which both approaches return a definitive answer (terminating or non-terminating) is 2.02.

**Modular termination analysis is precise** Again, we compare IPTA with MTA. Table IV shows that IPTA proves 99.6 % of the terminating benchmarks, whereas 88.6 % were proven by MTA. MTA can prove 98.5 % of the non-terminating benchmarks including eight benchmarks that IPTA classifies potentially non-terminating due to imprecision introduced by the use of summaries to handle function calls. Therefore neither approach is strictly better than the other.

**2LS is competitive with existing termination analysis tools on procedural programs** We tried to compare 2LS with 10 termination tools for C programs. We succeeded to run the following 4 tools, namely TAN [TAN 2014], Ultimate [Heizmann et al. 2016], T2 [T2 2016], and KITTeL [KiTTeL/KoAT 2016]. For the latter two tools we used the front-end llvm2KITTeL [llvm2KITTeL 2016] for generating the required input formats. Moreover, we tried AProVE [AProVE 2016], CppInv [CppInv 2015] and SeaHorn [SeaHorn 2016], but these tools abort with errors (either during parsing or later when translating into an SMT formula) on all benchmarks. We include these results in the replication package. Unfortunately, the tools Loopus [Loopus 2014], ARMC [Podelski and Rybalchenko 2007], FuncTion [FuncTion 2015], and HipTNT [Le et al. 2014] have limitations regarding the subset of C that they can handle that make them unable to analyze any of the benchmarks out of the box. We provide details on our experiences with all these tools on our website.[8]

TAN [Kroening et al. 2010] and KITTeL [Falke et al. 2012] support bit-precise C semantics. Ultimate uses mathematical integer reasoning but tries to ensure conformance with bit-vector semantics. Also, Ultimate uses a semantic decomposition of the program [Heizmann et al. 2014] to make its analysis efficient.

For each of the tools, Table IV lists the number of instances solved, timed out, run out of memory or aborted because of an internal error. None of the tools reported wrong results w.r.t. the expected outcome (column "expected").

[7]We provide a replication package for the experiments at http://www.cprover.org/termination/modular/replication-package.tgz.
[8]http://www.cprover.org/termination/modular/results-logs.txt.

```
void ex15(int m, int n, int p, int q) {
  for (int i = n; i >= 1; i = i - 1)
    for (int j = 1; j <= m; j = j + 1)
      for (int k = i; k <= p; k = k + 1)
        for (int l = q; l <= j; l = l + 1)
          ;
}
```

Fig. 7: Example ABC_ex15.c from the Loopus benchmarks

Ultimate proves 56.8 % of the terminating benchmarks correctly, and 97.6 % of the non-terminating ones. T2 proved 2.3 % of the terminating benchmarks and 40.5 % of the non-terminating ones, whereas KITTeL proved 67.4 % of the terminating ones. KITTeL cannot prove non-termination. The comparison with the latter two tools suffered from the deficiencies of the available front-ends. llvm2KITTeL timed out on some benchmarks and was unable to generate syntactically correct T2 files for almost half the benchmarks (error).

This suggests that current termination tools are quite specialised in handling and performing well on different sets of benchmarks, which makes it difficult to compare them in a fair manner.

Our results show that the technique implemented in 2LS is very efficient in analyzing procedural programs. However, one has to be careful in comparing the run times as the tools have different capabilities w.r.t. finding termination and non-termination arguments. In particular, 2LS uses relatively weak abstractions (linear lexicographic ranking functions and very simple polyhedral invariants), which make it very fast, but these abstractions are not expressive enough to provide as complex termination arguments as the other tools are able produce. Implementing more powerful domains is ongoing work.

Regarding non-termination, 2LS can currently only show that a program never terminates for all inputs, whereas Ultimate and T2 can show that there exists a non-terminating execution for some inputs. An extension of 2LS w.r.t. this capability is ongoing work.

**2LS's analysis is bit-precise** To demonstrate that one has to be careful when using tools that are not bit-precise for proving termination of C programs, we compared 2LS with Loopus, a tool that uses mathematical integers, on a collection of 15 benchmarks (ABC_ex01.c to ABC_ex15.c) taken from the Loopus benchmark suite [Loopus 2014]. While they are short (between 7 and 41 LOC), the main characteristic of these programs is the fact that they exhibit different terminating behaviours for mathematical integers and bit-vectors. For illustration, ABC_ex15.c (Figure 7) terminates with mathematical integers, but not with machine integers if, for instance, m equals INT_MAX. Next, we summarise the results of our experiments on these benchmarks when considering machine integers:

— Only two of the programs terminate, and are correctly identified by both 2LS and Loopus.
— For the rest of the 13 non-terminating programs, Loopus claims they terminate, whereas 2LS correctly classifies nine as potentially non-terminating (including ABC_ex15.c in Figure 7) and times out on four.

As a further example, let us consider the two procedures in Figure 8, which involve *floating-point arithmetic*. We assume that the argument for x that is passed to the

```
void h(float x)
{
  while(x>0.0f)
    x *= 0.1f;
}
```

```
void g(float x)
{
  while(x>0.0f)
    x *= 0.9f;
}
```

Fig. 8: Examples of loops with floating-point arithmetic

procedures is not NaN, i.e., `-FLT_MAX<=x && x<=FLT_MAX` (using the constants defined in `float.h`). For NaN these loops would trivially terminate because any comparison with NaN is *false*. Obviously, a tool that interprets `x` as a rational or real-valued variable would classify both procedures non-terminating. However, 2LS uses a bit-precise interpretation of `x` and determines that `h` is terminating whereas `g` is non-terminating (with round-to-nearest/ties-to-even rounding mode). The reason why this happens is that `x` gets eventually rounded to `0.0f` in `h`, whereas in `g` it keeps being rounded up to the `FLT_MIN` (the smallest representable floating-point number above zero) and therefore never reaches zero.

**2LS computes valuable preconditions for termination** This experiment was performed on benchmarks extracted from Debian packages and the linear algebra library CLapack.

The quality of preconditions, i.e., usability or ability to help the developer to spot problems in the code, is difficult to quantify. We give several examples where functions terminate conditionally. The `abe` package of Debian contains a function, given in Figure 9, where increments of the loop counter are not constant but dynamically depend on the dimensions of an image data structure. We make assumptions on the maximum image dimensions (using `__CPROVER_assume`), which are justified to prevent the functionally incorrect overflow behaviour in the assignments to `pos.x` and `pos.y`. We infer the precondition $img.h > 0 \wedge img.w > 0$.

The program in Figure 10 is a code snippet taken from the summation procedure `sasum` within [CLAPACK 2014], the C version of the popular LAPACK linear algebra library. The loop in procedure `f` does not terminate if $incx = 0$. If $incx > 0$ ($incx < 0$), the termination argument is that $i$ increases (decreases). Therefore, $incx \neq 0$ is a termination precondition for `f`. In this example, we make assumptions on the array size and the increment `incx`. Note that lifting these assumptions results in a very complex precondition due to overflows of $i$. This demonstrates a limitation of our current implementation that uses convex domains that cannot express these preconditions; we would therefore report the precondition *false* in this case.

The example in Figure 11 is taken from the benchmark `basename` in the `busybox`-category of SV-COMP 2016, which contains simplified versions of Debian packages. The termination of function `full_write` depends on the return value of its callee function `safe_write`. Here, we infer the calling context $cc > 0$, i.e., the contract for the function `safe_write`, such that the termination of `full_write` is guaranteed. Given a proof that `safe_write` terminates and returns a strictly positive value regardless of the arguments it is called with, we can conclude that `full_write` terminates universally.

## 8. LIMITATIONS, RELATED WORK AND FUTURE DIRECTIONS

Our approach makes significant progress towards analysing real-world software, advancing the state of the art of termination analysis of large programs. Conceptually, we decompose the analysis into a sequence of well-defined second-order predicate logic formulae with existentially quantified predicates. In addition to [Grebenshchikov et al. 2012], we consider context-sensitive analysis, under-approximate backwards analysis,

```
struct SDL_Surface {
  unsigned int h;
  unsigned int w;
};
struct SDL_Rect
{
  signed short int x;
  signed short int y;
  unsigned short int h;
  unsigned short int w;
};

void createBack(struct SDL_Surface back_surf, struct SDL_Surface img)
{
  __CPROVER_assume(back_surf.w <= 16383 &&
                   back_surf.h <= 16383 &&
                   0 <= img.w && img.w <= 16383 &&
                   0 <= img.h && img.h <= 16383);
  struct SDL_Rect pos;
  for(int x = 0; !(x >= back_surf.h); x += img.h) {
    for(int y = 0; !(y >= back_surf.w; y += img.w) {
      pos.x = (signed short int)x;
      pos.y = (signed short int)y;
      // SDL_UpperBlit(&img, NULL, &back_surf, &pos);
      // ...
    }
  }
}
```

Fig. 9: Example `createBack` from Debian package `abe`

```
int f(int *sx, int n, int incx) {
  __CPROVER_assume(1 <= n && n <= 32768 &&
                  -32768 <= incx && incx <= 32768);
  int nincx = n * incx;
  int stemp = 0;
  for (int i = 0; incx < 0 ? i >= nincx: i <= nincx; i += incx) {
    stemp += sx[i-1];
  }
  return stemp;
}
```

Fig. 10: Non-unit increment from `CLapack`

```
signed long int full_write(
    signed int fd,
    const void *buf,
    unsigned long int len,
    signed long int cc) {
  signed long int total = (signed long int)0;
  for( ; !(len == 0ul); len = len - (unsigned long int)cc) {
    cc=safe_write(fd, buf, len);
    __CPROVER_assume(-1 <= cc && cc <= 1);
    if(cc < 0l) {
      if(!(total == 0l))
        return total;
      return cc;
    }
    total = total + cc;
    buf = (const void *)((const char *)buf + cc);
  }
}
```

Fig. 11: Example from SV-COMP 2016 busybox

and make the interaction with termination analysis explicit. Notably, these seemingly tedious formulae are actually solved by our generic template-based synthesis algorithm, making it an efficient alternative to predicate abstraction.

An important aspect of our analysis is that it is bit-precise. As opposed to the synthesis of termination arguments for linear programs over integers (rationals) [Cook et al. 2006; Lee et al. 2012; Ben-Amram and Genaim 2013; Podelski and Rybalchenko 2004; Heizmann et al. 2013; Bradley et al. 2005a; Cook et al. 2013; Ben-Amram and Genaim 2014; Gonnord et al. 2015], this subclass of termination analyses is substantially less covered. Moreover, Cook et al. [2006] construct a termination argument as a disjunction of ranking functions for paths, which has the drawback of having to consider the transitive closure of the transition relation when checking for disjunctive well-foundedness of the termination argument. We incrementally (in the number of components) construct a lexicographic ranking function, which is disjunctively well-founded by construction.

While Kroening et al. [2010] and Cook et al. [2010] present methods 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. Falke et al. [2012] and Hensel et al. [2016] present restricted encodings of machine integer semantics into integer arithmetic. Maurica et al. [2016] use rational enclosures of floating point arithmetic in order to reuse existing techniques that use reasoning over rationals. David et al. [2015] synthesise intraprocedural termination arguments in the form of unrestricted boolean functions.

There are still a number of limitations to be addressed, all of which connect to open challenges subject to active research. While some are orthogonal (e.g., data structures, strings, refinement) to our interprocedural analysis framework, others (recursion, necessary preconditions) require extensions of it. In this section, we discuss related work, as well as, characteristics and limitations of our analysis, and future directions (cost analysis and concurrency).

**Dynamically allocated linked data structures** We currently ignore such data structures. This limitation could be lifted by using specific abstract domains. For illustration, let us consider the following example traversing a singly-linked list.

```
List x;
while (x != NULL) { x = x->next; }
```

Deciding the termination of such a program requires knowledge about the shape of the data structure pointed by $x$, namely, the program only terminates if the list is acyclic. Thus, we would require an abstract domain capable of capturing such a property and also relate the shape of the data structure to its length. Similar to Cook et al. [2013], we could use a technique introduced by Magill et al. [2010] in order to abstract heap-manipulating programs to arithmetic ones; they introduce additional ghost variables that symbolically record a snapshot of the depths of pieces of the heap used by the list. Another option is using an abstract interpretation based on separation logic formulae that tracks the depths of pieces of heaps similarly to Berdine et al. [2006] and Manevich et al. [2016]. Examples of analysers that do consider the heap are AProVE [Giesl et al. 2017], JULIA [Spoto et al. 2010], and COSTA [Albert et al. 2007]. The former focuses on the termination of term rewrite systems and, in order to analyze programs, it transforms them into a symbolic execution graph. Sharing effects of heap operations in Java, pointer arithmetic and memory safety in C are handled when generating this graph. The termination analysis in JULIA is based on a path-length abstraction and it is targeted at Java bytecode. As opposed to our analysis, none of these generates preconditions for termination or is bit-precise. When applying these solution, we would assume that the length of each list is bounded by a constant. Note that a bound such as $2^{64}$ is sufficient to verify every program running on a 64-bit processor.

**Strings and arrays** Similar to dynamically allocated data structures, handling strings and arrays requires specific abstract domains. String abstractions (e.g. [Dor et al. 2001]) that reduce null-terminated strings to integers (indices, length, and size) are usually sufficient in many practical cases; scenarios where termination is dependent on the content of arrays (e.g. [Albert et al. 2014] are much harder and would require quantified invariants [McMillan 2008]. Note that it is favourable to run a safety checker before the termination checker. The latter can assume that assertions for buffer overflow checks hold which strengthens invariants and makes termination proofs easier.

**Recursion** We currently use downward fixed point iterations for computing calling contexts and invariants that involve summaries (see Remark 5.7). This is cheap but gives only imprecise results in the presence of recursion, which would impair the termination analysis. We could handle recursions by detecting cycles in the call graph and switching to an upward iteration scheme in such situations. Moreover, an adaptation regarding the generation of the ranking function templates is necessary. An alternative approach would be to make use of the theoretic framework presented by Podelski et al. [2005] for verifying total correctness and liveness properties of while programs with recursion that could be used for the verification of termination properties.

**Template refinement** We currently use interval templates together with heuristics for selecting the variables that should be taken into consideration. This is often sufficient in practice, but it does not exploit the full power of the machinery in place. While counterexample-guided abstraction refinement (CEGAR) techniques are prevalent in predicate abstraction [Clarke et al. 2000], attempts to use them in abstract interpretation are rare [Ranzato et al. 2008]. We consider our template-based abstract interpretation that automatically synthesises abstract transformers more amenable to refinement techniques than classical abstract interpretations where abstract transformers are implemented manually.

**Sufficient preconditions to termination** Conditional termination has recently attracted increased interest [Cook et al. 2008; Bozga et al. 2012; Ganty and Genaim 2013; Massé 2012; Massé 2014; Urban and Miné 2014; Urban and Miné 2017]. In this paper, we compute *sufficient* preconditions, i.e., under-approximating preconditions to termination via computing over-approximating preconditions to potential non-termination. The same concept is used by other works [Cook et al. 2008; Bozga et al. 2012; Ganty and Genaim 2013; Urban and Miné 2014]. However, they consider only a single procedure and do not leverage their results to perform interprocedural analysis on large benchmarks which adds, in particular, the additional challenge of propagating under-approximating information up to the entry procedure. Moreover, in contrast to Cook et al. [2008], who use a heuristic FINITE-operator left unspecified for bootstrapping their preconditions, our bootstrapping is systematic through constraint solving.

The inference of conditions under which programs are guaranteed to terminate has also been an active area of research in the context of logic programming. In [King and Lu 2002], the authors use a backward analysis to infer conditions on a query which, if satisfied, guarantee that resulting derivations satisfy propertis such as termination, whereas in [Genaim and Codish 2005] Genaim and Codish combine traditional termination analysis and backwards analysis to infer modes for which a logic program is guaranteed to terminate. In [Mesnard 1996], Mesnard et al. compute a termination condition ensuring that the Prolog computation tree for a certain goal is finite. Urban and Miné [2017] aim at inferring sufficient precondition for certain subclasses of liveness properties.

We could compute *necessary* preconditions by computing over-approximating preconditions to non-termination (and negating the result). This requires a method for proving that there exist non-terminating executions, which is a well-explored topic. While Gupta et al. [2008] dynamically enumerate lasso-shaped candidate paths for counterexamples, and then statically prove their feasibility, Chen et al. [2014] prove non-termination via reduction to safety proving and Le et al. [2015] use bi-abduction to construct summaries of terminating and non-terminating behaviours for each method. Massé [2014] uses policy iteration techniques in order to compute an over-approximation of the potentially non-terminating states. In order to prove both termination and non-termination, Harris et al. [2010] compose several program analyses (termination provers for multi-path loops, non-termination provers for cycles, and safety provers).

**Cost analysis** A potential future application for our work is cost and resource analysis. Instances of this type of analyses are worst case execution time (WCET) analysis [Wilhelm et al. 2008; Knoop et al. 2011; Knoop et al. 2012], as well as bound and amortised complexity analysis [Alias et al. 2010; Albert et al. 2012; Sinn et al. 2014; Flores-Montoya and Hähnle 2014; Brockschmidt et al. 2016]. A symbolic bound is calculated by expressing the complexity function as a system of recurrence relations; combinatorial techniques are then used to solve these relations. Automatic techniques for computing closed-form solutions of recurrence relations are based on rewriting rules [Métayer 1988; Rosendahl 1989] or solving difference equations using symbolic algebra manipulation systems [Wegbreit 1975; Knoop et al. 2011; Knoop et al. 2012]. The control flow refinement approach [Gulwani et al. 2009; Chen et al. 2013] instruments a program with counters and uses progress invariants to compute worst case or average case bounds.

**Concurrency** Our current analysis handles single-threaded C programs. We envisage extending the analysis to multi-threaded C programs. One way of extending the analysis to multi-threaded programs is using the rely-guarantee technique which is proposed by Jones [1983], and explored in several works [Cook et al. 2007; Gupta et al. 2011; Popeea and Rybalchenko 2012; Albert et al. 2017] for termination analysis. In our setting, the predicates for environment assumptions can be used in a similar way as invariants and summaries are used in the analysis of sequential programs. Other approaches to checking termination of multi-threaded programs characterize the existence of non-terminating executions as concurrent traces and apply causality-based transformation rules to refine them until a contradiction can be shown [Kupriyanov and Finkbeiner 2014].

## 9. CONCLUSIONS

Many termination provers mainly target small, hard programs, and consequently, the termination analysis of larger code bases has received little attention. We present an algorithm for *interprocedural termination analysis* for non-recursive programs. We describe in full detail the entire machinery necessary to perform such an analysis. Our approach relies on a bit-precise static analysis combining SAT/SMT solving, template polyhedra and lexicographic, linear ranking function templates. We provide an implementation of the approach in the static analysis tool 2LS, and demonstrate the applicability of the approach to programs with thousands of lines of code.

**Acknowledgements**

The authors would like to thank the anonymous reviewers for their in-depth comments, which have led to significant improvements of the paper.

## REFERENCES

2LS 2016. 2LS: Static Analyzer and Verifier. (2016). http://www.cprover.org/2LS (version 0.4).

Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Guillermo Román-Díez. 2014. Conditional termination of loops over heap-allocated data. *Science of Computer Programming* 92 (2014), 2–24.

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2007. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Formal Methods for Components and Objects (LNCS)*, Vol. 5382. Springer, 113–132.

Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2012. Cost analysis of object-oriented bytecode programs. *Theor. Comput. Sci.* 413, 1 (2012), 142–159.

Elvira Albert, Antonio Flores-Montoya, Samir Genaim, and Enrique Martin-Martin. 2017. Rely-Guarantee Termination and Cost Analyses of Loops with Concurrent Interleavings. *Journal of Automated Reasoning* 59, 1 (2017), 47–85.

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis Symposium (LNCS)*, Vol. 6337. Springer, 117–133.

AProVE 2016. Automated Program Verification Environment (AProVE). (2016). https://sv-comp.sosy-lab.org/2016/downloads/AProVE2016.zip (SV-COMP 2016).

Amir M. Ben-Amram and Samir Genaim. 2013. On the Linear Ranking Problem for Integer Linear-constraint Loops. In *Principles of Programming Languages*. ACM, 51–62.

Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4 (2014), 26:1–26:55.

Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2006. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *Computer-Aided Verification (LNCS)*. Springer, 386–400.

Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 9636. Springer.

Marius Bozga, Radu Iosif, and Filip Konecný. 2012. Deciding Conditional Termination. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 7214. Springer, 252–266.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005a. Linear Ranking with Reachability. In *Computer-Aided Verification (LNCS)*. Springer, 491–504.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005b. Termination of Polynomial Programs. In *Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 3385. Springer, 113–129.

Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. 2015. Safety Verification and Refutation by $k$-Invariants and $k$-Induction. In *Static Analysis Symposium (LNCS)*, Vol. 9291. Springer, 145–161.

Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 13:1–13:50.

Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. 2015. Synthesising Interprocedural Bit-Precise Termination Proofs. In *Automated Software Engineering*. IEEE Computer Society, 53–64.

Hong Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter W. O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 8413. Springer, 156–171.

Hong Yi Chen, Supratik Mukhopadhyay, and Zheng Lu. 2013. Control Flow Refinement and Symbolic Computation of Average Case Bound. In *Automated Technology for Verification and Analysis (LNCS)*, Vol. 8172. Springer, 334–348. DOI:http://dx.doi.org/10.1007/978-3-319-02444-8_24

CLAPACK 2014. CLAPACK linear algebra library. (2014). http://www.netlib.org/clapack/cblas/sasum.c.

Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer-Aided Verification (LNCS)*, Vol. 1855. Springer, 154–169.

Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. 2008. Proving Conditional Termination. In *Computer-Aided Verification (LNCS)*, Vol. 5123. Springer, 328–340.

Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. 2010. Ranking Function Synthesis for Bit-Vector Relations. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 6015. Springer, 236–250.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *Programming Language Design and Implementation*. ACM, 415–426.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2007. Proving Thread Termination. In *Programming Language Design and Implementation*. ACM, 320–330.

Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*. Springer, 47–61.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages*. 238–252.

CppInv 2015. CppInv termination prover. (2015). http://www.lsi.upc.edu/~albert/cppinv-term-bin.tar.gz.

CVE 2009. Apache Common Vulnerabilities and Exposures CVE-2009-1890. (2009). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1890.

Cristina David, Daniel Kroening, and Matt Lewis. 2015. Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs. In *European Symposium on Programming*. Springer, 183–204.

Nurit Dor, Michael Rodeh, and Shmuel Sagiv. 2001. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *Static Analysis Symposium (LNCS)*, Vol. 2126. Springer, 194–212.

Stephan Falke, Deepak Kapur, and Carsten Sinz. 2012. Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In *Verified Software: Theories, Tools, Experiments (LNCS)*, Vol. 7152. Springer, 261–277.

Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *Programming Languages and Systems – 12th Asian Symposium*. 275–295.

FuncTion 2015. FuncTion termination prover. (2015). http://www.di.ens.fr/~urban/sv-comp2015.zip (version SV-COMP-2015).

Pierre Ganty and Samir Genaim. 2013. Proving Termination Starting from the End. In *Computer-Aided Verification (LNCS)*, Vol. 8044. Springer, 397–412.

Thomas M. Gawlitza and Helmut Seidl. 2007. Precise Relational Invariants Through Strategy Iteration. In *Computer Science Logic (LNCS)*, Vol. 4646. Springer, 23–40.

Samir Genaim and Michael Codish. 2005. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *Theory and Practice of Logic Programming* 5, 1-2 (2005), 75–91.

Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning* 58, 1 (2017), 3–31.

Laure Gonnord, David Monniaux, and Gabriel Radanne. 2015. Synthesis of ranking functions using extremal counterexamples. In *Programming Language Design and Implementation*. ACM, 608–618.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *Programming Language Design and Implementation*. 405–416.

Orna Grumberg and David E. Long. 1994. Model Checking and Modular Verification. *Transactions on Programming Languages and Systems* 16, 3 (1994), 843–871. DOI:http://dx.doi.org/10.1145/177492.177725

Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow Refinement and Progress Invariants for Bound Analysis. In *Programming Language Design and Implementation*. ACM, 375–385.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program Analysis as Constraint Solving. In *Programming Language Design and Implementation*. ACM, 281–292.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-termination. In *Principles of Programming Languages*. ACM, 147–158.

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate Abstraction and Refinement for Verifying Multi-threaded Programs. In *Principles of Programming Languages*. ACM, 331–344.

William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Static Analysis Symposium (LNCS)*, Vol. 6337. Springer, 304–319.

Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. 2016. Ultimate Automizer with Two-track Proofs (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 9636. Springer, 950–953. http://ultimate.informatik.uni-freiburg.de/ (special build based on version SV-COMP-2016).

Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. 2013. Linear Ranking for Linear Lasso Programs. In *Automated Technology for Verification and Analysis*. Springer, 365–380.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer-Aided Verification (LNCS)*, Vol. 8559. Springer, 797–813.

Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. 2016. Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution. In *Software Engineering and Formal Methods (LNCS)*, Vol. 9763. Springer, 234–252.

Cliff B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Trans. Program. Lang. Syst.* 5, 4 (Oct. 1983), 596–619. DOI:http://dx.doi.org/10.1145/69575.69577

Andy King and Lunjin Lu. 2002. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming* 2, 4-5 (2002), 517–547.

KiTTeL/KoAT 2016. KITTeL/KoAT termination prover. (2016). https://github.com/s-falke/kittel-koat (version 6ee36da).

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. 2011. Symbolic Loop Bound Computation for WCET Analysis. In *Perspectives of Systems Informatics, PSI (LNCS)*, Vol. 7162. Springer, 227–242.

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. 2012. r-TuBound: Loop Bounds for WCET Analysis (Tool Paper). In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*. 435–444.

Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *Computer-Aided Verification (LNCS)*, Vol. 6174. Springer, 89–103.

Andrey Kupriyanov and Bernd Finkbeiner. 2014. Causal Termination of Multi-threaded Programs. In *Computer Aided Verification (CAV) (LNCS)*, Vol. 8559. Springer, 814–830.

Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. 2014. A Resource-Based Logic for Termination and Non-termination Proofs. In *Formal Methods and Software Engineering (LNCS)*, Vol. 8829. Springer, 267–283. http://loris-5.d2.comp.nus.edu.sg/hiptnt/plus/.

Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and Non-termination Specification Inference. In *Programming Language Design and Implementation*. ACM, 489–498.

Wonchan Lee, Bow-Yaw Wang, and Kwangkeun Yi. 2012. Termination Analysis with Algorithmic Learning. In *Computer-Aided Verification (LNCS)*. Springer, 88–104.

Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 8413. Springer, 172–186.

Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic Optimization with SMT Solvers. In *Principles of Programming Languages*. ACM, 607–618.

llvm2KITTeL 2016. llvm2KITTeL converter. (2016). https://github.com/hkhlaaf/llvm2kittel (version e37be65e).

Loopus 2014. Loopus termination prover. (2014). http://forsyte.at/software/loopus/ with http://sourceforge.net/projects/virtualboximage/ files/Ubuntu%20Linux/11.10/ubuntu_11.10-x86.7z/download.

Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2010. Automatic Numeric Abstractions for Heap-Manipulating Programs. In *Principles of Programming Languages*. ACM, 211–222.

Roman Manevich, Boris Dogadov, and Noam Rinetzky. 2016. From Shape Analysis to Termination Analysis in Linear Time. In *Computer-Aided Verification (LNCS)*, Vol. 9779. Springer, 426–446.

Damien Massé. 2012. Proving Termination by Policy Iteration. *Electr. Notes Theor. Comput. Sci.* 287 (2012), 77–88.

Damien Massé. 2014. Policy Iteration-Based Conditional Termination and Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 8318. Springer, 453–471.

Fonenantsoa Maurica, Frédéric Mesnard, and Étienne Payet. 2016. Termination analysis of floating-point programs using parameterizable rational approximations. In *Symposium on Applied Computing*. ACM, 1674–1679.

Kenneth L. McMillan. 2008. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 4963. Springer, 413–427.

Frédéric Mesnard. 1996. Inferring Left-terminating Classes of Queries for Constraint Logic Programs. In *Logic Programing, Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming, Bonn, Germany, September 2-6, 1996*. 7–21.

Daniel Le Métayer. 1988. ACE: An Automatic Complexity Evaluator. *Transactions on Programming Languages and Systems* 10, 2 (1988), 248–266.

Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100. DOI:http://dx.doi.org/10.1007/s10990-006-8609-1

Andreas Podelski and Andrey Rybalchenko. 2004. Transition Invariants. In *Logic in Computer Science*. IEEE Computer Society, 32–41.

Andreas Podelski and Andrey Rybalchenko. 2007. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *Practical Aspects of Declarative Languages (LNCS)*, Vol. 4354. Springer. https://www7.in.tum.de/~rybal/armc/ (version August 2011).

Andreas Podelski, Ina Schaefer, and Silke Wagner. 2005. Summaries for While Programs with Recursion. In *European Symposium on Programming (LNCS)*, Vol. 3444. Springer, 94–107.

Corneliu Popeea and Andrey Rybalchenko. 2012. Compositional Termination Proofs for Multi-threaded Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 7214. Springer, 237–251.

Francesco Ranzato, Olivia Rossi-Doria, and Francesco Tapparo. 2008. A Forward-Backward Abstraction Refinement Algorithm. In *Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 4905. Springer, 248–262.

Mads Rosendahl. 1989. Automatic Complexity Analysis. In *Functional Programming Languages and Computer Architecture*. ACM, 144–156.

Olivier Roussel. 2011. Controlling a Solver Execution with the *runsolver* Tool. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 4 (2011), 139–144.

Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. 2005. Scalable Analysis of Linear Systems Using Mathematical Programming. In *Verification, Model Checking, and Abstract Interpretation (LNCS)*, Vol. 3385. Springer, 25–41.

Peter Schrammel. 2016. Challenges in Decomposing Encodings of Verification Problems. In *HCVS@ETAPS 2016 (EPTCS)*, Vol. 219. 29–32.

Peter Schrammel and Daniel Kroening. 2016. 2LS for Program Analysis (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 9636. Springer, 905–907.

SeaHorn 2016. SeaHorn, A Verification Framework. (2016). https://sv-comp.sosy-lab.org/2016/downloads/SeaHorn-0.1.0-Linux-x86_64.tar.gz.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Computer-Aided Verification (LNCS)*, Vol. 8559. Springer, 745–761.

Fausto Spoto, Fred Mesnard, and Étienne Payet. 2010. A Termination Analyzer for Java Bytecode based on Path-length. *ACM Trans. Program. Lang. Syst.* 32, 3 (2010).

SV-COMP 2016. (2016). https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16.

T2 2016. T2, Temporal Logic Prover. (2016). https://github.com/mmjb/T2 (version 90c5d0e).

TAN 2014. TAN Termination Prover. (2014). http://www.cprover.org/termination/ (version SV-COMP-2014).

Caterina Urban and Antoine Miné. 2014. A Decision Tree Abstract Domain for Proving Conditional Termination. In *Static Analysis Symposium (LNCS)*, Vol. 8723. Springer, 302–318.

Caterina Urban and Antoine Miné. 2017. Inference of ranking functions for proving temporal properties by abstract interpretation. *Computer Languages, Systems & Structures* 47 (2017), 77–103.

Ben Wegbreit. 1975. Mechanical Program Analysis. *Commun. ACM* 18, 9 (1975), 528–539.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools. *Transactions on Embedded Computing Systems* 7, 3, Article 36 (2008).