Termination and Non-termination Specification Inference

Ton Chanh Le[1] Shengchao Qin[2,3] Wei-Ngan Chin[1]

[1]Department of Computer Science, National University of Singapore
[2]School of Computing, Teesside University [3]Shenzhen University
{chanhle,chinwn}@comp.nus.edu.sg, s.qin@tees.ac.uk

Abstract

Techniques for proving termination and non-termination of imperative programs are usually considered as orthogonal mechanisms. In this paper, we propose a novel mechanism that analyzes and proves both program termination and non-termination at the same time. We first introduce the concept of second-order termination constraints and accumulate a set of relational assumptions on them via a Hoare-style verification. We then solve these assumptions with case analysis to determine the (conditional) termination and non-termination scenarios expressed in some specification logic form. In contrast to current approaches, our technique can construct a summary of terminating and non-terminating behaviors for each method. This enables modularity and reuse for our termination and non-termination proving processes. We have tested our tool on sample programs from a recent termination competition, and compared favorably against state-of-the-art termination analyzers.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Languages, Theory, Verification

Keywords Program termination and non-termination analysis; Bi-abductive inference; Hoare logic

1. Introduction

For the last ten years, we have seen a fruitful line of research on proving termination [2–5, 7, 11–17, 19, 26, 29, 30, 34, 37–39] and non-termination [1, 6, 8, 24, 31, 35, 47] of imperative programs. Although these techniques for proving program termination and non-termination are often considered separately, a termination prover might deploy its own non-termination analysis mechanism to search for a feasible counterexample when a termination proof fails. TREX [25] is one of the first works to combine two different termination and non-termination proving techniques to alternatively assist each other in a whole program analysis for non-recursive programs. Nevertheless, current techniques for proving non-termination are mostly standalone techniques to existing termination proving mechanisms.

PLDI'15, June 13–17, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2737993

To capture the termination and non-termination behaviors of each program, Le *et al.* [33] has recently proposed an integrated specification logic with three temporal predicates `Term M`, `Loop` and `MayLoop`, which denote, respectively, the scenarios for definite program termination (with a lexicographic ranking measure `M` made of a list of positive integers), definite non-termination (with an unreachable post-condition) and possible (unknown) non-termination. However, this framework currently requires temporal specifications to be given by programmers.

We propose in this paper a *modular* inference framework that can analyze *both* the termination and non-termination of each method in a program. This approach is novel in that it guides us to perform suitable case-splits on pre-conditions that lead to definite non-termination or definite termination, where possible. If a definite termination (or non-termination) case is not yet attained, we may perform a further case-split via an abductive inference [36] or decide to finish with a `MayLoop` classification to signify an unknown outcome. For each method, our inference mechanism incrementally constructs a *summary* of its termination, non-termination or unknown behaviors, so that it can be reused in the inference of the remaining methods higher-up in the calling hierarchy.

```
void foo (int x, int y)
  requires U_pr(x, y)
  ensures U_po(x, y);
{ if (x < 0) return;
  else foo(x + y, y); }
```

Figure 1. The `foo` example

To support termination and non-termination inference, we introduce *unknown* temporal pre- and post-predicates in our specification logic to capture termination or non-termination behaviors (that are to be resolved by our inference). For example, in Figure 1, the pair of unknown pre-predicate $\mathtt{U_{pr}(x,y)}$ and post-predicate $\mathtt{U_{po}(x,y)}$ in the specification of method `foo` denotes that the termination or non-termination status of `foo` is currently unknown. While the pre-predicate $\mathtt{U_{pr}(x,y)}$ in the method's precondition guides the overall inference process with suitable case-splits, the post-predicate $\mathtt{U_{po}(x,y)}$ in its postcondition is meant to capture the reachability or unreachability of the method's exits. This post-predicate will be strengthened to `false` in scenarios where `foo` is definitely non-terminating. Moreover, it can also be used to trivially determine base-case scenarios with immediate termination property. This combined use of unknown pre- and post-predicates is novel, since it allows us to modularly analyze each method (with the help of case-splits where needed) to obtain a comprehensive summary of the method's termination and non-termination characteristics.

Our main contribution lies in a single modular bi-abductive analysis that automatically infers sufficient preconditions respectively for termination and non-termination of each method in a program. The rest of this paper is organized as follows. Section 2 introduces the novel *unknown pre/post predicates* and illustrates their use for inferring termination and non-termination properties of the `foo` example. Section 3 summarizes the background on the termination

$$\begin{array}{rcl@{\qquad}rcl}
\mathit{hpred} & ::= & c(\overline{v}) \equiv \bigvee(\exists\overline{u}\cdot\rho) & \rho & ::= & \kappa \wedge \pi \\
\mathit{spec} & ::= & (\Psi_{\text{Pre}}, \Phi_{\text{Post}}) & \kappa & ::= & \texttt{emp} \mid v{\mapsto}d(\overline{u}) \mid c(\overline{v}) \mid \kappa_1{*}\kappa_2 \\
\Psi_{\text{Pre}} & ::= & \bigvee(\exists\overline{u}\cdot(\rho\wedge\theta)) & \pi & ::= & b \mid a \mid \pi_1{\wedge}\pi_2 \mid \neg\pi \mid \exists v\cdot\pi \\
\Phi_{\text{Post}} & ::= & \bigvee(\exists\overline{u}\cdot(\rho\wedge\texttt{U}_{\texttt{po}}(\overline{v}))) & b & ::= & \texttt{true} \mid \texttt{false} \mid v \mid b_1{=}b_2 \\
\theta & ::= & \texttt{Term}\ [\overline{e}] \mid \texttt{Loop} \mid & a & ::= & e_1{=}e_2 \mid e_1{<}e_2 \mid v{=}\texttt{null} \\
 & & \texttt{MayLoop} \mid \texttt{U}_{\texttt{pr}}(\overline{v}) & e & ::= & k \mid v \mid k{\times}e \mid e_1{+}e_2 \mid -e
\end{array}$$

where emp denotes an empty heap; $v{\mapsto}d(\overline{u})$ specifies a heap node of data type d; k is a constant; u, v are variables

Figure 2. A Specification Language

and non-termination reasoning with the temporal predicates. Section 4 shows how to generate a set of relational assumptions over the unknown temporal predicates. Section 5 introduces the inference algorithm to resolve the unknown predicates from their relational assumptions. Section 6 provides the experimental results. Section 7 discusses related work and Section 8 concludes our paper.

2. Overview of Our Approach

Specification Language. Le *et al.* [33] proposed three temporal predicates, `Term M`, `Loop` and `MayLoop`, to help reason about program termination and non-termination. For the current evaluation, we adopt these predicates as well as a rich underlying specification language proposed by [9] that is able to express both heap properties with separation logic (κ in Figure 2) and pure (non-heap) properties with Presburger arithmetic (π in Figure 2). This logic uses a fragment of separation logic with the separation conjunction $*$ to denote the disjointness of heap portions and the heap predicate *hpred* (Figure 2) to specify various data structures. Moreover, to simplify the presentation, we express a specification as a pair of pre- and post-condition (see *spec* in Figure 2). In our example programs, specifications will be written using the usual `requires...ensures...` form for better readability. Specifically for termination reasoning, we have designed the termination measure `M` as a (finite) list of arithmetic expressions $[\overline{e}]$ whose order is based on the lexicographic ordering $<_l$ (defined below) and $e{:}es$ denotes a non-empty list with e and es as its head and tail, respectively.

$$\frac{}{[\,] <_l e{:}\_} \qquad \frac{(e_1 < e_2) \vee (e_1 = e_2 \wedge es_1 <_l es_2)}{e_1{:}es_1 <_l e_2{:}es_2}$$

To facilitate termination and non-termination inference, we allow the use of an unknown temporal pre-predicate $\texttt{U}_{\texttt{pr}}(\overline{v})$ and a post-predicate $\texttt{U}_{\texttt{po}}(\overline{v})$ in the specification language to indicate the unknown termination status of a program. The solutions of these unknown predicates would be then derived by the inference mechanism, as shown next. Note that the inferred result for each unknown pre-predicate $\texttt{U}_{\texttt{pr}}(\overline{v})$ will be of the form $\bigvee(\pi \wedge \theta)$ with θ ranging over $\{\texttt{Term}\ [\overline{e}], \texttt{Loop}, \texttt{MayLoop}\}$; while the inferred result for each unknown post-predicate $\texttt{U}_{\texttt{po}}(\overline{v})$ will be in a guarded conjunction $\bigwedge(\pi \Rightarrow \mathit{post})$ with *post* being `true` or `false`. Such a guarded form is equivalent to a disjunctive form $\bigvee(\pi \wedge \mathit{post})$ when the set of guards are complete. These unknown temporal predicates are expressed in pure arithmetic domain. Heap-based properties in our logic are currently handled prior to termination analysis.

Illustrating Example. We now demonstrate how our inference mechanism derives the preconditions for termination and non-termination of method `foo` in Figure 1. Initially, the termination and non-termination behaviors of method `foo` are captured by a pair of unknown pre-predicate $\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y})$ and unknown post-predicate $\texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$. Like the other known temporal predicates for termination and non-termination reasoning, these unknown predicates are part of the specification logic's formulas and can therefore be reasoned in a similar way via Hoare-style verification. With the help of an enhanced entailment procedure, we shall prove that the precondition of each method call is always satisfied and the postcondition always holds at the end of the method body.

For example, the verification conditions (VCs) encountered by Hoare-style forward verification of method `foo` are:

(c_1) $\texttt{x}{<}0 \wedge \texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y}) \vdash \texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$

(c_2) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y} \wedge \texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y}) \vdash \texttt{U}_{\texttt{pr}}(\texttt{x}',\texttt{y}')$

(c_3) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y} \wedge \texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y}) \wedge \texttt{U}_{\texttt{po}}(\texttt{x}',\texttt{y}') \vdash \texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$

The first VC (c_1) is obtained from the base-case scenario when the post-condition of the `foo` method is being proven. The second VC (c_2) captures the proving of precondition for the recursive call, while the last VC (c_3) captures the entailment proving of the postcondition of method `foo` in the recursive branch. These VCs capture the unknown termination behaviors of both the caller (*i.e.* denoted by the pair of predicates $\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y})$ and $\texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$) and the callee (*i.e.* denoted by $\texttt{U}_{\texttt{pr}}(\texttt{x}',\texttt{y}')$ and $\texttt{U}_{\texttt{po}}(\texttt{x}',\texttt{y}')$).

For these unknown predicates, we attempt to derive the strongest possible post-predicate, where possible. As we intend to capture the unreachability of each post-predicate, the strongest post-predicate in our analysis is actually `false`. If our inference for falsity of post-predicates fails, we denote its possible reachability by `true` instead and then attempt to infer the weakest pre-predicate, where possible. The temporal pre-predicates are ordered by the following implication hierarchy `MayLoop` $\Rightarrow_r$ `Loop` and `MayLoop` $\Rightarrow_r$ `Term` $[\overline{e}]$. Amongst them, `MayLoop` is considered as the strongest one, which is analogous to `false` in the domain of logical specification. The intuition is that `MayLoop` can be used to denote the termination property of any program though such a use would form a rather poor specification, which is similar to how `false` could be naively (and redundantly) used as the precondition for any program. On the other hand, the `Loop` and `Term` $[\overline{e}]$ predicates are incomparable since they denote disjoint classes of programs (*i.e.* definitely non-terminating vs. definitely terminating programs, respectively). Our inference thus attempts to discover the weaker `Loop` and `Term` $[\overline{e}]$ for its unknown pre-predicate, where possible.

From the earlier VCs, we infer three relational assumptions where unknown pre-predicate $\texttt{U}_{\texttt{pr}}(\texttt{x}',\texttt{y}')$ is related inductively to an earlier pre-predicate $\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y})$ (see (a_2^0)), while unknown post-predicate $\texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$ is either expressed in base-case form (see (a_1^0)) or related inductively to an earlier occurrence of the post-predicate $\texttt{U}_{\texttt{po}}(\texttt{x}',\texttt{y}')$ (see (a_3^0)).

(a_1^0) $\texttt{x}{<}0 {\wedge} \texttt{true} \Rightarrow \texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$

(a_2^0) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y}{\wedge}\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y}) \Rightarrow \texttt{U}_{\texttt{pr}}(\texttt{x}',\texttt{y}')$

(a_3^0) $\texttt{x}{\geq}0 \wedge \texttt{x}'{=}\texttt{x}{+}\texttt{y} \wedge \texttt{y}'{=}\texttt{y}{\wedge}\texttt{U}_{\texttt{po}}(\texttt{x}',\texttt{y}') \Rightarrow \texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y})$

We derive inductive definitions for these unknown predicates, in order to give the best possible interpretations to their temporal predicates. In the case of post-predicate, we attempt to determine its reachability or unreachability, so that we can immediately decide on either (base-case scenario for) termination or (inductive-case scenario for) definite non-termination. From the relational assumption (a_1^0), we can immediately infer a base-case scenario $\texttt{x}{<}0$ where the `foo` method would terminate. The other two relational assumptions occur under a different scenario $\texttt{x}{\geq}0$ which neither indicates definite termination nor definite non-termination. From these partial instantiations on the two unknown temporal predicates, we refine them to the following definitions:

$$\begin{array}{lccc}
\texttt{U}_{\texttt{pr}}(\texttt{x},\texttt{y}) \equiv & \texttt{x}{<}0 \wedge \texttt{Term} & \vee & \texttt{x}{\geq}0 \wedge \texttt{U}^1_{\texttt{pr}}(\texttt{x},\texttt{y}) \\
\texttt{U}_{\texttt{po}}(\texttt{x},\texttt{y}) \equiv & (\texttt{x}{<}0 \Rightarrow \texttt{true}) & \wedge & (\texttt{x}{\geq}0 \Rightarrow \texttt{U}^1_{\texttt{po}}(\texttt{x},\texttt{y}))
\end{array}$$

where two auxiliary unknown predicates are introduced for the input scenario $\texttt{x}{\geq}0$. Note that `Term`, short for `Term` $[\,]$, is used to denote base-case termination scenario where its lexicographic ranking measure is trivially empty. Our unknown pre-predicate is being expressed as a disjunction on either known or unknown temporal resource constraints, while the post-predicate is being expressed as a guarded conjunction of either reachability (`true`), unreachability (`false`) or unknown. That is the two predicates are currently known for the input scenario $\mathtt{x}{<}0$ but unknown for the scenario $\mathtt{x}{\geq}0$.

As the precondition is now partially known, we could refine each (a_i^0) through a substitution with the partial definition of $\mathtt{U_{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U_{po}}(\mathtt{x},\mathtt{y})$ to get the new relational assumptions over the unknowns:

(a_{2a}^0) $\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{<}0{\wedge}\mathtt{U^1_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{Term}$

(a_{2b}^0) $\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{\geq}0{\wedge}\mathtt{U^1_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{U^1_{pr}}(\mathtt{x}',\mathtt{y}')$

(a_{3a}^0) $\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge (\mathtt{x}'{\geq}0{\Rightarrow}\mathtt{U^1_{po}}(\mathtt{x}',\mathtt{y}'))$
$\quad\Rightarrow (\mathtt{x}{\geq}0{\Rightarrow}\mathtt{U^1_{po}}(\mathtt{x},\mathtt{y}))$

The relational assumption (a_{2a}^0) describes the reachability of the base-case condition (*i.e.* $\mathtt{x}'{<}0$), denoted by `Term`, under the input scenario $\mathtt{x}{\geq}0$. We can thus attempt a termination proof by synthesizing a ranking function for $\mathtt{x}{\geq}0$ but this proof fails. We then try a non-termination proof by examining (a_{3a}^0) on unknown post-predicate to determine a condition for unreachability. Such condition must ensure that the base case is not reached in the next recursion, *i.e.* $\mathtt{x}'{\geq}0$, and we refer to this as *potential* non-termination pre-condition. Since $\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}$, the condition $\mathtt{x}{+}\mathtt{y}{\geq}0$ would be a trivial potential non-termination pre-condition. However, our inference engine would discover a better (or weaker) condition, namely $\mathtt{y}{\geq}0$, for definite non-termination with the help of abductive inference. With this, a case-split with the condition $\mathtt{y}{\geq}0$ and its negation $\mathtt{y}{<}0$ is used to refine the definitions for $\mathtt{U^1_{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U^1_{po}}(\mathtt{x},\mathtt{y})$ into:

$$\begin{array}{llcl}
\mathtt{U^1_{pr}}(\mathtt{x},\mathtt{y}) \equiv & \mathtt{y}{\geq}0 \wedge \mathtt{U^2_{pr}}(\mathtt{x},\mathtt{y}) & \vee & \mathtt{y}{<}0 \wedge \mathtt{U^3_{pr}}(\mathtt{x},\mathtt{y}) \\
\mathtt{U^1_{po}}(\mathtt{x},\mathtt{y}) \equiv & (\mathtt{y}{\geq}0 \Rightarrow \mathtt{U^2_{po}}(\mathtt{x},\mathtt{y})) & \wedge & (\mathtt{y}{<}0 \Rightarrow \mathtt{U^3_{po}}(\mathtt{x},\mathtt{y}))
\end{array}$$

Consequently, the following six specialized assumptions are derived from the earlier ones (a_{2a}^0), (a_{2b}^0) and (a_3^0).

(a_1^1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{<}0{\wedge}\mathtt{y}{\geq}0{\wedge}\mathtt{U^2_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{Term}$

(a_2^1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{\geq}0{\wedge}\mathtt{y}{\geq}0{\wedge}\mathtt{U^2_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{U^2_{pr}}(\mathtt{x}',\mathtt{y}')$

(a_3^1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y} \wedge (\mathtt{x}'{\geq}0{\wedge}\mathtt{y}'{\geq}0{\Rightarrow}\mathtt{U^2_{po}}(\mathtt{x}',\mathtt{y}'))$
$\quad\Rightarrow (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0{\Rightarrow}\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y}))$,

(a_4^1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{<}0{\wedge}\mathtt{y}{<}0{\wedge}\mathtt{U^3_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{Term}$

(a_5^1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{x}'{\geq}0{\wedge}\mathtt{y}{<}0{\wedge}\mathtt{U^3_{pr}}(\mathtt{x},\mathtt{y}) \Rightarrow \mathtt{U^3_{pr}}(\mathtt{x}',\mathtt{y}')$

(a_6^1) $\mathtt{x}{\geq}0{\wedge}\mathtt{x}'{=}\mathtt{x}{+}\mathtt{y}{\wedge}\mathtt{y}'{=}\mathtt{y} \wedge (\mathtt{x}'{\geq}0{\wedge}\mathtt{y}'{<}0{\Rightarrow}\mathtt{U^3_{po}}(\mathtt{x}',\mathtt{y}'))$
$\quad\Rightarrow (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0{\Rightarrow}\mathtt{U^3_{po}}(\mathtt{x},\mathtt{y}))$

The first three relational assumptions, $(a_1^1) - (a_3^1)$, form a group which will be analyzed together for the given input scenario $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0$. The next three relational assumptions, $(a_4^1) - (a_6^1)$, form another group for the input scenario $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0$.

The first group of relational assumptions, $(a_1^1) - (a_3^1)$, allows us to confirm a definite non-termination scenario, since we can use (a_3^1) to determine the unreachability of its post-predicate $\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y})$. By using the hypothesis $\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y}){\equiv}\mathtt{false}$ for both occurrences of the post-predicate $\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y})$ in (a_3^1), we can inductively determine the falsity (or unreachability) of $\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y})$. Note our use of inductive reasoning here which assumes the hypothesis that $\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y})$ is unreachable under pre-condition $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0$, in order to prove it.

The second group of relational assumptions, $(a_4^1) - (a_6^1)$, suggests us to prove the method's termination under the precondition $\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0$ first, since its base case (captured by (a_4^1)) is possibly reachable under this condition. This termination scenario is confirmed, once we have derived a lexicographic ranking measure $[\mathtt{x}]$ that is bounded and would moreover decrease with each recursive invocation for the pre-predicate $\mathtt{U^3_{pr}}(\mathtt{x},\mathtt{y})$ using (a_5^1).

As a summary of our combined analyses, we have effectively derived the following definitions for the two unknown predicates:

$$\begin{array}{ll}
\mathtt{U_{pr}}(\mathtt{x},\mathtt{y}) \equiv & \mathtt{x}{<}0{\wedge}\mathtt{Term} \vee \mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0{\wedge}\mathtt{Term}[\mathtt{x}] \\
 & \vee\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0{\wedge}\mathtt{Loop} \\
\mathtt{U_{po}}(\mathtt{x},\mathtt{y}) \equiv & (\mathtt{x}{<}0 \Rightarrow \mathtt{true}) \wedge (\mathtt{x}{\geq}0{\wedge}\mathtt{y}{<}0 \Rightarrow \mathtt{true}) \\
 & \wedge(\mathtt{x}{\geq}0{\wedge}\mathtt{y}{\geq}0 \Rightarrow \mathtt{false})
\end{array}$$

Note how the unknown temporal predicates $\mathtt{U^2_{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U^2_{po}}(\mathtt{x},\mathtt{y})$ are being resolved to be `Loop` and an unreachable `false` for input scenario $\mathtt{y}{\geq}0$, respectively. In contrast, the unknown predicates $\mathtt{U^3_{pr}}(\mathtt{x},\mathtt{y})$ and $\mathtt{U^3_{po}}(\mathtt{x},\mathtt{y})$ are being resolved to be $\mathtt{Term}\,[\mathtt{x}]$ and a reachable `true` state for input scenario $\mathtt{y}{<}0$, respectively.

Using the inferred predicate definitions, we can finally construct the following case-based specification ([20]) which fully captures termination and non-termination behaviors for method `foo`.

```
case {
  x < 0 → requires Term ensures true;
  x ≥ 0 → case {
    y < 0 → requires Term [x] ensures true;
    y ≥ 0 → requires Loop ensures false; }}
```

2.1 Other Examples

Our termination and non-termination inference is completely automated. By allowing unknown temporal predicates into functional correctness specifications, our inference mechanism can freely leverage on prior infrastructures to *(i)* handle a wider class of programs, and to *(ii)* improve the accuracy of the inference results. Note that prior safety specifications for the analyzed methods might be manually given or be automatically derived by other inference mechanisms, but they are orthogonal to our current proposal.

We list below some interesting examples to demonstrate how our inference mechanism works with programs that already have some safety specifications.

Nested Recursion. For examples with nested recursion, such as the Ackermann function and the McCarthy 91 function in Figure 3, some knowledge on its output may be crucial for the inference of their termination and non-termination properties. Without any specification, our inference mechanism returns incomplete summaries on the terminating and non-terminating behaviors of these two functions. The result for the Ackermann function is:

```
case {
  m=0         → requires Term ensures true;
  m<0 ∨ n<0   → requires Loop ensures false;
  m>0 ∧ n≥0   → requires MayLoop ensures true; }
```

While the inference shows that this function is terminating when $\mathtt{m}{=}0$ (base case) or non-terminating when $\mathtt{m}{<}0 \vee \mathtt{n}{<}0$, it cannot prove the termination of the function under the input scenario $\mathtt{m}{>}0{\wedge}\mathtt{n}{\geq}0$ since the value of the second argument in the last recursive call is unknown (or unbounded). However, with the stronger specification given in Figure 3(a), with an lower bound $\mathtt{res} \geq \mathtt{n}{+}1$ on the function's returned value, denoted by `res`, our inference mechanism can replace `MayLoop` in scenario $\mathtt{m}{>}0{\wedge}\mathtt{n}{\geq}0$ by $\mathtt{Term}\,[\mathtt{m},\mathtt{n}]$ where $[\mathtt{m},\mathtt{n}]$ is a valid lexicographic ranking function. Similarly, without specification, the inference only shows that the McCarthy 91 function terminates in its base case when $\mathtt{n}{>}100$. However, with the specification given in Figure 3(b), our inference can prove that the function terminates for all inputs.

While our termination inference mechanism does not directly infer postconditions, it has been made to work with other automated postcondition inference sub-systems, such as [23, 40]. Such postcondition inference sub-systems are orthogonal to our proposal, and can be leveraged to provide a more comprehensive solution for fully automated termination and non-termination inference.

```
int Ack (int m, int n)
  requires true   ensures res ≥ n+1;
{ if (m == 0) return n + 1;
  else if (n == 0) return Ack(m − 1, 1);
  else return Ack(m − 1,Ack(m, n − 1)); }
                  (a)
```

```
int Mc91 (int n)
  requires true
  ensures n≤100∧res=91 ∨ n>100∧res=n−10;
{ if (n > 100) return n − 10;
  else return Mc91(Mc91(n + 11)); }
                  (b)
```

Figure 3. Functions with Nested Recursion: Ackermann function (a) and McCarthy 91 function (b)

```
data node { node next; }

pred lseg(root, q, n) ≡ root=q ∧ n=0
  ∨ root↦node(p) ∗ lseg(p, q, n−1)

pred cll(root, n) ≡
  root↦node(p) ∗ lseg(p, root, n−1)
```

```
void append (node x, node y)
  requires lseg(x, null, n) ∧ x≠null
  ensures lseg(x, y, n);
  requires cll(x, n)   ensures true;
{ if (x.next == null) x.next = y;
  else append(x.next, y); }
```

Figure 4. Specification with Implementation for append method of two linked lists

Heap-Manipulating Programs. Our inference mechanism can be readily integrated into existing verification frameworks (such as [9], or even shape inference system [32]) that reason about safety properties of heap programs via separation logic [41]. Such an extension can be applied to help prove the termination and non-termination of heap-manipulating programs.

Figure 4 shows the specification and implementation (for the verification) of the method append that concatenates two linked lists x and y. With the separation conjunction $*$ and the points-to operator $\mapsto$ of separation logic, the heap predicate $\texttt{lseg}(\texttt{root}, \texttt{q}, \texttt{n})$ represents a list segment from root to q with n elements. This predicate can then be used in the declarations of other predicates, such as $\texttt{cll}(\texttt{root}, \texttt{n})$ for circular lists. Using these two predicates, we can capture two safety specifications of append in Figure 4.

In the first scenario when the input x is a null-terminating list with size n, our inference mechanism is able to show that the method append always terminates with the ranking function [n]. In the second scenario where x is a circular linked list, our inference can show that append is definitely non-terminating, after confirming (by induction) that its postcondition can be strengthened to false. These examples highlight the modular nature of our non-termination and termination inference mechanism, which can be built on top of other inference mechanisms.

3. Technical Background

So far we have illustrated a unified specification logic with three known temporal predicates: $\texttt{Term}\ [\overline{e}]$, Loop and MayLoop. Semantically, these predicates can be defined using resource capacities (on lower and upper bounds) of execution length, *i.e.* $\texttt{Term}\ [\overline{e}] =_{df} \texttt{RC}\langle 0, \texttt{f}([\overline{e}])\rangle$, $\texttt{Loop} =_{df} \texttt{RC}\langle\infty, \infty\rangle$, and $\texttt{MayLoop} =_{df} \texttt{RC}\langle 0, \infty\rangle$. The resource predicate $\texttt{RC}\langle \texttt{L}, \texttt{U}\rangle$ specifies a resource capacity with a lower bound L and an upper bound U. It is satisfied by each program state whose resource capacity (l, u) is subsumed by $(\texttt{L}, \texttt{U})$, *i.e.* $\texttt{L}{\leq}l$ and $u{\leq}\texttt{U}$. Note that the function $\texttt{f}([\overline{e}])$ obtains a finite bound through an order-embedding of $[\overline{e}]$ into naturals.

Verification conditions involving these temporal predicates can be discharged by a resource consumption entailment $\vdash_t$, that is used to account for (lower and upper bound) resources that are utilized by each code fragment. Such entailment can be used to analyze termination or non-termination property for some given method via resource reasoning. Given the temporal constraint θ_a associated with the current program state ρ and the temporal resource constraint θ_c (of some code fragment that must be executed), the entailment $\rho \wedge \theta_a \vdash_t \theta_c \blacktriangleright \theta_r$ checks whether the execution resource required by constraint θ_c can be met by the execution resource of constraint θ_a or not. If it succeeds, the entailment will return the remaining execution resource that is denoted by residue θ_r.

In terms of the actual execution capacity, this consumption entailment can be formalized by the following rule:

$$\frac{\rho \Rightarrow \texttt{U}_\texttt{c}{\leq}\texttt{U}_\texttt{a} \quad \texttt{L}_\texttt{r} = \texttt{L}_\texttt{a}{-_\texttt{L}}\texttt{L}_\texttt{c} \quad \texttt{U}_\texttt{r} = \texttt{U}_\texttt{a}{-_\texttt{U}}\texttt{U}_\texttt{c} \quad \rho \Rightarrow \texttt{L}_\texttt{r}{\leq}\texttt{U}_\texttt{r}}{\rho \wedge \texttt{RC}\langle \texttt{L}_\texttt{a}, \texttt{U}_\texttt{a}\rangle \vdash_t \texttt{RC}\langle \texttt{L}_\texttt{c}, \texttt{U}_\texttt{c}\rangle \blacktriangleright \texttt{RC}\langle \texttt{L}_\texttt{r}, \texttt{U}_\texttt{r}\rangle}$$

where two subtraction operators are designed to cater to an integer domain extended with the ∞ value (*i.e.* $\texttt{N}^\infty$):

$$\texttt{L}_1 -_\texttt{L} \texttt{L}_2 \equiv \mathit{min}\{\texttt{r} \in \texttt{N}^\infty \mid \texttt{r} + \texttt{L}_2 \geq \texttt{L}_1\}$$
$$\texttt{U}_1 -_\texttt{U} \texttt{U}_2 \equiv \mathit{max}\{\texttt{r} \in \texttt{N}^\infty \mid \texttt{r} + \texttt{U}_2 \leq \texttt{U}_1\},\ \mathit{if}\ \texttt{U}_1{\geq}\texttt{U}_2$$

These two operators are essentially integer subtraction operators, except that their results are never negative and such that $\infty{-_\texttt{L}}\infty = 0$ and $\infty{-_\texttt{U}}\infty = \infty$. They are formulated in this way to give the best (or largest) possible lower and upper bound values to denote the execution capacity of residue. In addition, the subtraction $\texttt{U}_\texttt{a} -_\texttt{U} \texttt{U}_\texttt{c}$ requires a check for upper bound execution capacity, namely $\rho \Rightarrow \texttt{U}_\texttt{c}{\leq}\texttt{U}_\texttt{a}$. This check is important to ensure that resource consumption is within the specified upper bound, and will also ensure that the residue is a valid resource capacity.

The resource implication operator $\Rightarrow_r$ on execution capacity, used in the implication hierarchy of known temporal predicates, can be defined based on the following subsumption relation:

$$\frac{\texttt{L}_1{\leq}\texttt{L}_2 \quad \texttt{U}_2{\leq}\texttt{U}_1}{\texttt{RC}\langle \texttt{L}_1, \texttt{U}_1\rangle \Rightarrow_r \texttt{RC}\langle \texttt{L}_2, \texttt{U}_2\rangle}$$

From this definition, MayLoop is the strongest pre-predicate in the subsumption hierarchy since it has the maximum execution capacity $(0, \infty)$. It can subsume either Loop (with execution capacity (∞, ∞)) or $\texttt{Term}\ [\overline{e}]$ (with execution capacity $(0, \texttt{f}([\overline{e}]))$) predicates. Note that the implication operator $\Rightarrow_r$ is only weakly related to the resource consumption entailment operator, $\vdash_t$, as follows:

$$(\theta_a \Rightarrow_r \theta_c) \Rightarrow \exists \theta_r \cdot \theta_a \vdash_t \theta_c \blacktriangleright \theta_r$$

For termination and non-termination inference, we have introduced unknown predicates $\texttt{U}_\texttt{pr}(\overline{v})$ for precondition and $\texttt{U}_\texttt{po}(\overline{v})$ for postcondition for each method, with $\texttt{U}_\texttt{pr}(\overline{v})$ denoting some execution capacity, and $\texttt{U}_\texttt{po}(\overline{v})$ specifying reachability of a method with a set of formal parameters $\overline{v}$. To support its inference, we will have to extend the resource entailment procedure to handle entailments between known and unknown temporal constraints.

The most general form of temporal entailment is $\rho \wedge \bigwedge_i \texttt{U}^i_\texttt{po}(\overline{v_i}) \wedge \theta_a \vdash_t \theta_c \blacktriangleright (\theta_r, \mathcal{R})$, where each $\texttt{U}^i_\texttt{po}(\overline{v_i})$ is an unknown post-predicate accumulated into the program state after a recursive method call. The temporal constraint θ_a in the antecedent of the entailment might be an unknown pre-predicate $\texttt{U}_\texttt{pr}(\overline{v})$ or a known temporal predicate. The temporal constraint θ_c can be either an unknown post-predicate $\texttt{U}_\texttt{po}(\overline{v})$ or a known predicate. The residue constraint θ_r denotes the residual capacity after entailment. Each relational assumption $\mathcal{R}$ for the unknown temporal predicates is a pre-requisite to ensure the validity of the entailment when either θ_a or θ_c is unknown. It is defined as below.

$$\begin{array}{rcl}
\textit{Prog} & ::= & \overline{\textit{tdecl}}\ \overline{\textit{meth}} \\
\textit{tdecl} & ::= & \texttt{data}\ c\ \{\ \overline{\textit{field}}\ \} \\
\textit{field} & ::= & t\ v \qquad\quad t ::= c \mid \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
\textit{meth} & ::= & t\ mn(\overline{[\texttt{ref}]\ t\ v})\ (\Psi_{\text{Pre}}, \Phi_{\text{Post}})\ \{e\} \\
e & ::= & \texttt{null} \mid k \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \\
 & & \texttt{new}\ c(\overline{v}) \mid e_1; e_2 \mid t\ v; e \mid mn(\overline{v}) \mid \\
 & & \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{return}\ v
\end{array}$$

where c is a data type name; mn is a method name;
k is a primitive constant; f is a field name; v is a variable

Figure 5. A Core Imperative Language

DEFINITION 1. *The temporal relational assumption $\mathcal{R}$ in the residue of a temporal entailment $\rho \wedge \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) \wedge \theta_a \vdash_t \theta_c \blacktriangleright (\theta_r, \mathcal{R})$ can be defined as follows:*

(i) $\mathcal{R} \equiv \mathtt{true}$, *if both θ_a and θ_c are known predicates from* $\{\mathtt{Term}\ [\overline{e}], \mathtt{Loop}, \mathtt{MayLoop}\}$.
(ii) $\mathcal{R}{\equiv}\rho{\wedge}\bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) \Rightarrow \theta_c$, *if θ_c is an unknown post-predicate.*
(iii) $\mathcal{R} \equiv \rho{\wedge}\theta_a \Rightarrow \theta_c$, *otherwise.*

This temporal entailment can be integrated into an entailment system with frame $\Psi \vdash \Phi \leadsto \Psi_r$, to obtain a new entailment procedure of the form $\Psi \vdash \Phi \leadsto (\Psi_r, \mathcal{S})$, that also captures in its residue the set of relational assumptions $\mathcal{S}$ generated by the temporal sub-entailments. The rules to discharge entailments of logic formulas with disjunctions are:

[ENT−DISJ−LHS]

$$\frac{\Psi = \bigvee_i \exists \overline{v_i} \cdot (\rho_i {\wedge} \bigwedge_j \theta^j_i) \qquad \forall i \cdot (\rho_i {\wedge} \bigwedge_j \theta^j_i) \vdash \Phi \leadsto (\Psi^i_r, \mathcal{S}_i)}{\Psi \vdash \Phi \leadsto (\bigvee_i \exists \overline{v_i} \cdot \Psi^i_r,\ \bigcup_i \mathcal{S}_i)}$$

[ENT−CONJ]

$$\frac{\rho_a \vdash \rho_c \leadsto \rho_r \qquad \rho_r \wedge \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) \wedge \theta_a \vdash_t \theta_c \blacktriangleright (\theta_r, \mathcal{R})}{\rho_a {\wedge} \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) {\wedge} \theta_a \vdash \rho_c {\wedge} \theta_c \leadsto (\rho_r {\wedge} \bigwedge_i \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}) {\wedge} \theta_r, \{\mathcal{R}\})}$$

4. Generating Temporal Assumptions

In this section, we show how our new entailment procedure is incorporated into Hoare logic to generate a set of relation assumptions over the unknown temporal constraints.

Language. To formalize this task, we provide a core language (Figure 5) with usual constructs, such as data structure declaration *tdecl*, method declaration *meth*, method call, assignment. This core language does not include the while-loop construct, as it assumes an automatic translation of loops into tail-recursive methods. For `int` type, we assume the use of arbitrary precision integers.

A method declaration consists of a specification with pre- and post-condition and its body. Primitive/library methods do not have a body and must have their specifications (including termination) pre-declared. For termination and non-termination inference, a pair of unknown pre- and post-predicate are automatically added into the specification of each method whose termination status is unknown.

Hoare rules. To support inference, Hoare judgment is formalized as $\vdash \{\Psi_{\text{Pre}}\}\ e\ \{\Phi_{\text{Post}}, \mathcal{S}\}$, where $\mathcal{S}$ is a generated set of temporal assumptions. For illustration, we show the rule for method call:

[TNT−CALL]

$$\frac{\begin{array}{c} t_0\ mn(\overline{t\ v})\ (\Psi_{\text{Pre}}, \Phi_{\text{Post}})\ \{e\} \in \textit{Prog} \\ \Psi \vdash \Psi_{\text{Pre}} \leadsto (\Phi, \mathcal{S}_1) \quad \Psi_r = \Phi * \Phi_{\text{Post}} \quad \mathcal{S}_2 = \mathtt{filter}(\mathcal{S}_1) \end{array}}{\vdash \{\Psi\}\ mn(\overline{v})\ \{\Psi_r, \mathcal{S}_2\}}$$

To facilitate the termination inference, at method calls, we collect only *nontrivial* assumptions of unknown temporal constraints. We list below trivial relational assumptions, which will be removed by the function `filter` as shown in the rule [TNT−CALL].

Firstly, the relational assumption $\rho{\wedge}\theta_a \Rightarrow \theta_c$ is trivial for any θ_a and θ_c if the context ρ is unsatisfiable. Secondly, the assumptions $\rho{\wedge}\mathtt{Loop} \Rightarrow \theta_c$ and $\rho{\wedge}\mathtt{MayLoop} \Rightarrow \theta_c$ are trivially valid for any program state ρ because the constraints `Loop` and `MayLoop` can accept any temporal constraints in the RHS [33]. Finally, $\rho{\wedge}\theta_a \Rightarrow \mathtt{Term\,M}$ is trivial if the callee n, whose termination is denoted by the temporal constraint `Term M`, and the caller m are *not* mutually recursive.

```
1:  procedure solve($\mathcal{M}$)
    // $\mathcal{M} = \{t_i\ mn_i(\overline{t\ v})\ (\mathtt{U}^i_{\mathtt{pr}}, \mathtt{U}^i_{\mathtt{po}})\ \{e\}\ \{\mathcal{S}_i, \mathcal{T}_i\} \mid 1{\leq}i{\leq}n\}$
2:      $\Theta \leftarrow \{\mathtt{U}^i_{\mathtt{pr}} \equiv \mathtt{U}^i_{\mathtt{pr}}, \mathtt{U}^i_{\mathtt{po}} \equiv \mathtt{U}^i_{\mathtt{po}} \mid 1{\leq}i{\leq}n\}$
        // Initial defns for unknown pre/post predicates
3:      for each $mn_i$ in $\mathcal{M}$ do
4:          $\beta \leftarrow$ syn_base($\mathcal{S}_i, \mathcal{T}_i$)
5:          $\Theta \leftarrow$ refine_base($\Theta, \mathtt{U}^i_{\mathtt{pr}}, \mathtt{U}^i_{\mathtt{po}}, \beta$)
6:      $\mathcal{S} \leftarrow \bigcup \mathcal{S}_i$; $\mathcal{T} \leftarrow \bigcup \mathcal{T}_i$; iter $\leftarrow 0$
7:      $\mathcal{S} \leftarrow$ spec_relass($\mathcal{S}, \Theta$); $\mathcal{T} \leftarrow$ spec_relass($\mathcal{T}, \Theta$)
8:      $\mathcal{G} \leftarrow$ reach_graph($\mathcal{S}$)
        // Reachability graph for unknown pre-predicates
9:      for each $scc$ in $\mathcal{G}$ do
10:         $(r, \Theta) \leftarrow$ TNT_analysis($\mathcal{G}, scc, \mathcal{T}, \Theta$)
11:         if $\neg r \wedge$ iter < MAX_ITER then iter++; goto 7
12:         if iter $\geq$ MAX_ITER then break
13:         $\mathcal{T} \leftarrow$ spec_relass($\mathcal{T}, \Theta$)
14:         $\mathcal{G} \leftarrow$ graph_update($\mathcal{G}, \Theta$)
15:     return finalize($\Theta$)
```

Figure 6. Overall Inference Algorithm

Note that assumptions of the form $\rho{\wedge}\mathtt{U}^1_{\mathtt{pr}}(\overline{v_1}) \Rightarrow \mathtt{U}^2_{\mathtt{pr}}(\overline{v_2})$ are not trivial for any caller m and callee n. However, when m and n are in two different *scc* groups, this kind of assumptions can be avoided if we do a bottom-up verification and inference in which the (non-)termination of the callee n is inferred and the unknown $\mathtt{U}^2_{\mathtt{pr}}(\overline{v_2})$ is instantiated before the caller m is processed.

For each method declaration, we collect a set of relational assumptions $\mathcal{S}$ during the verification of its body, and another set of relational assumptions $\mathcal{T}$ at the method's exit points via the entailment for proving the post-condition, as shown below:

[TNT−METH]

$$\frac{\vdash \{\Psi_{\text{Pre}}\}\ e\ \{\Psi, \mathcal{S}\} \qquad \Psi \vdash \Phi_{\text{Post}} \leadsto (\Psi_r, \mathcal{T})}{t_0\ mn(\overline{t\ v})\ (\Psi_{\text{Pre}}, \Phi_{\text{Post}})\ \{e\}\ \{\mathcal{S}, \mathcal{T}\}}$$

The termination and non-termination inference engine is invoked when a whole group of mutually recursive methods are verified and their sets of relational assumptions are collected, as shown in the rule [TNT−INF] below.

[TNT−INF]

$$\frac{M_{scc} = \{t^0_i\ mn_i(\overline{t_i\ v_i})\ (\mathtt{U}^i_{\mathtt{pr}}(\overline{v_i}), \mathtt{U}^i_{\mathtt{po}}(\overline{v_i}))\ \{e\}\ \{\mathcal{S}_i, \mathcal{T}_i\} \mid 1{\leq}i{\leq}n\}}{M_{scc} \leadsto \mathtt{solve}(M_{scc})}$$

The `solve` procedure infers definitions for unknown temporal predicates and will be depicted in detail next.

5. Inferring Termination and Non-Termination

This section is devoted to the `solve` procedure used to infer the definitions for the unknown pre/post-predicates, based on the set of relational assumptions generated by Hoare-style verification. The overall algorithm is shown in Figure 6.

In this algorithm, Θ is used to store the set of definitions inferred thus far for the unknown temporal predicates. Since a key idea of our inference mechanism is *case analysis* that incrementally separates the terminating and non-terminating behaviors of the analyzed methods, the definition for each unknown predicate might be split into multiple scenarios, for which termination is either known or unknown.

DEFINITION 2 (Unknown Temporal Predicates). *During the inference process, the definitions for a pair of unknown pre-predicate*

$\mathtt{U}_{\mathtt{pr}}(\overline{v})$ *and post-predicate* $\mathtt{U}_{\mathtt{po}}(\overline{v})$ *are of the form*

$$\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \bigvee_i (\pi_i \wedge \theta^i_{\mathtt{pr}}) \textit{ and } \mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \bigwedge_i (\pi_i \Rightarrow \theta^i_{\mathtt{po}})$$

where each $\theta^i_{\mathtt{pr}}$ *is either a known or unknown pre-predicate and* $\theta^i_{\mathtt{po}}$ *is either* `true`*,* `false` *or an unknown post-predicate. The set of guards* $\{\pi_1, \ldots, \pi_n\}$ *must be (1)* feasible*, i.e.* $\forall i \cdot \mathtt{SAT}(\pi_i)$*, (2)* exclusive*, i.e.* $\forall i, j \cdot i{\neq}j \Rightarrow \mathtt{UNSAT}(\pi_i{\wedge}\pi_j)$*, and (3)* exhaustive*, i.e.* $\pi_1{\vee}\pi_2{\vee}\ldots{\vee}\pi_n \equiv \mathtt{true}$.

The initial form of each unknown predicate is the predicate itself with guard condition `true`, *e.g.* $\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \mathtt{true}{\wedge}\mathtt{U}_{\mathtt{pr}}(\overline{v})$. At the end of the analysis, all $\theta^i_{\mathtt{pr}}$ and $\theta^i_{\mathtt{po}}$ become known.

The inference deals with two groups of temporal relational assumptions collected by rule $\boxed{\text{TNT-METH}}$, namely

1. Pre-assumptions $\mathcal{S}$ collected when proving preconditions at method calls. They can be used to infer *(i)* ranking functions for termination proving, and *(ii) temporal reachability* graph that guides our search for proving termination vs. non-termination.
2. Post-assumptions $\mathcal{T}$ collected when proving postconditions contain information about unknown post-predicates. They can be used to infer *(i)* termination base cases, *(ii)* inductive unreachability to prove non-termination or *(iii)* new conditions for the case analysis.

The algorithm in Figure 6 first derives the base case of each analyzed method (line 4), and then refines the definitions of unknown temporal predicates in Θ with these newly inferred cases (line 5). After updating the set of relational assumptions (line 7), our algorithm (re-)builds the *temporal reachability* graph $\mathcal{G}$ from the latest $\mathcal{S}$ (line 8).

For each *scc* of the graph $\mathcal{G}$ in the bottom-up topological order, the analysis attempts to prove either termination or non-termination or to infer new cases for case-splitting and then updates the set Θ with the inferred result (line 10). If every unknown temporal predicate corresponding to the current *scc* is resolved into known predicates, the inference continues with the next *scc*s after updating the post-assumptions in $\mathcal{T}$ (line 13) and the graph $\mathcal{G}$ (line 14) with the new inferred known predicates. Otherwise, it restarts the core algorithm (line 11) with the updated Θ, whose elements have been refined into new sub-cases.

The algorithm halts when every unknown predicate has been resolved or the number of iterations reaches the maximum `MAX_ITER` pre-set by users. In the latter case, the remaining unknown predicates in Θ will be marked as `MayLoop` by an auxiliary procedure `finalize`. Next we will explain each inference step in some detail.

5.1 Inferring Base Case Termination

Identifying the conditions for base-case termination is an important first step before any other analyses. Formally:

DEFINITION 3 (Base Case Pre-Condition). *Each base case termination precondition of a method must satisfy the following three conditions:*

(i) Its method's exit is reachable from it.
(ii) No mutually recursive method call is met in executions starting from this pre-condition.
(iii) All other method calls encountered from this pre-condition must have been proven to terminate.

While a syntactic-based approach that identifies base-case termination from its control-flow may be sufficient, we propose a semantics-based approach which infers a method's base case precondition from the two sets of assumptions $\mathcal{S}$ and $\mathcal{T}$ collected from the method, as follows:

$$\frac{\begin{array}{c}\rho = \bigvee\{(\rho_i/\{\overline{v}\}) \mid \rho_i{\wedge}\mathtt{U}_{\mathtt{pr}}(\overline{v}) \Rightarrow \theta^i_c \in \mathcal{S}\} \\ \varrho = \bigvee\{(\beta_j/\{\overline{v}\}) \mid \beta_j{\wedge}\mathtt{true} \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}) \in \mathcal{T}\}\end{array}}{\mathtt{syn\_base}(\mathcal{S}, \mathcal{T}) = \varrho \wedge \neg\rho}$$

where $\rho/\{\overline{v}\} \equiv \exists(\mathtt{FV}(\rho){-}\{\overline{v}\}) \cdot \rho$. Using our running example, we have $\mathcal{S} = \{a^0_2\}$ and $\mathcal{T} = \{a^0_1, a^0_3\}$:

(a^0_1) $\mathtt{x}{<}\mathtt{0}{\wedge}\mathtt{true} \Rightarrow \mathtt{U}_{\mathtt{po}}(\mathtt{x}, \mathtt{y})$

(a^0_2) $\mathtt{x}{\geq}\mathtt{0} \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y}{\wedge}\mathtt{U}_{\mathtt{pr}}(\mathtt{x}, \mathtt{y}) \Rightarrow \mathtt{U}_{\mathtt{pr}}(\mathtt{x}', \mathtt{y}')$

(a^0_3) $\mathtt{x}{\geq}\mathtt{0} \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{true}{\wedge}\mathtt{U}_{\mathtt{po}}(\mathtt{x}', \mathtt{y}') \Rightarrow \mathtt{U}_{\mathtt{po}}(\mathtt{x}, \mathtt{y})$,

Each post-assumption $\beta_j{\wedge}\mathtt{true} \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}) \in \mathcal{T}$, whose antecedent does not contain any unknown post-predicate, capture a potential base-case termination condition. Due to over-approximation, the actual base-case condition (over the method's parameters $\overline{v}$) must be formed by such conditions ($\bigvee \beta_j$), conjoined with the negation of contexts ($\neg\rho$) for the recursive calls. By identifying the base-case condition in $\{a^0_1\}$ and conditions for recursive pre-assumption in $\{a^0_2\}$, we can precisely infer $\mathtt{syn\_base}(\mathcal{S}, \mathcal{T}) = x{<}0{\wedge}\neg(x{\geq}0)$.

With the inferred base case $\beta = \mathtt{syn\_base}(\mathcal{S}, \mathcal{T})$ (line 4), we can now invoke the procedure `refine_base` (line 5) to refine (or specialize) the unknown predicates $\mathtt{U}_{\mathtt{pr}}(\overline{v})$ and $\mathtt{U}_{\mathtt{po}}(\overline{v})$, before updating their definitions in Θ (via the operator $\oplus$) as shown below.

$$\frac{\begin{array}{c}\bigvee \mu_i \equiv \neg\beta \\ \Delta_{\mathtt{pr}} = (\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \bigvee(\mu_i \wedge \mathtt{U}^i_{\mathtt{pr}}(\overline{v})) \vee (\beta \wedge \mathtt{Term})) \\ \Delta_{\mathtt{po}} = (\mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \bigwedge(\mu_i \Rightarrow \mathtt{U}^i_{\mathtt{po}}(\overline{v}))) \\ \Omega = \bigcup\{\mathtt{U}^i_{\mathtt{pr}}(\overline{v}) \equiv \mathtt{U}^i_{\mathtt{pr}}(\overline{v}), \mathtt{U}^i_{\mathtt{po}}(\overline{v}) \equiv \mathtt{U}^i_{\mathtt{po}}(\overline{v})\}\end{array}}{\mathtt{refine\_base}(\Theta, \mathtt{U}_{\mathtt{pr}}(\overline{v}), \mathtt{U}_{\mathtt{po}}(\overline{v}), \beta) = \Theta \oplus (\{\Delta_{\mathtt{pr}}, \Delta_{\mathtt{po}}\} \cup \Omega)}$$

Since the method's termination status in the remaining condition $\mu = \neg\beta$ is unknown. In the new definitions of $\mathtt{U}_{\mathtt{pr}}(\overline{v})$ and $\mathtt{U}_{\mathtt{po}}(\overline{v})$, each pair of fresh predicates $\mathtt{U}^i_{\mathtt{pr}}(\overline{v})$ and $\mathtt{U}^i_{\mathtt{po}}(\overline{v})$ is associated with a disjunct μ_i in the disjunctive normal form of μ. For our running example, this refinement leads to:

$$\begin{array}{llcl}\mathtt{U}_{\mathtt{pr}}(\mathtt{x}, \mathtt{y}) \equiv & \mathtt{x}{<}\mathtt{0} \wedge \mathtt{Term} & \vee & \mathtt{x}{\geq}\mathtt{0} \wedge \mathtt{U}^1_{\mathtt{pr}}(\mathtt{x}, \mathtt{y}) \\ \mathtt{U}_{\mathtt{po}}(\mathtt{x}, \mathtt{y}) \equiv & \mathtt{x}{<}\mathtt{0} \Rightarrow \mathtt{true} & \wedge & \mathtt{x}{\geq}\mathtt{0} \Rightarrow \mathtt{U}^1_{\mathtt{po}}(\mathtt{x}, \mathtt{y})\end{array}$$

After the unknown predicates have been updated with base-case termination conditions, we transform the sets of relation assumptions by using the procedure `spec_relass` (line 7) described next.

5.2 Specializing Relational Assumptions

Whenever some unknown predicates in Θ receive new definitions, our inference algorithm will update its sets of relational assumptions with the procedure `spec_relass`. Its first parameter is a set of relational assumptions. Its second parameter Θ contains the definitions of unknown predicates.

For each relational assumption with unknown predicates, the procedure `spec_relass` finds the current definitions of these unknown predicates in Θ and substitutes them directly into the assumption. As the definition of each unknown predicate consists of exclusive and complete guards, we can further split each substituted assumptions into multiple specialized assumptions. We show below just one example where `spec_relass` is called with a new pre-assumption with two unknown predicates.

$$\frac{\begin{array}{c}\mathtt{U}^1_{\mathtt{pr}}(\overline{v_1}){\equiv}\bigvee^n_{i=1}(\rho_{1i}{\wedge}\theta^{1i}_{\mathtt{pr}}) \in \Theta \qquad \mathtt{U}^2_{\mathtt{pr}}(\overline{v_2}){\equiv}\bigvee^m_{j=1}(\rho_{2j}{\wedge}\theta^{2j}_{\mathtt{pr}}) \in \Theta \\ \mathcal{C} = \{\rho{\wedge}\rho_{1i}{\wedge}\rho_{2j}{\wedge}\theta^{1i}_{\mathtt{pr}} \Rightarrow \theta^{2j}_{\mathtt{pr}} \mid 1{\leq}i{\leq}n, 1{\leq}j{\leq}m\}\end{array}}{\begin{array}{c}\mathtt{spec\_relass}(\{\rho{\wedge}\mathtt{U}^1_{\mathtt{pr}}(\overline{v_1}) \Rightarrow \mathtt{U}^2_{\mathtt{pr}}(\overline{v_2})\} \cup \mathcal{S}, \Theta) = \\ \mathcal{C} \cup \mathtt{spec\_relass}(\mathcal{S}, \Theta)\end{array}}$$

For our running example, the relational assumption (a^0_2) was specialized by its earlier partial definition into two more specialized assumptions: (a^0_{2a}) and (a^0_{2b}).

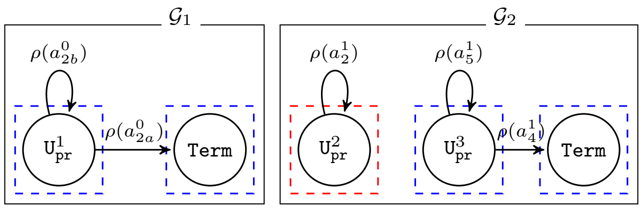
```
16: procedure TNT_analysis($\mathcal{G}, scc, \mathcal{T}, \Theta$)
17:     $r \leftarrow$ true
18:     $\mathcal{O} \leftarrow$ scc_succ($scc, \mathcal{G}$)
19:     if $\mathcal{O} = \{\}$ then
20:         if $scc$ has one node $\mathtt{U_{pr}}$ without cyclic edge then
21:             $\mathtt{U_{po}} \leftarrow$ the post-pred corresponding to $\mathtt{U_{pr}}$
22:             $\Theta \leftarrow \Theta \oplus \{\mathtt{U_{pr}} \equiv \mathtt{Term}, \mathtt{U_{po}} \equiv \mathtt{true}\}$
23:         else $(r, \Theta) \leftarrow$ prove_NonTerm($scc, \mathcal{T}, \Theta$)
24:     else if $\forall \theta \in \mathcal{O} \cdot \theta \equiv \mathtt{Term}\ [\overline{e}]$ then
25:         $(r, \Theta) \leftarrow$ prove_Term($\mathcal{G}, scc, \Theta$)
26:         if $\neg r$ then $(r, \Theta) \leftarrow$ prove_NonTerm($scc, \mathcal{T}, \Theta$)
27:     else $(r, \Theta) \leftarrow$ prove_NonTerm($scc, \mathcal{T}, \Theta$)
28:     return $(r, \Theta)$
```

Figure 7. Core TNT Inference Algorithm

5.3 Resolving Temporal Reachability Graph

The core of our inference algorithm (in Figure 6) incrementally resolves the unknown predicates present in the (specialized) relational pre-assumptions. If its attempt fails, it would also derive conditions for the next case analysis. This core algorithm uses a *reachability graph* $\mathcal{G}$, constructed from pre-predicates in $\mathcal{S}$, to guide its proof search. Formally:

DEFINITION 4 (Temporal Reachability Graph). *Given a set of pre-assumptions $\mathcal{S}$, a* temporal reachability graph *$\mathcal{G} = (V, E)$ is constructed from a set of vertices V and a set of labeled edges E, as follows. For each pre-assumption $\rho \wedge \theta_a \Rightarrow \theta_c \in \mathcal{S}$, we add two vertices θ_a and θ_c into V and an edge $(\theta_a, \rho, \theta_c)$ from θ_a to θ_c labeled by ρ into E.*



For example, the two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are built for the inference of the running example. $\mathcal{G}_1$ is constructed from pre-assumptions (a^0_{2a}) and (a^0_{2b}) obtained after base case inference. The edges of $\mathcal{G}_1$ are labeled by $\rho(a^0_{2a})$ and $\rho(a^0_{2b})$, the contexts in (a^0_{2a}) and (a^0_{2b}) resp., *e.g.* $\rho(a^0_{2b}) \equiv (\mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{\geq}0)$. The self-loop edge on node $\mathtt{U^1_{pr}}$ denotes the case when the latest values of program variables (*i.e.* $\mathtt{x}'{\geq}0$), are still in the same loop condition as their initial values ($\mathtt{x}{\geq}0$). The edge from $\mathtt{U^1_{pr}}$ to Term indicates the base case is reached when $\mathtt{x}'{<}0$. Similarly, the graph $\mathcal{G}_2$ is constructed from pre-assumptions (a^1_4), (a^1_5) and (a^1_2) after a new case split $\mathtt{y}{\geq}0$ and $\mathtt{y}{<}0$ has been inferred.

Our core algorithm firstly partitions $\mathcal{G}$ into strongly connected components (scc), (*e.g.* dashed boxes in $\mathcal{G}_1$ and $\mathcal{G}_2$), whereby each unknown temporal predicate denotes an unknown behavior. Moreover, this unknown predicate is mutually dependent on the other predicates in the same scc. Using a bottom-up approach, the inference mechanism processes each scc in a topologically sorted order. With this approach, termination and non-termination proofs for *phase-change* programs [14] and that for mutual recursion are easily supported.

DEFINITION 5 (**scc**'s successors). *Given a graph $\mathcal{G}$, the outside successors of a scc in $\mathcal{G}$ is the set of all successors of any vertex in this scc but excluding the scc itself,*

$$\mathtt{scc\_succ}(scc, \mathcal{G}) = \bigcup\{\mathtt{succ}(v, \mathcal{G}) \mid v \in scc\} \setminus scc$$

where $\mathtt{succ}(v, \mathcal{G})$ *returns all successors of the vertex* v.

```
29: procedure prove_Term($\mathcal{G}, scc, \Theta$)
30:     $\mathcal{C} \leftarrow \{\mathtt{gen}(e) \mid e{\equiv}(\mathtt{U^i_{pr}}, \rho, \mathtt{U^j_{pr}}){\in}\mathcal{G}(E) \wedge \mathtt{U^i_{pr}}, \mathtt{U^j_{pr}}{\in}scc\}$
31:     $\Gamma \leftarrow$ syn_rank($\mathcal{C}$)
32:     if $\Gamma \neq \{\}$ then
33:         $\mathcal{D} \leftarrow$ subst_rank($scc, \Gamma$)
34:         return $(\mathtt{true}, \Theta \oplus \mathcal{D})$
35:     else return $(\mathtt{false}, \Theta)$
```

Figure 8. Procedure for Proving Termination over a scc

Our core algorithm, named TNT_analysis, for manipulating each scc is outlined in Figure 7. After this analysis, if all vertices in the scc are resolved as known temporal predicates, our procedure returns the result $r{=}\mathtt{true}$. Otherwise, it returns $r{=}\mathtt{false}$ to allow inference mechanism to restart for the next iteration (from line 7 in Figure 6). Moreover, upon termination of this procedure, some unknown pre- and post-predicates in store Θ, are updated with their new definitions.

Our procedure (Figure 7) uses the set $\mathcal{O}$ of the scc's successors to determine whether termination proof (by sub-procedure prove_Term), or non-termination proof (by prove_NonTerm), should be applied to resolve the unknown temporal predicates. Specifically, when the scc has only one unknown node $\mathtt{U_{pr}}$ without any cyclic edge and successor (line 20), we resolve the unknown pre-predicate $\mathtt{U_{pr}} \equiv \mathtt{Term}$ and its corresponding post-predicate $\mathtt{U_{po}} \equiv \mathtt{true}$ for trivial termination (line 22). Moreover, when the set $\mathcal{O}$ is nonempty, the procedure invokes prove_Term with ranking function synthesis only if every element of $\mathcal{O}$ is a known $\mathtt{Term}\ [\overline{e}]$ predicate (line 24-25).

For the running example, the procedure applies termination proofs for the left scc in $\mathcal{G}_1$ and the middle scc in $\mathcal{G}_2$. For the left scc in $\mathcal{G}_2$, it applies a non-termination proof directly. In the next sub-sections, we present the sub-procedures for proving termination and non-termination over a scc.

5.4 Inferring Ranking Function

For proving termination on a scc, we implement the procedure prove_Term (sketched in Figure 8) to find a linear ranking function for each unknown pre-predicate in this scc by using a constraint-based technique [22, 23, 38, 42] with Farkas' lemma [43].

Initially, we create a unique ranking function template for each unknown pre-predicate $\mathtt{U_{pr}}(v_1, .., v_n) \in scc$ by the procedure gen_rank, defined as

$$\mathtt{gen\_rank}(\mathtt{U_{pr}}(v_1, \ldots, v_n)) = c_0 + \textstyle\sum_{i=1}^{n} c_i v_i$$

where $c_0, c_1, \ldots, c_n$ are unknown coefficients of the ranking function. Next, we generate a set of constraints over these ranking functions from every edge in $\mathcal{G}$ that connects two nodes in the scc (line 30). That is, given an edge $e \equiv (\mathtt{U^i_{pr}}(\overline{v_i}), \rho, \mathtt{U^j_{pr}}(\overline{v_j})) \in \mathcal{G}(E)$ s.t. $\mathtt{U^i_{pr}}(\overline{v_i}), \mathtt{U^j_{pr}}(\overline{v_j}) \in scc$, the constraint generated from it is

$$\frac{r_i(\overline{v_i}){=}\mathtt{gen\_rank}(\mathtt{U^i_{pr}}(\overline{v_i})) \quad r_j(\overline{v_j}){=}\mathtt{gen\_rank}(\mathtt{U^j_{pr}}(\overline{v_j}))}{\mathtt{gen}(e) = \forall \overline{v_i}, \overline{v_j} \cdot \rho \Rightarrow (r_i(\overline{v_i}) > r_j(\overline{v_j}) \wedge r_i(\overline{v_i}) \geq 0)}$$

This constraint indicates that the ranking function $r_i(\overline{v_i})$ is bounded and decreasing across a (mutually) recursive method call under the call context ρ. For example, the constraint generated from the middle scc in $\mathcal{G}_2$ is

$$\begin{array}{c}\forall \mathtt{x}, \mathtt{y} \cdot \mathtt{x}{\geq}0 \wedge \mathtt{x}'{=}\mathtt{x}{+}\mathtt{y} \wedge \mathtt{y}'{=}\mathtt{y} \wedge \mathtt{x}'{\geq}0 \wedge \mathtt{y}{<}0 \Rightarrow \\ r(\mathtt{x}, \mathtt{y}){>}r(\mathtt{x}', \mathtt{y}') \wedge r(\mathtt{x}, \mathtt{y}){\geq}0\end{array}$$

which is then solved by syn_rank to obtain the ranking function $r(\mathtt{x}, \mathtt{y}) = \mathtt{x}$. The method syn_rank (line 31) solves the generated constraints by applying Farkas' lemma on them to obtain another set of constraints over their unknown coefficients, which can be solved by a nonlinear solver, such as [28], to get the actual values of these

```
36: procedure prove_NonTerm($scc, \mathcal{T}, \Theta$)
37:     for each $\mathtt{U}^i_{\mathtt{pr}} \in scc$ do
38:         $\mathcal{T}_i \leftarrow \mathtt{filter\_rel}(\mathcal{T}, \mathtt{U}^i_{\mathtt{pr}})$
39:         $\mathcal{C}_i \leftarrow \bigcup\{\mathtt{abd\_inf}(t) \mid t \in \mathcal{T}_i\}$
40:     $r \leftarrow \bigwedge_i(\mathcal{C}_i \neq \{\} \wedge \forall c \in \mathcal{C}_i \cdot (c \equiv \mathtt{true}))$
41:     if $r$ then $\mathcal{D} \leftarrow \{\mathtt{U}^i_{\mathtt{pr}} \equiv \mathtt{Loop}, \mathtt{U}^i_{\mathtt{po}} \equiv \mathtt{false} \mid \mathtt{U}^i_{\mathtt{pr}} \in scc\}$
42:     else $\mathcal{D} \leftarrow \bigcup_i \mathtt{subst\_unk}(\mathcal{C}_i, \mathtt{U}^i_{\mathtt{pr}}, \mathtt{U}^i_{\mathtt{po}})$
43:     return $(r, \Theta \oplus \mathcal{D})$
```

Figure 9. Proc. for Proving Non-Termination over a *scc*

unknowns. The result is a substitution Γ which maps each unknown coefficient to its actual value.

If the ranking function synthesis succeeds, we update each unknown pre-predicate in this *scc* into `Term` with an actual ranking function (line 34 in Figure 8). Otherwise, we prove non-termination on this *scc* (line 26 in Figure 7). The ranking function for a pre-predicate can be obtained by applying the substitution Γ to its ranking function template, as shown below. Note that $\mathtt{subst\_rank}(\{\}, \Gamma) = \{\}$.

$$\frac{\mathtt{r} = \Gamma(\mathtt{gen\_rank}(\mathtt{U}_{\mathtt{pr}}(\overline{v}))) \quad \mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \mathtt{Term}\,[\mathtt{r}] \quad \mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \mathtt{true}}{\begin{array}{l}\mathtt{subst\_rank}(\{\mathtt{U}_{\mathtt{pr}}(\overline{v})\} \cup \mathcal{U}, \Gamma) = \\ \qquad \{\mathtt{U}_{\mathtt{pr}}(\overline{v}), \mathtt{U}_{\mathtt{po}}(\overline{v})\} \cup \mathtt{subst\_rank}(\mathcal{U}, \Gamma)\end{array}}$$

We also support the synthesis of lexicographic ranking functions, details are omitted for simplicity of presentation.

5.5 Inferring Inductive Unreachability

Procedure $\mathtt{prove\_NonTerm}(scc, \mathcal{T}, \Theta)$ finds non-termination on a *scc* by unreachability of its post-predicates in $\mathcal{T}$. For each $\mathtt{U}_{\mathtt{pr}}(\overline{v}) \in scc$, the method $\mathtt{filter\_rel}(\mathcal{T}, \mathtt{U}_{\mathtt{pr}})$ selects a set of post-assumptions $\mathcal{T}_s \subseteq \mathcal{T}$ such that their RHS post-predicate is the corresponding $\mathtt{U}_{\mathtt{po}}(\overline{v})$. The general form of such post-assumptions is either:

1. $\rho \wedge \mathtt{true} \Rightarrow (\mu \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}))$, or
2. $\rho \wedge \bigwedge(\eta_i \Rightarrow \mathtt{false}) \wedge \bigwedge(\mu_j \Rightarrow \mathtt{U}^j_{\mathtt{po}}(\overline{v_j})) \Rightarrow (\mu \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}))$.

These post-assumptions capture possible non-termination of its method due to predicate $\mathtt{U}_{\mathtt{po}}(\overline{v})$ being unknown, under the condition $\rho \wedge \mu$ where the context ρ is satisfiable. The first post-assumption describes a base-case scenario. In order to ensure unreachability of its post-predicate, we must check that $\rho \wedge \mu$ is unsatisfiable. The second post-assumption shows that we can meet a non-terminating method call (with the postcondition `false`) if the condition η_i is satisfied by $\rho \wedge \mu$. In addition, we can meet a (mutually) recursive call whose termination is unknown if μ_j is satisfied, and thus the respective pre-predicate of $\mathtt{U}^j_{\mathtt{po}}(\overline{v_j})$ also belongs to the analyzed *scc*. We call the conditions η_i, μ_j and μ *potential non-termination conditions* as they could lead to an actual non-termination.

By induction, we prove that a caller is definitely non-terminating under a condition μ, assuming that one of its callee is definitely non-terminating under the same condition. Given a set of post-assumptions $\mathcal{T}_s$, we prove that if each unknown post-predicate in their LHS is `false` then every unknown post-predicate in their RHS is also `false`. This is done by the procedure `abd_inf` (line 39).

- For $t \equiv \rho \wedge \mathtt{true} \Rightarrow (\mu \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}))$, $\mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \mathtt{false}$ if and only if $\rho \wedge \mu$ is unsatisfiable. So the proof succeeds and $\mathtt{abd\_inf}(t)$ returns $\{\mathtt{true}\}$ if $\vdash \rho \wedge \mu \Rightarrow \mathtt{false}$.
- For $t \equiv \rho \wedge \bigwedge(\eta_i \Rightarrow \mathtt{false}) \wedge \bigwedge(\mu_j \Rightarrow \mathtt{U}^j_{\mathtt{po}}(\overline{v_j})) \Rightarrow (\mu \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}))$, given that $\forall j \cdot \mathtt{U}^j_{\mathtt{po}}(\overline{v_j}) \equiv \mathtt{false}$, we have $\mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \mathtt{false}$ if and only if $\rho \wedge \mu \Rightarrow \bigvee \eta_i \vee \bigvee \mu_j$, this means that under the precondition μ, at least one of the callees' non-termination conditions is satisfied, so that the caller is also non-terminating. The proof succeeds and $\mathtt{abd\_inf}(t)$ returns $\{\mathtt{true}\}$ if $\vdash \rho \wedge \mu \Rightarrow \bigvee \eta_i \vee \bigvee \mu_j$.

If the proof succeeds for all pre-predicates in *scc* (signified by r in line 40), we mark the unknown termination status as definitely non-terminating. This procedure thus refines, where possible, each unknown pre-predicate as $\mathtt{U}_{\mathtt{pr}}(\overline{\mathtt{v}}) \equiv \mathtt{Loop}$ and its post-predicate as $\mathtt{U}_{\mathtt{po}}(\overline{\mathtt{v}}) \equiv \mathtt{false}$ (line 41) and updates Θ before returning $(\mathtt{true}, \Theta)$.

For our running example, (a^0_1) and (a^0_3) from $\mathcal{T}$ would cause $\mathtt{prove\_NonTerm}(scc, \mathcal{T}, \Theta)$ to return false, but provide an abductive condition $\mathtt{y} \geq \mathtt{0}$ that facilitates case-splitting (see next subsection). In contrast, (a^1_3) would be used to show that $\mathtt{U}^2_{\mathtt{po}}(\mathtt{x}, \mathtt{y})$ is inductively `false` (or unreachable).

5.6 Case-Splitting

If non-termination proving fails, the method `abd_inf` abductively infers new sub-conditions from the failed proof to refine the potential non-termination condition by case-split.

In the case $t \equiv \rho \wedge \mathtt{true} \Rightarrow (\mu \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}))$, if the proof fails, *i.e.* $\nvdash \rho \wedge \mu \Rightarrow \mathtt{false}$, $\mathtt{abd\_inf}(t)$ simply returns $\{\}$ as any condition that makes the entailment to hold would contradict with the antecedent $\rho \wedge \mu$.

If $t \equiv \rho \wedge \bigwedge(\eta_i \Rightarrow \mathtt{false}) \wedge \bigwedge(\mu_j \Rightarrow \mathtt{U}^j_{\mathtt{po}}(\overline{v_j})) \Rightarrow (\mu \Rightarrow \mathtt{U}_{\mathtt{po}}(\overline{v}))$, and the proof fails, *i.e.* $\nvdash \rho \wedge \mu \Rightarrow \bigvee \eta_i \vee \bigvee \mu_j$, $\mathtt{abd\_inf}(t)$ returns a set of conditions $\mathcal{C}_t$ such that: for each $\beta_k \in \{\eta_i\} \cup \{\mu_j\}$ s.t. $\rho \wedge \mu \wedge \beta_k$ is satisfiable, there exists $\alpha_k \in \mathcal{C}_t$ such that *(i)* $\rho \wedge \mu \wedge \alpha_k$ is satisfiable and *(ii)* $\vdash \rho \wedge \mu \wedge \alpha_k \Rightarrow \beta_k$. That is, if the potential non-termination condition μ of the caller is strengthened by α_k then the (potential) non-termination condition β_k of a callee is satisfied.

For each condition β_k, the solution $\alpha_k \equiv \beta_k$ is a trivial but the weakest solution for α_k. For a more effective case-split, we aim to derive a stronger abductive condition α_k. By the same constraint-based approach used for the ranking function synthesis, we assume the template $\alpha_k \equiv c_0 + \sum^n_{i=1} c_i v_i \geq 0$, where $v_1, .., v_n \equiv \overline{v}$ and $c_0, .., c_n$ are unknown coefficients. We might solve these unknown coefficients with additional optimal constraints, *e.g.* the number of zero-coefficients is maximum, so that we can obtain a better solution with minimum number of program variables.

Given a set of collective abductive conditions $\mathcal{C}$, the procedure `subst_unk` (line 42) refines the pair of $(\mathtt{U}_{\mathtt{pr}}(\overline{v}), \mathtt{U}_{\mathtt{po}}(\overline{v}))$ with these new sub-cases for the update of Θ.

$$\frac{\begin{array}{c}\mathtt{split}(\mathcal{C}) = \{\mu_j\}^m_{j=1} \quad \mu_{m+1} = \neg\mu_1 \wedge \ldots \wedge \neg\mu_m \\ \Delta_1 = (\mathtt{U}_{\mathtt{po}}(\overline{v}) \equiv \bigwedge(\mu_j \Rightarrow \mathtt{U}^j_{\mathtt{po}}(\overline{v}))) \quad \Delta_2 = (\mathtt{U}_{\mathtt{pr}}(\overline{v}) \equiv \bigvee(\mu_j \wedge \mathtt{U}^j_{\mathtt{pr}}(\overline{v}))) \\ \Omega = \bigcup\{\mathtt{U}^j_{\mathtt{pr}}(\overline{v}) \equiv \mathtt{U}^j_{\mathtt{pr}}(\overline{v}), \mathtt{U}^j_{\mathtt{po}}(\overline{v}) \equiv \mathtt{U}^j_{\mathtt{po}}(\overline{v})\}\end{array}}{\mathtt{subst\_unk}(\mathcal{C}, \mathtt{U}_{\mathtt{pr}}(\overline{v}), \mathtt{U}_{\mathtt{po}}(\overline{v})) = \{\Delta_1, \Delta_2\} \cup \Omega}$$

As the conditions in $\mathcal{C}$ might be overlapping, we use the function `split` defined below to partition these conditions into the new set of mutually exclusive conditions $\{\mu_j\}^m_{j=1}$ such that $\bigvee \mathcal{C} \equiv \bigvee\{\mu_j\}$. We also add into the new set the condition $\mu_{m+1} = \neg\mu_1 \wedge \ldots \wedge \neg\mu_m$, if it is satisfiable, to cover the missing case, so that $\{\mu_j\}^{m+1}_{j=1}$ is complete. Note $\mathtt{split}(\{\}) = \{\}$.

$$\frac{\begin{array}{c}\mathcal{C}_2 = \mathtt{split}(\mathcal{C}_1) \quad \mathcal{C}_3 = \{c_i \mid c_i \in \mathcal{C}_2 \wedge \mathtt{UNSAT}(c_i \wedge c_1)\} \\ \mathcal{C}_4 = \{c_i \mid c_i \in \mathcal{C}_2 \wedge \mathtt{SAT}(c_i \wedge c_1)\} \quad c = c_1 \wedge \bigwedge\{\neg c_i \mid c_i \in \mathcal{C}_4\} \\ \mathcal{C}_5 = \{c_i \wedge c_1 \mid c_i \in \mathcal{C}_4\} \cup \{c_i \wedge \neg c_1 \mid c_i \in \mathcal{C}_4 \wedge \mathtt{SAT}(c_i \wedge \neg c_1)\}\end{array}}{\mathtt{split}(\{c_1\} \cup \mathcal{C}_1) = \text{if } \mathtt{SAT}(c) \text{ then } \{c\} \cup \mathcal{C}_3 \cup \mathcal{C}_5 \text{ else } \mathcal{C}_3 \cup \mathcal{C}_5}$$

6. Experiments

We have implemented the proposed inference mechanism on top of HipTnt [33], an existing verification system that can verify both termination and non-termination specifications given by users. The new inference system HipTNT+ and this paper's artifact are available for both online use and download at

http://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/plus/.

Benchmark	crafted					crafted-lit					numeric					memory-alloca					Total				
	Y	N	U	T/O	Time	Y	N	U	T/O	Time	Y	N	U	T/O	Time	Y	N	U	T/O	Time	Y	N	U	T/O	Time
AProVE	20	0	15	4	373.1	106	0	23	21	2122.1	68	0	0	0	859.0	69	0	7	5	2475.9	263	0	45	30	5830.0
ULTIMATE	21	15	1	2	331.0	113	17	14	6	1704.1	56	0	9	3	508.8	37	6	8	30	2659.0	227	38	32	41	5202.8
HIPTNT+	19	13	7	0	35.1	115	20	15	0	114.5	66	0	2	0	35.3	67	6	8	0	123.6	267	39	32	0	308.4

Figure 10. Termination Outcomes on SV-COMP'15's Benchmarks

Total	Y	N	U	T/O	Time
T2 (rev. 051de80)	161	30	28	2	729.0
HIPTNT+	170	30	21	0	204.3

Figure 11. Comparison on Loop-based Integer Programs

To evaluate our approach, we compare our tool against state-of-the-art systems, *i.e.* T2 [7], ULTIMATE [26] and AProVE [21]. The last tool is the recent winner for several categories of problems in the annual Termination Competition 2014 (TermCOMP'14) [46] and the Termination category in the Competition on Software Verification 2015 (SV-COMP'15) [45]. We made our preliminary comparison based on a set of numerical and pointer-based C programs selected from four benchmarks used for the Termination category of the SV-COMP'15. These benchmarks were largely contributed by the teams of AProVE and ULTIMATE. We have excluded 61 programs with arrays and strings from the total 399 programs in 4 benchmarks, since these two aspects[1] have not yet been handled by our specification inference and verification system. In addition, we made a comparison with T2 for only 221 loop-based integer programs from the first 3 benchmarks because the tool llvm2KITTeL [18] (which translates C programs into T2's format) cannot properly handle pointers and recursive methods. For AProVE and ULTIMATE, we used their SV-COMP'15 versions, which are described in [44] and [27], respectively. The experiments were performed on an Ubuntu 12.04 machine with the AMD Opteron 6172 (2.1GHz) processor and 64GB of RAM.

In Figure 10 and 11, we report the number of programs whose `main` methods' termination or non-termination were proven successfully in columns labeled by Y (for termination) or N (for non-termination), respectively. The columns U (*i.e.* unknown) show the number of programs in which the tools cannot decide whether they are definitely terminating or non-terminating. The number of unsuccessful cases in which the tools give no answer after a timeout is provided in the columns T/O. As in the TermCOMP'14 competition, we set a wall-clock timeout of 300 seconds for the proving process on each program. Finally, the last column in each benchmark presents the total time (excluding timeouts) each tool took to prove the termination and non-termination of the whole benchmark. In this evaluation, we only report the wall-clock time instead of the consumed CPU time of all the verifier's processes because CPU time of tools executing jobs in parallel, such as AProVE, would be much higher otherwise. Our tool's timing covers memory safety analysis and post-condition inference, where required.

The overall result shows that our HIPTNT+ can efficiently (without any timeout) infer more (non-)termination properties than the other tools. Note that all answers (specifications inferred) that were returned by our tool have been successfully re-verified[2] by an underlying automated verification system. Thus, our tool does not have any false positive nor negative for this set of benchmarks. For the other analyzers, we check their answers and found that T2 initially had five unsound outcomes which were subsequently fixed[3].

[1] These are orthogonal to termination and non-termination reasoning.

[2] The re-verification is optional but useful for testing whether an un-certified program analyzer is returning sound results or not.

[3] These problems have been fixed at commit 051de80 at https://github.com/mmjb/T2 after we reported them to the developers of T2.

7. Related Work

Over the last decade, there has been a large body of work on proving program termination. Most of these termination provers, such as TERMINATOR [12] and its successor T2 [7, 16], ARMC [39], TAN [29] and ULTIMATE [26], either show that a program terminates for all (given) inputs or return a counterexample to termination upon the failure of termination proofs. However, due to the incompleteness of termination-based techniques, these provers cannot guarantee that every returned counterexample (from failed termination proofs) leads to a definitely non-terminating execution. Thus, each tool might deploy a separate non-termination proving technique to prove that the counterexample is feasible. Also, each such counterexample is only an under-approximation of its program execution, so that it does not capture the wider scenarios for non-terminating behaviors of the analyzed program.

We have also seen much related work on proving program non-termination, *e.g.* [1, 6, 8, 24, 31, 35, 47]. Non-termination provers, such as TNT [24] and INVEL [47], attempt to disprove program termination by searching for some initial configurations that act as witnesses for non-termination. To find a wider class of non-termination bugs, these approaches attempt to discover sufficient pre-conditions for non-termination. Nevertheless, since non-termination proving techniques are also incomplete, the analyzed program is not guaranteed to terminate under the complement of the inferred pre-condition for non-termination.

The dual problem of conditional termination, first addressed in [14], identifies initial configurations that ensure termination. In [14], such termination preconditions are derived from potential ranking functions, which are bounded but not decreasing. Later, the tools of FLATA [4] and ACABAR [19] infer the sufficient precondition for termination from (the negation on over-approximation of) the set of initial states from which the program might not terminate. However, FLATA differs from ACABAR by limiting itself to classes of loops with restricted forms in which the precise non-termination conditions are definable.

8. Conclusion

We have proposed a modular inference framework for program termination and non-termination. By incorporating unknown pre/post-predicates into specification logic for termination reasoning, our proposed framework employs a Hoare-style forward verification to collect a set of relational assumptions to help soundly discover termination and non-termination properties. Our technique analyzes program termination and non-termination at the same time, and constructs a summary of these behaviors for each method. This enables better modularity and reuse for our proving processes. Furthermore, it is integrated with a verification system allowing us to use partial specification and to re-check our inference outcome. As seen in the experiments over SV-COMP'15 benchmarks, our tool compares favorably against current state-of-the-art termination analyzers.

We shall now discuss two current limitations of our tool. Firstly, our tool handles non-deterministic inputs by identifying conditional statements that directly depend on non-deterministic values. Each such non-deterministic conditional is marked as non-terminating if either of its two branches is non-terminating. This works for most examples, but is less general than the proposal in [8]. Secondly, while we support lexicographic linear ranking functions (LLRF) in our termination reasoning, we cannot handle programs that critically depend on Ramsey's theorem [10, 37] or those that are based on size-change principles [34] but do not have LLRF counterpart. It would be interesting to explore future extensions to our specification logic to better support non-determinism and these other kinds of termination proofs.

Acknowledgements: We thank the reviewers for their insightful suggestions. We are grateful to Long H. Pham, Quang-Trung Ta, Quang Loc Le, Andreea Costea and Zhuohong Cai for their help in preparing HIPTNT+ for the SV-COMP 2015 competition. This work is supported by the MoE Tier-2 grant MOE2013-T2-2-146.

References

[1] M. F. Atig, A. Bouajjani, M. Emmi, and A. Lal. Detecting Fair Non-termination in Multithreaded Programs. In *CAV*, 2012.

[2] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *CAV*, 2006.

[3] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. In *POPL*, 2007.

[4] M. Bozga, R. Iosif, and F. Konečný. Deciding Conditional Termination. In *TACAS*, 2012.

[5] A. R. Bradley, Z. Manna, and H. B. Sipma. The Polyranking Principle. In *ICALP*, 2005.

[6] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode. In *FoVeOOS*, 2011.

[7] M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *CAV*, 2013.

[8] H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn. Proving nontermination via safety. In *TACAS*, 2014.

[9] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9), 2012.

[10] M. Codish, S. Genaim, M. Bruynooghe, J. Gallagher, and W. Vanhoof. One loop at a time. In *Intl. Workshop on Termination (WST)*, 2003.

[11] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS*, 2005.

[12] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.

[13] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI*, 2007.

[14] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV*, 2008.

[15] B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking Function Synthesis for Bit-Vector Relations. In *TACAS*, 2010.

[16] B. Cook, A. See, and F. Zuleger. Ramsey vs. Lexicographic Termination Proving. In *TACAS*, 2013.

[17] P. Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI*, 2005.

[18] S. Falke, D. Kapur, and C. Sinz. Termination Analysis of C Programs Using Compiler Intermediate Languages. In *RTA*, 2011.

[19] P. Ganty and S. Genaim. Proving termination starting from the end. In *CAV*, 2013.

[20] C. Gherghina, C. David, S. Qin, and W.-N. Chin. Structured specifications for better verification of heap-manipulating programs. In *FM*, 2011.

[21] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *IJCAR*, 2014.

[22] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[23] S. Gulwani, S. Srivastava, and R. Venkatesan. Program Analysis As Constraint Solving. In *PLDI*, 2008.

[24] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving non-termination. In *POPL*, 2008.

[25] W. R. Harris, A. Lal, A. V. Nori, and S. K. Rajamani. Alternation for Termination. In *SAS*, 2010.

[26] M. Heizmann, J. Hoenicke, and A. Podelski. Termination Analysis by Learning Terminating Programs. In *CAV*, 2014.

[27] M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with Array Interpolation (Competition Contribution). In *TACAS*, 2015.

[28] D. Jovanovic and L. M. de Moura. Solving non-linear arithmetic. In *IJCAR*, 2012.

[29] D. Kroening, N. Sharygina, A. Tsitovich, and C. M. Wintersteiger. Termination Analysis with Compositional Transition Invariants. In *CAV*, 2010.

[30] D. Larraz, A. Oliveras, E. Rodriguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD*, 2013.

[31] D. Larraz, K. Nimkar, A. Oliveras, E. Rodriguez-Carbonell, and A. Rubio. Proving Non-termination Using Max-SMT. In *CAV*, 2014.

[32] Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape Analysis via Second-Order Bi-Abduction. In *CAV*, 2014.

[33] T. C. Le, C. Gherghina, A. Hobor, and W.-N. Chin. A resource-based logic for termination and non-termination proofs. In *ICFEM*, 2014.

[34] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.

[35] É. Payet and F. Spoto. Experiments with Non-Termination Analysis for Java Bytecode. *Electr. Notes Theor. Comput. Sci.*, 253(5), 2009.

[36] C. S. Peirce. *Collected papers of Charles Sanders Peirce*. Harvard University Press., 1958.

[37] A. Podelski and A. Rybalchenko. Transition Invariants. In *LICS*, 2004.

[38] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI*, 2004.

[39] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL*, 2007.

[40] C. Popeea and W.-N. Chin. Inferring Disjunctive Postconditions. In *ASIAN*, 2006.

[41] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.

[42] A. Rybalchenko. Constraint Solving for Program Verification: Theory and Practice by Example. In *CAV*, 2010.

[43] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[44] T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In *TACAS*, 2015.

[45] SV-COMP. The Competition on Software Verification. http://sv-comp.sosy-lab.org/2015/, 2015.

[46] TermCOMP. The Annual International Termination Competition. http://termination-portal.org/wiki/Termination_Competition_2014/, 2014.

[47] H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *TAP*, 2008.