

A Specification Logic for Termination Reasoning

Ton-Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin

Department of Computer Science, National University of Singapore

Abstract. We propose a logical framework for specifying and proving assertions about program termination. Although termination of programs has been well studied, it is usually added as an external component to the specification logic. Here we propose to integrate termination requirements directly into our specification logic, as *temporal constraints* for each phase of every method. Our temporal constraints can specify a strict decrease in a bounded measure for termination proofs and the unreachability of method exit for non-termination proofs. Furthermore, our termination-infused logic can leverage richer specification logics to help conduct more complex termination reasoning for programs with structural specification, heap manipulation, exception handling, and multiple phases. We expect our termination reasoning to directly benefit from any future improvements to our specification mechanism because it is fully integrated into our specification logic. Through an experimental evaluation, we report on the usability and practicality of a verification system, based on separation logic, that has been enhanced with our termination constraints.

1 Introduction

Termination proving is an important part of correctness proofs for software systems. Hoare logic has long distinguished two notions of verification proofs, called *partial* and *total correctness*, with the latter requiring a termination obligation. Given a prestate P and a poststate Q , partial correctness is denoted by Hoare triple $\{P\}c\{Q\}$ which makes no assumption on the termination of the code fragment c . The poststate Q is only expected to hold if code c indeed terminates.

Total correctness, denoted by a different Hoare triple $[P]c[Q]$, requires that code fragment c be shown to terminate in addition to meeting the poststate Q after execution. Though termination of code is expected for total correctness, no formal rules (in terms of Hoare triples) were given to help verify termination. Instead, such termination proof obligations are typically conducted using a different termination proving system.

Most state-of-the-art verification systems, such as Dafny [26], Frama-C [12], and KeY [1] also support termination proving in addition to correctness proofs for programs. As an example, consider a method to compute factorial and its simple specification.

```
int fact(int n)
  requires n ≥ 0
  variance n
  ensures res ≥ n ∧ res ≥ 1
  { if (n == 0) return 1;
    else return n * fact(n - 1); }
```

Fig. 1. Code and Spec for fact.

Apart from the precondition $n \geq 0$ and postcondition $res \geq n \wedge res \geq 1$ where res denotes the result of the method, the specification also captures a *variance annotation* n that denotes a well-founded (decreasing) measure for the method to support its termination proof. While

the variance annotation is given in the specification, it is not part of the logical formulae used for writing pre/post specifications. As a result, termination proofs are often conducted separately from correctness proofs. Thus, current approaches have a disadvantage that two proof systems are required, one for correctness and another for termination.

We propose to integrate termination proving requirements directly into our specification logic by introducing three temporal constraints that specify if a computation must terminate, may terminate, or must diverge. This approach allows our existing separation logic infrastructure to be directly utilized for proving the termination of heap-based programs. For example, we have recently developed structured specifications [19], which can be used to construct scenarios for termination reasoning. We also can verify programs that use exceptions [18] and non-linear arithmetic.

Our paper is organized as follows. Sec 2 highlights the motivations behind our proposal with the help of three small but practical examples. Sec 3 presents a calculus to support the reasoning of our temporal termination constraints, and show how its checks can be realized inside an entailment procedure. Sec 4 shows how we can embed termination constraints into our specification logic and then utilize them for termination (and non-termination) proofs through Hoare-style code verification rules. Sec 5 highlights how phase numbering can be automatically inferred for the terminating constraints of given methods. Sec 6 presents a soundness proof for our termination calculus. Sec 7 describes our implementation and evaluates our development on a large set of small benchmark programs. Sec 8 compares with related work prior to a short conclusion.

As a quick summary of contributions, we list key achievements below.

- A high-level specification logic, enriched with three temporal constraints, to support both correctness proofs and termination proofs.
- A set of entailment and Hoare rules to support the proof of well-founded measures for termination, and unreachability of method exit for non-termination.
- An inference mechanism to discover phase numbers needed by terminating methods with multiple phases.
- We outline soundness proofs for both termination and non-termination reasoning.
- We have implemented a verifier for our new specification logic for termination reasoning. The new logic is expressive: we have used it to specify and verify the termination behaviors for some 200+ benchmark programs collected from a variety of sources. Our implementation is available for download from:

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

2 Motivations for Our Proposal

We extend specification logics with *temporal constraints* that can specify the termination behaviors of programs. Each method of our program can be partitioned into a finite set of *phases*, with each phase described by a precondition on its inputs. The preconditions of these phases can be either disjoint or overlapping.

We introduce three temporal constraints: (i) Term M (ii) Loop (iii) MayLoop, that can be specified in the precondition of each *phase* of a method. These constraints can be used for termination reasoning, as follows:

If `Term M` is declared for the phase of a method m_1 , it signifies that any execution from this phase of the method will always terminate. We check this terminating behavior by ensuring that M (denoting a list of integer measures¹) is *bounded*; implemented as non-negative constraint $\forall i \in M \cdot i \geq 0$. Furthermore, every callee m_2 reachable from this phase must also have all its reachable phases be declared (and verified) with a terminating constraint each, say `Term N`, with a measure N that is lexicographically smaller than M ; implemented as $M >_l N$. These checks for temporal constraint `Term M` will guarantee that the given phase for our method m_1 always terminates.

If `Loop` is declared for a given phase, it signifies that any execution from this phase of the method will go into an infinite loop. This means that when we invoke such a phase of the method, its execution never returns. We may check for such non-terminating behavior by showing that the method's exit for such a phase is *unreachable*.

Lastly, temporal constraint `MayLoop` is to declare that the termination behavior of its phase is unknown. Each execution of this phase may or may not terminate. This constraint is handy for two reasons. Firstly, as termination detection is an undecidable problem, there are always some programs for which this declaration is necessary. Secondly, this constraint can always be used as a default, for any phase for which we have not yet proven its termination or its non-termination. As a result, we can easily support incremental specification of termination behaviors for our methods.

Let us look at some examples, starting with a termination-enriched specification for the `fact` method given earlier.

```
int fact (int n)
  case { n<0 → requires Loop ensures false;
         n=0 → requires Term[] ensures res=1;
         n>0 → requires Term[n] ensures res>0 ∧ res≥n; }
```

With the help of case structured specification (see [19]), we can identify three distinct phases for the `fact` method. When the input is $n < 0$, we have specified that the method will go into a loop². When the input is $n = 0$, we have specified its phase to be terminating with `Term[]`. No measure is needed here since there are no call invocations under this phase. When the input is $n > 0$, we have declared its phase to be terminating with `Term[n]` where n is a bounded measure that will decrease with each transition to its (recursive) callees. Note that callees to libraries (or primitives) for which termination behavior has already been verified can be simply annotated as `Term[]` since they can each be considered logically as a single terminating step.

Let us now look at how our temporal constraint can be added to a richer specification, based on separation logic. This can allow termination behaviors to be verified for

¹ We use a list of integers for simplicity, though other kinds of well-founded measures would work just as well.

² Our verification system assumes the use of arbitrary precision integer. In case finite integer is being modelled, we may give a different temporal behavior for this phase.

heap-manipulating programs. Consider a method to find the length of a linked-list.

```

data node { int val; node next }
predicate lseg(root, n, p)  $\equiv$  root=null  $\wedge$  n=0
     $\vee \exists v, q \cdot \text{root} \mapsto \text{node}(v, q) * \text{lseg}(q, n-1, p)$  inv n $\geq$ 0;
predicate clist(root, n)  $\equiv \exists v, q \cdot \text{root} \mapsto \text{node}(v, q) * \text{lseg}(q, n-1, \text{root})$ 
    inv n $>$ 0;
lemma clist(root, n)  $\leftarrow \exists v, q \cdot \text{lseg}(\text{root}, n-1, q) * q \mapsto \text{node}(v, \text{root})$ ;
int length (node x)
    requires lseg(x, n, null)@I  $\wedge$  Term[n]
    ensures res=n;
    requires clist(x, _)@I  $\wedge$  Loop
    ensures false;
    { if (x==null) return 0;
      else return 1+length(x.next); }

```

Such a `length` method is terminating for any acyclic linked list that can be specified by $\text{lseg}(x, n, \text{null})@I^3$ which denotes a list segment of length n , terminated by a `null` value. This n integer, courtesy of the `lseg` predicate from separation logic, can be directly used in our temporal constraint $\text{Term}[n]$ as the terminating phase's well-founded measure. Furthermore, we have also specified a different scenario with a circular list, $\text{clist}(x, _)@I$, for which the method can be proven to be non-terminating. The specification for this scenario is aided by the `clist` predicate, together with a (provable) lemma that gives another view of the circular list, which is required when checking the precondition of the recursive `length` call. Though the specification logic for this example is considerably more complex, we can perform termination reasoning in pretty much the same way. Thus, termination reasoning for heap-manipulating programs can be directly handled within the proving infrastructure of its richer specification logic.

Our last example is a simple loop whose termination proof is *not* so obvious.

```

while (x>y)
  case { x $\leq$ y  $\rightarrow$  requires Term[] ensures x' $\leq$ y';
        x>y  $\wedge$  x $\leq$ 1  $\rightarrow$  requires Term[1, -x+1] ensures x' $\leq$ y';
        x>y  $\wedge$  x>1  $\rightarrow$  requires Term[0, x-y] ensures x' $\leq$ y'; }
  { y=x+y; x=x+1; }

```

We provide a specification for this loop using pre/post conditions with three distinct phases. The first phase with pre-condition $x \leq y$ denotes the loop's exit. The second phase with temporal constraint $\text{Term}[1, -x+1]$ denotes a cyclic phase where the bounded measure $-x+1$ is progressively decreased. The last phase with temporal constraint $\text{Term}[0, x-y]$ denotes a second cyclic phase for which the bounded measure $x-y$ is progressively decreased. The two cyclic phases have two distinct measures $-x+1$ and $x-y$ that are decreasing for their respective phases, but may be incompatible otherwise. To maintain a well-founded measure for both phases, we prefix the two cyclic phases with two distinct numbers, 1 and 0, respectively. This is allowed since it is possible for

³ The $@I$ annotation from [13] specifies that the linked-list is never mutated.

the first cyclic phase to reach the second phase, but not vice-versa. Such an annotation on the phases could come from the programmer’s knowledge on the loop’s behavior, but we have also developed an automatic inference mechanism for phase numbers. The concept of phases is not new, and has been previously proposed in [27, 9]. What is new in our work is the concept of phase numbering and the context from which they are being inferred, namely through an integration of termination reasoning into specification logics. One pleasant outcome is that we did not have to develop this inference mechanism (for phase numbers) from scratch. Instead, we can leverage on an existing inference mechanism that has been independently developed for our specification logic. This gives a good motivation for infusing termination constraints into our logic, since it can directly benefit from inferencing efforts that are being developed for the underlying logic. Sec 5 describes more details on how inference mechanism is used to discover phase number annotations for our temporal constraints.

3 Temporal Constraints for Termination Reasoning

The core of our approach is the embedding of termination arguments in a logical framework by allowing formulae to contain constraints on temporal properties. The formula $\text{Term } M$ (for a given phase of a method) denotes that the execution is required to terminate (in that phase). In contrast, the formula Loop denotes that the execution is required to run forever (diverge). Finally, MayLoop indicates that the execution can diverge or terminate. In the general case of our calculus, termination arguments are equipped with arbitrary measures that can be shown to be well-founded; however, our implementation specializes them to lists of arithmetic expressions over integers.

The *temporal entailment* judgment, written $\Phi \vdash \vartheta_1 \gg \vartheta_2 \rightsquigarrow \Phi_r$, allows us to state a set of valid reasoning rules for our temporal formulae that are independent of both the overarching separation logic (that is being used as our underlying logic) and the domain of termination measures. Temporal entailment means that starting from an initial context (state) that satisfies the flat (non-temporal) formula Φ and temporal formula ϑ_1 , we can transition to a state that satisfies the temporal formula ϑ_2 with a possible residue Φ_r which our tool uses to aid frame (and precondition) inference.

Temporal entailment is useful during program verification because termination related entailments appear during the verification of method calls. That is, from a state

$$\begin{array}{c}
 \Phi \vdash (x:X)_{>l} [] \rightsquigarrow \Phi \quad \frac{\Phi \vdash x > y \rightsquigarrow \Phi_1 \quad \Phi \vdash x \geq y \rightsquigarrow \Phi_2 \quad \Phi_2 \vdash X >_l Y \rightsquigarrow \Phi_3}{\Phi \vdash (x:X)_{>l} (y:Y) \rightsquigarrow \Phi_1 \vee \Phi_3} \\
 \\
 \begin{array}{cc}
 \Phi \vdash \text{MayLoop} \gg \text{Term } X \rightsquigarrow \Phi & \Phi \vdash \text{Loop} \gg \text{Term } X \rightsquigarrow \Phi \\
 \Phi \vdash \text{MayLoop} \gg \text{MayLoop} \rightsquigarrow \Phi & \Phi \vdash \text{Loop} \gg \text{MayLoop} \rightsquigarrow \Phi \\
 \Phi \vdash \text{MayLoop} \gg \text{Loop} \rightsquigarrow \Phi & \Phi \vdash \text{Loop} \gg \text{Loop} \rightsquigarrow \Phi
 \end{array} \\
 \\
 \frac{\Phi \vdash X >_l Y \rightsquigarrow \Phi_r}{\Phi \vdash \text{Term } X \gg \text{Term } Y \rightsquigarrow \Phi_r}
 \end{array}$$

Fig. 2. Entailment for Temporal Constraints

that requires termination condition ϑ_1 , we can only call (the phases of) methods whose termination guarantee ϑ_2 is compatible with ϑ_1 . For example, if we are verifying the phase of a method that must terminate, then we are only allowed to call other terminating (phases of) methods (including ourselves), in such a way that its specified termination measure is decreased across the call transition. In contrast, if we are verifying the phase of a method that must or may loop, we are allowed to call (the phases of) both terminating and nonterminating methods. This idea is formalized as key properties of temporal entailment in Fig. 2. Take note that unspecified transitions, such as $\text{Term } X \gg \text{Loop}$, are flagged as temporal errors detected by our reasoning system.

Fig. 2 also defines the lexicographical ordering relation $>_l$. Note that a list is either empty, $[],$ or non-empty, $(x:X),$ with x as its head and X as its tail. One benefit of our approach is our ability to offload the decreasing measure proof to the entailment checker for the underlying logic. This allows us to leverage on other capabilities (such as inference) from the underlying entailment engine.

4 A Specification Logic for Termination

The key point of our termination formulae is that they can be straightforwardly embedded in existing specification logics. We have found that the integration of our termination calculus with separation logic is well suited for reasoning about the termination of an interesting class of heap manipulating programs. Our base separation logic includes complex features such as structured specifications and case analysis; our new mixed logic can freely use these features, resulting in an expressive specification language.

$Y ::= \text{case } \{\pi_1 \Rightarrow Y_1; \dots; \pi_n \Rightarrow Y_n\} \quad \quad \text{requires } \Psi \quad Y \quad \quad \text{ensures } \Phi$ $\Phi ::= \bigvee \exists v^* \cdot (\kappa \wedge \pi) \quad \Psi ::= \bigvee \exists v^* \cdot (\kappa \wedge \pi [\wedge \vartheta])$ $\vartheta ::= \text{Loop} \quad \quad \text{MayLoop} \quad \quad \text{Term } X \quad \kappa ::= \text{emp} \quad \quad v \mapsto c(v^*) \quad \quad p(v^*) \quad \quad \kappa_1 * \kappa_2$
--

Fig. 3. Specification Logic with Temporal Termination Constraints

We summarize the syntax for our specifications in separation logic with termination in Fig. 3. We denote sequences of variables v_1, \dots, v_n by v^* and disjunction-free pure formulae by π (e.g., in/equality of program variables and arithmetic expressions). A heap formula κ is either: emp , denoting the empty heap; $v \mapsto c(v^*),$ denoting a singleton heap containing the simple data record $c(v^*)$ (e.g. the node of a tree); an inductively defined predicate $p(v^*);$ or $\kappa_1 * \kappa_2,$ where κ_1 and κ_2 are heap formulae and $*$ is the separating conjunction. Predicates are defined as the equivalence between a predicate symbol $p(v^*)$ and disjunctions of the form $\exists v^* \cdot (\kappa \wedge \pi),$ where variables v^* may appear free; for further details see [28].

The novel part of Fig. 3 is the temporal constraint $\vartheta.$ As indicated in Sec 3, we have three temporal constraints to capture: guaranteed non-termination $\text{Loop},$ possible non-termination $\text{MayLoop},$ or guaranteed termination $\text{Term } X,$ where X is a ranking function (termination measure) on variables shared with the separation logic formula. Termination requires that the ranking function X be bounded below and decreasing.

It may be difficult to construct a single ranking function for the entire program, so it is common to give the verifier the flexibility to specify a (lexicographically ordered) list of ranking functions. As described later in Sec 5, we automatically insert measures that are related to call hierarchy and phase numbering, leaving only method-specific ranking functions to be explicitly specified.

Observe that adding the termination fragment of our logic changes the interpretation of formulae in a subtle way. Typically, formulae describe a program state at a fixed point in time. In contrast, our termination annotations have a temporal flavor: rather than describing the current state, they prescribe the *future execution* of the program.

4.1 Entailment System

Here we show how to construct an entailment system with frame inference capabilities for separation logic with temporal constraints from an entailment system for separation logic and the procedure described in Sec 3. The entailment system for separation logic uses judgments of the form $\Phi \vdash Y \rightsquigarrow \Phi_r$ meaning that Φ implies Y with frame Φ_r ; here Y must not contain any temporal constraints. Although our implementation is more general in handling proof search through set of states of the form $\{\Phi_1, \dots, \Phi_n\}$, for the presentation we assume only singleton state of the form Φ .

Updating the entailment procedures to handle termination constraints required only minimal changes due to the structure of the entailment system and the modularity of the termination calculus. The resulting judgment, namely $\Psi \vdash Y \rightsquigarrow \Psi_r$, has a more general form (with temporal constraints) than just separation logic entailment. Furthermore, Y is allowed to contain temporal constraints too. The original separation logic entailment deconstructs the antecedent disjunctions until the $\kappa \wedge \pi$ form is reached and then proceeds to deconstruct the consequent until the pure π form is encountered. Updating the entailment to handle temporal constraints required only minor modifications to the deconstruction rules; as an example we show the rule $\boxed{\text{ENT-LHS-D}}$. The only significant change to the entailment procedure was the addition of the $\boxed{\text{ENT-term}}$ rule. The $\boxed{\text{ENT-term}}$ rule splits an entailment containing termination constraints into two parts, *temporal* and *spatial*. The temporal goal is solved using the rules presented in Fig. 2, which discharge the temporal constraints and return a residue context Φ that is usually just $\kappa_a \wedge \pi_a$ (except when inference is activated) The spatial goal feeds this context into the underlying separation logic entailment procedures to discharge the remaining goals.

$$\begin{array}{c}
 \boxed{\text{ENT-LHS-D}} \\
 \frac{\Psi = \bigvee \exists v_i^* \cdot (\kappa_i \wedge \pi_i [\wedge \vartheta_i]) \quad \forall i \cdot (\kappa_i \wedge \pi_i \wedge [\wedge \vartheta_i] \vdash Y \rightsquigarrow \Psi_r^i)}{\Psi \vdash Y \rightsquigarrow \bigvee \exists v_i^* \cdot \Psi_r^i}
 \end{array}
 \qquad
 \begin{array}{c}
 \boxed{\text{ENT-term}} \\
 \frac{\kappa_a \wedge \pi_a \vdash \vartheta_a \gg \vartheta_c \rightsquigarrow \Phi \quad \Phi \vdash \pi_c \rightsquigarrow \Phi_r}{\kappa_a \wedge \pi_a \wedge \vartheta_a \vdash \pi_c \wedge \vartheta_c \rightsquigarrow (\Phi_r \wedge \vartheta_a)}
 \end{array}$$

Observe that due to the incorporation of temporal constraints, the \vdash symbol does not precisely mean “entails” in the usual sense. For example, from MayLoop on the LHS we can reach both Loop and Term X on the RHS, an obvious contradiction under the usual meaning of entailment. Instead, given a temporal constraint ϑ on the LHS, a temporal constraint ϑ' on the RHS means that *calling a function whose termination behavior is ϑ' is allowed by our current temporal constraint ϑ .*

$$\begin{array}{c}
\frac{}{\text{CheckLoop}(\kappa \wedge \pi \wedge \text{Term } \bar{X})} \quad \frac{}{\text{CheckLoop}(\kappa \wedge \pi \wedge \text{MayLoop})} \\
\frac{\text{CheckLoop}(\Psi_1) \quad \text{CheckLoop}(\Psi_2)}{\text{CheckLoop}(\Psi_1 \vee \Psi_2)} \quad \frac{\kappa \wedge \pi \vdash \text{false}}{\text{CheckLoop}(\kappa \wedge \pi \wedge \text{Loop})} \\
\frac{}{\text{Bound}(\kappa \wedge \pi \wedge \text{MayLoop})} \quad \frac{}{\text{Bound}(\kappa \wedge \pi \wedge \text{Loop})} \\
\frac{\text{Bound}(\Psi_1) \quad \text{Bound}(\Psi_2)}{\text{Bound}(\Psi_1 \vee \Psi_2)} \quad \frac{\forall i \in \{1..n\} \cdot (\kappa \wedge \pi \vdash x_i \geq 0)}{\text{Bound}(\kappa \wedge \pi \wedge \text{Term}[x_1, \dots, x_n])} \\
\frac{\frac{[\text{FV-call}]}{t_0 \text{ mn}((t v)^*) \text{ Y } \{\text{code}\} \in \text{Program} \quad \Psi \vdash \text{Y} * \Psi_r}}{\vdash \{\Psi\} \text{mn}(v^*) \{\Psi_r\}}}{\vdash \{\Psi\} \text{mn}((t v)^*) \text{ Y } \{\text{code}\}} \quad \frac{\frac{[\text{FV-ret}]}{\Psi_r = \Psi \wedge \text{res} = v} \quad \text{CheckLoop}(\Psi)}{\vdash \{\Psi\} \text{return } v \{\Psi_r\}} \\
\frac{[\text{FV-method}]}{\frac{\{(\Psi_{\text{Pre}}^i, \Phi_{\text{Post}}^i)\}_{i=1}^n = \text{split_specs}(\text{Y})}{\forall i \in \{1..n\} \cdot (\text{Bound}(\Psi_{\text{Pre}}^i) \wedge \vdash \{\Psi_{\text{Pre}}^i\} \text{code } \{\Psi^i\} \wedge \Psi^i \vdash \Phi_{\text{Post}}^i \rightsquigarrow \Psi_r^i)}}{\vdash t_0 \text{ mn}((t v)^*) \text{ Y } \{\text{code}\}}}
\end{array}$$

Fig. 4. Hoare Verification Rules: Method Call and Declaration

4.2 Hoare Logic

In Fig. 4, we give a fragment of a Hoare logic for an imperative programming language⁴. We only show the rules relevant to the termination logic, which are the Hoare rules for return and function call, as well as the whole-method verification judgment.

It is common in termination reasoning to explicitly modify the method call rule to require that the current measure has decreased. In our system, this check is done implicitly within the entailment check, resulting in a standard-looking Hoare call rule $[\text{FV-call}]$.

In contrast, we have modified the Hoare rule for function return $[\text{FV-ret}]$ to include the loop consistency check CheckLoop . The CheckLoop test guarantees nontermination if our temporal context is Loop by proving that method exit points are unreachable.

The final rule of interest is the whole-method verification rule $[\text{FV-method}]$. This rule first relies on an auxiliary method split_specs to split a given specification Y into a set of pairs of pre and post conditions whose structure is described by Ψ and Φ from Fig. 3. Then, for each pair $(\Psi_{\text{Pre}}^i, \Phi_{\text{Post}}^i)$, we check that:

1. the measure in precondition Ψ_{Pre}^i is properly bounded using the Bound check, and
2. the Hoare judgment verifies that the function body code transforms the precondition Ψ_{Pre}^i to some poststate Ψ^i , and

⁴ Our core language does not include loops, which will be automatically translated from the source language into tail-recursive methods with pass by reference parameters.

- the calculated poststate Ψ^i entails the postcondition of the function Φ_{Post}^i with some residue Ψ_r^i (we can prove the absence of memory leaks by requiring $\Psi_r^i = \text{emp}$).

5 A Benefit of Integration

One of the benefits of integrating termination constraints into an existing specification logic is the possibility of utilizing the infrastructure (either proving or inference) that has been developed for that logic. In a separate work, we have recently developed inference mechanisms that can help systematically and selectively derive partial specifications for our logic. Let us firstly describe the general form of our inference rule, that is captured as a judgment below:

$$\text{infer } [v^*] \Phi_1 \vdash \Phi_2 \rightsquigarrow (\Phi_{res}, \Phi_{pre})$$

Given a state Φ_1 and a set of input variables v^* , the above judgment attempts to prove that the given Φ_1 would entail Φ_2 and result in a frame residue Φ_{res} . Furthermore, it is allowed to strengthen the given state *selectively* using the input variables v^* through a derived precondition formula Φ_{pre} . Such an inference mechanism for suitable precondition was proposed for separation logic in [8], and has been referred to as an *abduction* mechanism. An example in pure logic is $\text{infer } [y] x=y+1 \vdash x>0 \rightsquigarrow (\dots, y \geq 0)$, where $y \geq 0$ denotes the weakest precondition that would allow the entailment to succeed. We have recently implemented a version of this abduction mechanism for our specification logic, with support for user-defined predicates and lemmas. We found it practical enough to be used for inferring phase numbers needed for termination constraints. We describe below how it is done.

In any given program, we will have a set of methods some of which would have phases that are annotated with termination constraints of the form $\text{Term } M$. For each of such terminating constraint $\text{Term } M_i$, we shall add two extra measures, c_i and p_i , to give us a more comprehensive measure $(c_i : p_i : M_i)$. The c_i measure is a constant that corresponds to the position of the mutual-recursive set of methods (including loops) in the call hierarchy, such that those in the same mutual-recursive set will have the same constant measure, while methods that are lower in the hierarchy would have a smaller constant compared to their caller. This call numbering can be done automatically after a call graph is built. Phase numbering is then performed for each set of mutual-recursive methods (including loops) but this is more tricky since some call transitions may be unreachable. To do that, we use a logical variable for each phase before performing a verification process with the help of entailment with inference capability. As illustration, for our loop example in Sec 2, we would initially generate the following specification where p_1, p_2, p_3 as logical variables for the three terminating phases, and the first 0 constant measure denotes the call measure for the loop:

```
infer [p1, p2, p3]
case { x ≤ y → requires Term[0, p1] ensures x' ≤ y';
      x > y ∧ x ≤ 1 → requires Term[0, p2, -x+1] ensures x' ≤ y';
      x > y ∧ x > 1 → requires Term[0, p3, x-y] ensures x' ≤ y'; }
```

$S, \mathcal{H}, \mathcal{I} \models \Psi_1 \vee \Psi_2$	\equiv	$S, \mathcal{H}, \mathcal{I} \models \Psi_1$ or $S, \mathcal{H}, \mathcal{I} \models \Psi_2$
$S, \mathcal{H}, \mathcal{I} \models \exists x_i^*. (\kappa \wedge \pi \wedge \vartheta)$	\equiv	$\exists \nu_i^*. \mathcal{S}[(x_i \mapsto \nu_i)^*], \mathcal{H}, \mathcal{I} \models \kappa$ and $\mathcal{S}[(x_i \mapsto \nu_i)^*] \models \pi$ and $\mathcal{S}[(x_i \mapsto \nu_i)^*], \mathcal{H}, \mathcal{I} \models \vartheta$
$S, \mathcal{H}, \mathcal{I} \models \text{Term } X$	\equiv	$\text{top}(\mathcal{I}) = X$
$S, \mathcal{H}, \mathcal{I} \models \text{Loop}$	\equiv	$\text{top}(\mathcal{I}) = \infty_L$
$S, \mathcal{H}, \mathcal{I} \models \text{MayLoop}$	\equiv	$\text{top}(\mathcal{I}) = \infty_M$
$S, \mathcal{H}, \mathcal{I} \models \kappa_1 * \kappa_2$	\equiv	$\exists \mathcal{H}_1, \mathcal{H}_2. \mathcal{H}_1 \perp \mathcal{H}_2$ and $\mathcal{H} = \mathcal{H}_1 \oplus \mathcal{H}_2$ and $S, \mathcal{H}_1, i \models \kappa_1$ and $S, \mathcal{H}_2, i \models \kappa_2$
$S, \mathcal{H}, \mathcal{I} \models \text{emp}$	\equiv	$\text{dom}(\mathcal{H}) = \emptyset$
$S, \mathcal{H}, \mathcal{I} \models p \mapsto c \langle v_i^* \rangle$	\equiv	exists a data type decl. $\text{data } c \{ (t_i f_i)^* \}$ and $\mathcal{H} = [\mathcal{S}(p) \mapsto r]$ and $r = [(f_i \mapsto \mathcal{S}(v_i))^*]$
$S, \mathcal{H}, \mathcal{I} \models p \langle v_i^* \rangle$	\equiv	exists a pred. def. $p \langle v_i^* \rangle \equiv \Phi$ and $S, \mathcal{H}, \mathcal{I} \models \Phi$

Fig. 5. Model for Formulae of Separation Logic with Temporal Constraints

Due to selective inference on phase variables, p_1, p_2, p_3 , our entailment prover (with inference) is able to return a precondition: $p_2 \triangleright p_3 \wedge p_3 \geq p_1$. This precondition captures two phase transitions that are shown to be feasible by our code verifier, and the conditions under which a well-founded decreasing measure can be guaranteed by our entailment. Using this precondition, we can directly generate a phase numbering substitution, $[p_1 \mapsto 0, p_2 \mapsto 1, p_3 \mapsto 0]$, that always satisfies the inferred precondition. Thus, phase numbering can be automatically synthesized and is a direct beneficiary from our decision to embed temporal termination constraints into our specification logic.

6 Semantics and Soundness

Our goal here is to demonstrate the soundness of our termination-aware Hoare logic. First, we provide a model for program state that allows us to define our new temporal formulae. Second, we define an operational semantics that gets stuck on transitions that would falsify termination constraints. Finally, we define that Hoare triple with respect to our operational semantics and prove soundness.

Temporal formulae. Traditionally, program state consists of a stack \mathcal{S} (locals) and heap \mathcal{H} (memory). To model the temporal constraints we add a new component \mathcal{I} called the *termination stack*, which is a list of *counters* i . Each counter is an element of the set $\mathbb{N}^{\leq c} \cup \infty_L \cup \infty_M$, where ∞_L and ∞_M are two distinguished symbols $\notin \mathbb{N}^{\leq c}$, and c is the maximum number of ranking functions. The domain $\mathbb{N}^{\leq c}$ captures lists of natural numbers with lengths that are not more than c . For any given program, c is always a constant. As we will see, the operational semantics will get stuck if we try to make a function call when the head of \mathcal{I} is less than the callee's termination specification (that is, when the callee will take more time than is available) or if we try to exit a function

$$\begin{array}{c}
\frac{}{\infty_L >_L i} \quad \frac{}{\infty_M >_L i} \quad \frac{}{(x:X) >_L []} \quad \frac{x > y}{(x:X) >_L (y:Y)} \quad \frac{x \geq y \quad X >_L Y}{(x:X) >_L (y:Y)} \\
\\
\frac{t_0 \text{ mn}((t v)^*) \{ \text{code} \} \in \text{Program} \quad \exists n \cdot \text{top}(\mathcal{I}) >_L n}{\langle (S, \mathcal{H}, \mathcal{I}), \text{mn}(w^*) \rangle \leftrightarrow \langle ([v \mapsto S[w]]^* : S, \mathcal{H}, n : \mathcal{I}), \text{code} \rangle} \\
\\
\frac{i_t \neq \infty_L}{\langle (s_t : S, \mathcal{H}, i_t : \mathcal{I}), \text{return } v \rangle \leftrightarrow \langle S \upharpoonright_{res \mapsto st[v]}, \mathcal{H}, \mathcal{I}, \text{nop} \rangle}
\end{array}$$

Fig. 6. Key Rules in Operational Semantics

(*i.e.*, `return`) when the head of \mathcal{I} is ∞_L . In Fig. 5, we model the formulae of our logic over the program state $(S, \mathcal{H}, \mathcal{I})$.

Operational semantics. We have modified a standard small-step operational semantics to incorporate our termination stack. In Fig. 6, we show the method call and return rules; the other rules do not interact with the termination stack in any interesting way. Fig. 6 also defines an ordering relation $>_L$ over counters, which determines if the metric allows a new method call or not. This $>_L$ relation is tightly related to the temporal entailment explained in Sec 3.

The method call operational rule ensures that the elements of the counter stack is always ordered with respect to the $>_L$ relation; that is, for any $\mathcal{I} = [i_1, \dots, i_n]$, we have $i_n >_L \dots >_L i_1$. The semantics will not allow (*e.g.*, eventually get stuck) infinite executions from states in which the counter stack contains only finite elements (*i.e.*, in $\mathbb{N}^{\leq c}$). This follows from two observations. First, finite counters place a finite bound on depth of the call stack; second, there are only finite number of recursion points in a method body⁵. Therefore, infinite executions are only possible if the counter stack contains an infinite element.

The return operational rule ensures that functions that are supposed to loop forever never return. If a function whose termination context forces `Loop` (*i.e.*, whose top termination counter is ∞_L) tries to return, it gets stuck.

Hoare Triples. We define the Hoare triple in a continuation-passing style as in Appel and Blazy [2]. A *configuration* is a pair of code k and state σ . We say a configuration is *safe*, written $\text{safe}(k, \sigma)$, if all reachable states are safely halted or can continue to step:

$$\text{safe}(k, \sigma) \equiv \forall k', \sigma' \cdot \langle \sigma, k \rangle \leftrightarrow^* \langle \sigma', k' \rangle \rightarrow \\
(k' = \text{nop} \vee \exists \sigma'', k'' \cdot \langle \sigma', k' \rangle \leftrightarrow \langle \sigma'', k'' \rangle)$$

We say that a formula F guards code k , written $\text{guards}(F, k)$ when the code k is safe on any state accepted by F :

$$\text{guards}(F, k) \equiv \forall \sigma \cdot \sigma \models F \rightarrow \text{safe}(k, \sigma)$$

⁵ As mentioned previously, our language does not have loops—in fact, a preprocessing step transforms loops into tail-recursive functions.

We now define the Hoare triple $\{\Psi\}c\{\Phi\}$ in a continuation passing style using guards:

$$\{\Psi\}c\{\Phi\} \equiv \forall F, k. \text{guards}(F * \Phi, k) \rightarrow \text{guards}(F * \Psi, c; k)$$

We conclude by stating and sketching the proofs for soundness.

Theorem 1 (Safety). *If $\vdash \{\Psi\}c\{\Phi\}$ then $\forall \sigma. \sigma \models \Psi \rightarrow \text{safe}(c, \sigma)$*

Proof. For all σ , $\text{safe}(\text{nop}, \sigma)$, so for all Φ , $\text{guards}(\Phi, \text{nop})$. If we instantiate $F = \text{emp}$ and $k = \text{nop}$ in the Hoare triple definition then safety follows immediately.

Theorem 2. *The standard Hoare rules (e.g., assignment, frame, conditional, sequential composition) are sound with respect to the semantics of our Hoare judgment.*

The proofs of these theorems follows Appel and Blazy [2].

Theorem 3. *The Hoare rules for method call and return are sound.*

Proof sketch. The proof of the return rule is standard except in the case of returning from a function satisfying `Loop`, for which the operational semantics get stuck. However, the Hoare rule further requires that the `CheckLoop` relation holds, meaning that the code is unreachable (*i.e.*, that the precondition of the `return` instruction was `false`). Thus, a proper Hoare derivation guarantees that looping code never returns.

The proof for the method call rule hinges on the proof that the precondition guarantees that there exists a smaller (in $>_L$) termination measure that suffices for the callee. The temporal entailment from Sec 4.1 guarantees exactly this, meaning that a proper Hoare derivation guarantees that all function calls are decreasing in the termination space. This guarantees termination once any finite (*i.e.*, in $\mathbb{N}^{\leq c}$) measure is obtained.

7 Experiments

We have implemented the proposed termination logic into an automated verification system, called *HipTNT*⁶, to allow us to verify both termination and non-termination properties, in addition to correctness properties. In this system, the final proof obligations are automatically discharged by off-the-shelf provers, such as Omega Calculator [31], MONA [24] and Redlog [16]. The new verification system is evaluated using a benchmark of over 200+ small programs selected from a variety of sources: (i) from the literatures [9, 14, 11, 4, 10, 22], (ii) from benchmarks used by other systems (*i.e.*, AProVE [20], Invel [32] and Pasta [17]) and (iii) some realistic programs, such as the Microsoft Zune’s clock driver that has a leap-year bug. Most of the methods in these benchmark programs contain either terminating or non-terminating code fragments, expressed in (mutual) recursive methods or (nested) loops. Many of these methods (or loops) also contains multiple phases which can be automatically handled.

Fig. 7 summarizes the characteristics and the verification times for a benchmark of small numerical programs. Columns 3-5 describes the number of phases that have been

⁶ Our implemented system and benchmark programs used can be downloaded from <http://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/>

Benchmarks	Programs	Term	Loop	MayLoop	PC(s)	TC(s)	Overhead (%)
Invel	58	121	75	6	10.51	10.69	1.68
AProVE	105	689	103	12	16.36	18.08	9.51
Pasta	44	219	10	3	4.50	5.23	13.96
Others	25	126	17	15	3.58	3.83	6.53
Totals/(%)	232	1155 (82.7%)	205 (14.7%)	36 (2.6%)	34.95	37.83	(7.6%)

Fig. 7. Termination Verification for Numerical Programs

Programs	Methods	Term	Loop	MayLoop	PC(s)	TC(s)	Overhead (%)
AVL	13	19	0	0	8.05	8.10	0.62
Linked List (LL)	13	13	0	0	0.41	0.44	6.82
Sorted LL	13	15	0	0	1.27	1.34	5.22
Circular LL	4	4	4	0	1.54	1.65	6.67
Doubly LL	11	12	0	0	0.54	0.58	6.90
Complete	6	7	0	0	2.57	2.99	14.05
Heap Tree	5	6	0	0	3.60	3.90	7.69
BST	6	6	0	0	0.86	0.87	1.15
Perfect Tree	5	5	1	0	0.32	0.34	5.88
Red-Black Tree	19	25	0	0	5.77	5.85	1.20
Totals/(%)	95	112(95.7%)	5(4.3%)	0	24.93	26.05	(4.3%)

Fig. 8. Termination Verification for Heap-manipulating Programs

verified as *terminating*, *non-terminating* or *unknown*, respectively. As hoped for, the number of MayLoop phases occupies the smallest fragment ($<3\%$) of the total number of phases. Such MayLoop phases were required in programs with undecidable termination property, such as the Collatz problem, or in programs whose termination property is dependent on users' inputs. There are also a small number of cases (4 programs) for which we have not figured the termination behavior of the loops, and have simply used MayLoop for expediency reasons. The Term phases are in the majority because most of the methods are expected to be terminating, except for the Invel's benchmark which focus on mostly non-terminating programs. Our verification system can perform both correctness and termination proofs. Column 7 gives the total timings (in seconds) needed to perform both termination and correctness proofs for all the programs in each row, while column 6 gives the timings needed for just correctness proofs. The difference in the two timings represent the small overheads needed for termination reasoning.

We have also conducted termination reasoning on our own benchmark of heap-based programs to classify the termination behaviors of these programs, as shown in Fig. 8. Due to tight integration with the underlying logic, this task of specifying the termination properties was pretty easy even though some of the programs use non-trivial data structures, such as red-black and AVL-trees. We have successfully determined that none of the methods have any unknown termination behaviors. All the methods were terminating, except for some methods in circular list and perfect tree. In the case of the latter, a method to create perfect tree would go into an infinite loop if a negative number was given as its height. Furthermore, during the specification of termination properties, we discovered a bug in our own merge method (for two AVL trees) that

went into a loop due to wrong parameter order. Partial correctness proof did not detect this problem. It was later corrected into a terminating method, courtesy of our newly integrated feature. The bug discovery benefit of termination reasoning was also seen for a known bug problem of the Zune’s clock driver. Termination property of this driver, which expects all methods to terminate with any input, cannot be proven due to the presence of a leap-year bug. Our tool can soundly confirm this non-termination bug through the Loop temporal constraint.

8 Related Work

Temporal logics [23] have been previously proposed for reasoning on a wide range of liveness properties, including those applicable to concurrent systems, such as fair termination. However they are less suited for our more limited goal of proving termination and program safety for sequential programs, for two reasons: (i) they are more difficult to integrate into existing safety-oriented correctness verifiers, and (ii) they tend to focus on low level descriptions of the execution behavior (over a variety of automata) making them less compatible with specification logics for conventional programs.

Many existing verification systems are also able to construct both termination and correctness proofs. One example is Frama-C [12] which uses ACSL [3] as a specification language for ANSI/ISO C and the Why platform for constructing and discharging proof obligations. Due to the lack of integration with the underlying logics, the variance arguments needed are limited in a number of ways. Firstly, they cannot be used to reason about non-termination. Secondly, while they can easily handle numerical programs, it is less clear how to prove termination properties for heap-based programs since predicates, lemmas and updates on heap would have to be suitably supported. Lastly, only linear variance and single phase are currently supported in most of the systems.

There were two past works that we are aware of on termination proving of heap-based programs. In [5], Berdine *et al.* used the number of inductive unfoldings on heap predicates as an implicit ranking function to support termination proofs. In [7], Brotherson *et al.* proposed a radically different approach based on the *cyclic proof* which did not require any explicit ranking function. In contrast, our approach uses explicit ranking measures that are possibly non-linear. Through our use of temporal constraints, we have also allowed non-termination to be specified and verified.

Most past works have focused on termination proving [25, 6, 29, 11, 15] where good inference mechanisms have been developed for (mostly linear) measures. However, more recent works [21, 32, 20, 30] have also focused their attention towards automatically proving non-termination. In [21], the authors introduced a model-checking based approach that searches for an infinite program execution as a counterexample to termination. Some of these systems could handle both termination and non-termination detections. For example, the AProVE system [20] uses term-rewriting system to perform its analysis, while the system proposed in [32] analyse imperative loops by using termination graphs in conjunction with constraint solving techniques. Our proposal is complimentary to these past works since our aim is to integrate both termination and non-termination constraints directly into specification logics. We have done so, and

have also successfully evaluated its applicability on a wide range of programs, covering both loops and recursion, as well as numeric and heap-manipulating programs.

9 Conclusion

Termination reasoning has been intensively studied in the past, but it remains a challenge for the technology developed there to keep up with improvements to specification logic infrastructure, and vice versa. We propose an approach that would seal the fate of both areas more closely together, through a tightly coupled union. Our unique contribution is to embed termination reasoning directly into specification logics which we believe would have long-term benefits. Its expressivity is enhanced by any improvement to the underlying logics. It can also benefit from infrastructures that have been developed for the underlying logics, including those that are related to inferencing. Last, but not least, it has placed termination reasoning as a first-class entity, much like what was originally envisioned (in principle) by Hoare's logic for total correctness.

Acknowledgement We thank Cristina David for helpful feedbacks on the paper, and Minh-Thai Trinh for his efforts in implementing the inference sub-system.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
2. Andrew W. Appel and Sandrine Blazy. Separation logic for small-step C minor. In *TPHOLS*, pages 5–21, 2007.
3. Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliatre, Claude Marche, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*.
4. Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano, and Peter O'Hearn. Variance analyses from invariance analyses. *SIGPLAN Not.*, 42(1):211–224, 2007.
5. Josh Berdine, Byron Cook, Dino Distefano, and Peter O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *Computer Aided Verification*, volume 4144 of *LNCIS*, pages 386–400. Springer Berlin / Heidelberg, 2006.
6. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The Polyranking Principle. In *ICALP*, volume 3580 of *LNCIS*, pages 1349–1361. Springer Berlin / Heidelberg, 2005.
7. James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *ACM POPL*, pages 101–112, New York, NY, USA, 2008.
8. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, pages 289–300, 2009.
9. Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving Conditional Termination. In *Proceedings of the 20th international conference on Computer Aided Verification*, pages 328–340, Berlin, Heidelberg, 2008. Springer-Verlag.
10. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.
11. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *ACM PLDI*, pages 415–426. ACM, 2006.
12. Pascal Cuoq, Benjamin Monate, Anne Pacalet, and Virgile Prevosto. Functional dependencies of C functions via weakest pre-conditions. *International Journal on Software Tools for Technology Transfer (STTT)*, 13:405–417, 2011.

13. Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. In *OOPSLA*, pages 359–374, 2011.
14. Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1st edition, 1997.
15. Robert Dockins and Aquinas Hobor. A theory of termination via indirection. In Amal Ahmed, Nick Benton, Lars Birkedal, and Martin Hofmann, editors, *Modelling, Controlling and Reasoning About State*, number 10351 in Dagstuhl Seminar Proceedings, 2010.
16. Andreas Dolzmann and Thomas Sturm. REDLOG: computer algebra meets computer logic. *SIGSAM Bull.*, 31:2–9, June 1997.
17. Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE*, pages 277–293, 2009.
18. Cristian Gherghina and Cristina David. A specification logic for exceptions and beyond. In *ATVA*, pages 173–187, 2010.
19. Cristian Gherghina, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Structured specifications for better verification of heap-manipulating programs. In *FM*, pages 386–401, 2011.
20. Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Automatic termination proofs in the dependency pair framework. In *IJCAR*, pages 281–286, 2006.
21. Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *ACM POPL*, pages 147–158. ACM, 2008.
22. William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. Alternation for Termination. In *SAS*, pages 304–319. Springer-Verlag, 2010.
23. Yonit Kesten, Amir Pnueli, and Li on Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. 25th Int. Colloq. Aut. Lang. Prog., volume 1443 of Lect. Notes in Comp. Sci.*, pages 1–16. Springer-Verlag, 1998.
24. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
25. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM POPL*, pages 81–92. ACM Press, 2001.
26. K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR*, pages 348–370. Springer-Verlag, 2010.
27. Zohar Manna, Anca Browne, Henny Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In *AMAST*, pages 28–41. Springer-Verlag, 1999.
28. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, pages 251–266, January 2007.
29. Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *ACM POPL*, pages 132–144, New York, NY, USA, 2005. ACM.
30. Corneliu Popeea and Wei-Ngan Chin. Dual analysis for proving safety and finding bugs. In *ACM Symposium on Applied Computing*, pages 2137–2143. ACM, 2010.
31. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
32. Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *TAP*, pages 154–170, Berlin, Heidelberg, 2008. Springer-Verlag.