A Resource-Based Logic for Termination and Non-Termination Proofs (Technical Report)

Ton Chanh Le[1], Cristian Gherghina[2], Aquinas Hobor[1], and Wei-Ngan Chin[1]

[1] Department of Computer Science, National University of Singapore
[2] Singapore University of Technology and Design
{chanhle, hobor, chinwn}@comp.nus.edu.sg cristian_gherghina@sutd.edu.sg

Abstract. We propose a unified logical framework for specifying and proving *both* termination and non-termination of various programs. Our framework is based on a resource logic which captures both upper *and* lower bounds on resources used by the programs. By an abstraction, we evolve this resource logic for execution length into a temporal logic with three predicates to reason about termination, non-termination or unknown. We introduce a new logical entailment system for temporal constraints and show how Hoare logic can be seamlessly used to prove termination and non-termination in our unified framework. Though this paper's focus is on the formal foundations for a new unified framework, we also report on the usability and practicality of our approach by specifying and verifying both termination and non-termination properties for about 300 programs, collected from a variety of sources. This adds a modest 5-10% verification overhead when compared to underlying partial-correctness verification system.

1 Introduction

Termination proving is an important part of correctness proofs for software systems as "so-called *partial correctness* is inadequate: if a program is intended to terminate, that fact must be part of its specification." – Cliff Jones [28]. Thus, *total correctness* proofs, denoted by the Hoare triple $[P]c[Q]$, require the code fragment c to be shown terminating in addition to meeting the postcondition Q after execution. The termination of a loop or a recursive method is usually proven by a well-founded termination measure given to the specification. However, such a measure is not a component of the logical formulas for pre/post specifications. A reason for this distinction is that specification logic typically describes program states, while the termination proofs are concerned with the existence of well-founded measures to bound the execution length of loops/recursions, as argued by Hehner in [25]. Due to this distinction, we cannot automatically leverage richer logics that have been developed for safety properties to conduct more intricate termination and non-termination reasoning.

For illustration, let us use the Shuffle problem proposed in the Java Bytecode Recursive category of the annual Termination Competition [34]. In this problem, an acyclic linked list is shuffled by the `shuffle` method together with the auxiliary `reverse` method, whose source code is shown in Fig. 1. To prove that `shuffle` terminates, we

```
public static List shuffle(List xs) {
  if (xs == null) return null;
  else {
    List next = xs.next;
    return new List(xs.value,
          shuffle(reverse(next)));
  }
}
```

```
public static List reverse(final List l) {
  if (l == null || l.next == null)
    return l;
  final List nextItem = l.next;
  final List reverseRest =
                  reverse(nextItem);
  l.next = null;  nextItem.next = l;
  return reverseRest; }
```

Fig. 1. The Shuffle problem from the Termination Competition

need to firstly show that `reverse` also terminates. While the termination of `reverse` can be easily proved by current approaches, such as [31,8,12], proving `shuffle` terminates is harder because it requires a functional correctness related fact: the `reverse` method does not change the length of the list. Based on this fact, it is possible to show that the linked list's length is also decreasing across the recursive method call `shuffle`; as a result, the method always terminates.

Therefore, without an integration of termination specification into logics for functional correctness, such as separation logic [37], the termination of `shuffle` is hardly specified and proved by verification systems based on the traditional Hoare logic for total correctness. Note that automated termination provers, such as AProVE [21] and COSTA [3], are not able to show that `shuffle` terminates, even after applying a numeric abstraction on the size property to `shuffle` [33], due to the lack of information flow between the correctness and the termination arguments. We believe that relatively complex problems, such as Shuffle, highlight the need of a more expressive logic with the ability of integration into various safety logics for termination reasoning.

Moreover, if the termination proof fails, *e.g.*, when the input list of `shuffle` is cyclic, the program will be *implicitly* assumed to be *possibly* non-terminating. That is, *definite* non-termination is neither *explicitly* stated nor proven by Hoare logic. Explicitly proving non-termination has two benefits. First, it allows more comprehensive specifications to be developed for better program understanding. Second, it allows a clearer distinction between expected non-termination (*e.g.*, reactive systems where loops are designed to be infinite) and failure of termination proofs, paving the way for focusing on real non-termination bugs that minimize on false positives.

Some specification languages, such as Dafny [32], ACSL [7] and JML [30], allow the specification of possible non-termination but their verifiers provide limited support for this feature. For example, the Dafny verifier only allows this specification on loops or tail-recursive methods[1] while Frama-C [16] for ACSL has not implemented it. On the other hand, we can use the `false` postcondition, which indicates that the method's exit is unreachable, to specify definite non-termination. However, such postcondition for partial correctness is not preferred as it is usually considered too strong for functional properties and distinct from termination proofs. This distinction has been designed into Dafny, Frama-C and KeY with JML [2], that makes the tools fail to take into account

[1] http://www.rise4fun.com/Dafny/PnRX

non-terminating behavior when proving termination. For example, Dafny succeeds in proving the termination of a recursive method[2] though this method contains a call to a non-terminating method.[3] In fact, for termination proofs, these tools simply check that there is a finite number of recursive calls to the analyzed methods, rather than the methods' termination per se.

Our proposal. We propose integrating both termination and non-termination requirements directly into the specification logic for functional properties. Our work follows Hoare and He [26] and Hehner [24], in which the termination is reasoned together with partial correctness proof. In [24], the program is instrumented with a time variable t and the termination is proven by a finite bound on the *exact* execution time $t'-t$, where t, t' are the initial, resp. final time. In [26], a special ghost variable ok is used to signify termination. However, these approaches presently do not handle non-termination.

As a formal foundation to unify termination and non-termination reasoning and integrate them into functional correctness proofs, we introduce a new resource logic which captures the concept of resource capacity; tracking both *minimum and maximum* amounts of resources used by some given code. Our logic uses a primitive predicate $\texttt{RC}\langle l, u\rangle$ with invariant $0{\leq}l{\leq}u$ to capture a semantic notion of resource capacity (l, u) with the lower bound l and the upper bound u. Through this resource logic, we can specify a variety of complexity-related properties, including the notions of termination and non-termination, by tracking the number of calls (and loop iterations) executed by the given code. Termination is denoted by the presence of a *finite* upper bound, while non-termination is denoted an *infinite* lower bound on the execution length.

To support a more effective mechanism, we shall derive a simpler *temporal* logic from the richer resource logic itself. We define three temporal predicates, $\texttt{Term}\ M$, $\texttt{Loop}$ and $\texttt{MayLoop}$, where M is a well-founded termination measure, and associate them with each method in a given program to denote the termination, definite non-termination and possible non-termination of these methods, respectively. In terms of resource reasoning, these predicates represent $\texttt{RC}\langle 0, \texttt{embed}(M)\rangle$, $\texttt{RC}\langle\infty, \infty\rangle$ and $\texttt{RC}\langle 0, \infty\rangle$, respectively, where $\texttt{embed}(M)$ is a finite bound obtained through an order-embedding of M into naturals. Using the enriched specification logic, functional correctness, termination and non-termination of methods can be verified under a single modular framework. With this unification, the predicate $\texttt{Term}\ M$ denotes exactly definite termination, instead of just denoting the bound on the number of loop iterations or method recursions like the termination measures used in the traditional Hoare logic for total correctness.

Our research contributions can be summarized as follows:

- A new *resource logic* that can capture lower and upper bounds on resource usage via the concept of resource capacity, together with an entailment procedure to support correctness proofs with resource-related properties. (Sec. 3)
- A *temporal logic* that is abstracted from the resource logic to reason about both program termination and non-termination. We introduce three new temporal constraints, its *entailment* and *Hoare rules* lifted from the resource logic. (Sec. 4)

[2] http://www.rise4fun.com/Dafny/6FuR

[3] The example in ACSL and JML are at http://loris-7.ddns.comp.nus.edu.sg/~project/hiptnt/others.zip

```
pred isEvenNat(int n) ≡ n≥0 ∧ ∃m · n = 2*m;
int sumE (int n)
  requires isEvenNat(n) ∧ Term[n] ∨
           ¬isEvenNat(n) ∧ Loop
  ensures true;
{ if (n==0) return 0;
  else return n + sumE(n−2); }
```
(a)

```
while (x>y)
  requires
    x≤y ∧ Term[] ∨
    x>y ∧ x<0 ∧ Loop ∨
    x>y ∧ x≥0 ∧ MayLoop
  ensures x′≤y′;
{ y=x+y; x=x−1; }
```
(b)

Fig. 2. Examples on numerical programs

– A successful integration of both resource and temporal logics into a separation logic based verifier [36]. The new temporal logic is expressive enough to specify and successfully verify the (non-)termination behaviors for about 300 benchmark programs collected from a variety of sources, including the SIR/Siemens test suite [18] and problems from the Termination Competition (Sec. 5). The prototype implementation and benchmark are available for online use and download at:

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

2 From Resource to Temporal Logic

We introduce a general resource predicate $\texttt{RC}\langle l, u\rangle$ where l is a lower bound and u is an upper bound on resource capacity, with invariant $0{\leq}l{\leq}u$. This resource predicate can be specialized to execution capacity to capture a variety of complexity-related properties, via lower and upper bounds on the total number of method calls during the execution of a given piece of code. We shall give an instrumented semantics for this specific resource logic, and also specialize it for reasoning about termination and non-termination. To prove termination, we simply use the predicate $\texttt{RC}\langle 0, u\rangle$ where u is some finite value, namely $u{<}\infty$. To prove non-termination, we can use the predicate $\texttt{RC}\langle\infty, \infty\rangle$ which signifies an infinite lower bound. Lastly, if we cannot prove either termination or non-termination, we use the predicate $\texttt{RC}\langle 0, \infty\rangle$ which covers all possibilities.

The resource logic we have outlined is quite expressive, and could moreover be specialized for reasoning on just termination and non-termination with the direct handling of infinity ∞ value. In order to design a simpler logic, we introduce a temporal logic with three distinct predicates, as follows: (i) `Term` M to denote $\texttt{RC}\langle 0, \texttt{embed}(M)\rangle$, (ii) `Loop` to denote $\texttt{RC}\langle\infty, \infty\rangle$ and (iii) `MayLoop` to denote $\texttt{RC}\langle 0, \infty\rangle$. Such a temporal logic is considerably simpler than the more expressive resource logic, since we can omit reasoning with ∞. We can also use a simpler termination measure M, based on depth of recursion rather than number of calls, but relate to the latter using $\texttt{embed}(M)$. Moreover, these temporal predicates can be made flow-insensitive, and thus need only appear in each method's precondition where they describe execution capacity required for the method's execution. This two-level approach simplifies both the design of a formal semantics, and the development of a verification framework for (non-)termination.

For illustration, let us look at some numerical examples, starting with the method `sumE` in Fig. $2(a)$. This method is required to return the sum of all even natural num-

bers that are less than or equal to the input n. However, the implementation satisfies this requirement only when n is an even natural number, denoted by the predicate isEvenNat(n); otherwise, the method does not terminate[4]. In our approach, these distinct scenarios can be described in a termination-enriched specification by seamlessly integrating the temporal constraints Term[n] and Loop into a logic with disjunctions.

JML and ACSL also support the specification of several method behaviors. However, the current ACSL implementation in Frama-C does not allow fine-grained termination related specification of each behavior and ignores conditional termination clauses. As a result, it cannot verify all the (non-)terminating behaviors of sumE together. KeY allows the specification of termination for each individual method behavior but it cannot disprove the termination of sumE when n is an odd positive number, because the variant n is still valid under this precondition. In contrast, our unified termination and non-termination reasoning does not accept the temporal constraint Term[n] in these prestates because the execution starting from them will eventually reach a non-terminating execution when $n<0$. In terms of resource reasoning, Term[n], denoting a finite resource, is invalid as it cannot satisfy the infinite resource required by the non-termination.

The next example in Fig. $2(b)$ illustrates a usage of MayLoop constraint. Starting from any prestate satisfying $x>y \wedge x\geq 0$, the execution of the given loop may reach either the base case (when $x\leq y$, indicated by Term[]) or the non-terminating case (when $x>y \wedge x<0$, indicated by Loop). We observe that this MayLoop precondition can be strengthened to the *non-linear* constraint $4x^2+4x+8y+9\geq 0$ for non-termination, but this requires stronger arithmetic solvers.

Though our proposal is independent of the underlying logics on functional properties, it can leverage infrastructures of richer logics[5] to conduct termination and non-termination reasoning for more complex domains. For example, our proposed temporal constraints are easily integrated into formulas of separation logic to reason about the termination and non-termination of heap-based programs. We choose a fragment of separation logic with the separating conjunction $*$ and the points-to operator $\mapsto$ to specify the heap assertions. These operators are used to describe several data structures, such as linked list and tree. For example, the inductive predicate $\texttt{lseg}(\texttt{root}, \texttt{p}, \texttt{n})$ declared in Fig. $3(a)$ describes a list segment size of n from root to p with an invariant property stated that the list's size is non-negative. This predicate can be used to specify either null-terminating lists (when $\texttt{p} = \texttt{null}$) or circular lists (when $\texttt{p} = \texttt{root}$).

We then use the predicate lseg for the pre and postconditions of two methods reverse and shuffle in the Shuffle problem. The specification of each method indicates that the method's result res is a linked list with the same size n as the input list. From these safety specifications, the temporal constraint Term[n] integrated into the precondition of each method is able to specify that the depth of recursion is bounded by the size of the input list, thus indicating the method's termination.

From the perspective of resource reasoning, a temporal constraint in the precondition of a method defines the bounds of available resource allowed for program executions from prestates satisfying (safety part of) this precondition. This idea is similar to

[4] The verification system assumes the use of arbitrary precision integers. When finite integers are used, we may give a different temporal specification for those prestates.

[5] In comparison with the first-order logic with linear arithmetic for numerical programs.

data List { int value; List next; } **pred** lseg(root, p, n) ≡ root=p ∧ n=0 ∨ ∃v, q · root↦List(v, q)∗lseg(q, p, n−1) **inv** n ≥ 0; (a)	List reverse (List l) requires lseg(l, null, n) ∧ Term[n] ensures lseg(res, null, n); List shuffle (List xs) requires lseg(xs, null, n) ∧ Term[n] ensures lseg(res, null, n); (b)

Fig. 3. A specification in separation logic to verify the correctness of Shuffle's methods

Atkey's logic [6], a type-based amortized resource analysis for imperative programs, which associates a piece of resource with each element of the data structures prior program execution. However, Atkey's approach only tracks the upper bound of resource usage, so that it cannot reason about non-termination. This shortcoming also applies to other type-based approaches for termination reasoning, such as [1,39]. In addition, while the amortized resource analysis accounts for individual time-step (or heap chunk), we use termination measures, which are much simpler, to facilitate termination proofs. For example, to analyze `shuffle`, Atkey's logic requires the global length property to present the polynomial resource associated with the input list using the technique of Hoffmann and Hofmann [27], which is much harder than locally reasoning about each node of the list as stated in his paper. Finally, this logic is built on top of just separation logic, rather than being generic as our proposal.

3 A Logic for Resource Reasoning

In proving termination and non-termination, our goal is to use resource reasoning based on execution capacity to provide a means for quantitatively assessing the execution length of a program. For this purpose, we introduce a resource logic to formally assess the minimum and a maximum bounds on a program's resource consumption. We first extend the program state model with a mechanism to track resource capacities of the underlying machine. Since the particular consumed resource is countable and possibly infinite, we use the set $\mathbb{N}^\infty$, short for $\mathbb{N} \cup \{\infty\}$, as its domain.

3.1 Resource Capacity

Definition 1 (Program states) *A program state σ is a triple (s, h, r) of stack $s \in \mathcal{S}$ (locals), heap $h \in \mathcal{H}$ (memory) and $r \in \mathcal{R}$, resource capacity where r is a pair (r_l, r_u) of bounds in $\mathbb{N}^\infty$, with $0 \leq r_l \leq r_u$, denoting the allowed minimum and maximum resource consumption for executions starting from the current program state.*

Intuitively, a program state's resource capacity (r_l, r_u) ensures that any execution starting from this state must consume *at least* r_l and *at most* r_u of the tracked resource.

Definition 2 (Resource Capacity Ordering) *Let $(\leq_c) \subset \mathbb{N}^\infty \times \mathbb{N}^\infty$ be the resource capacity ordering, such that $(b_l, b_u) \leq_c (a_l, a_u)$ iff $a_l \leq b_l$ and $b_u \leq a_u$.*

$(s,h,r) \models \Psi_1 \vee \Psi_2$	$\equiv$	$(s,h,r) \models \Psi_1$ or $(s,h,r) \models \Psi_2$
$(s,h,r) \models \Psi_1 \wedge \Psi_2$	$\equiv$	$(s,h,r) \models \Psi_1$ and $(s,h,r) \models \Psi_2$
$(s,h,r) \models \exists x_i^* \cdot \Psi$	$\equiv$	$\exists \nu_i^* \cdot (s[(x_i \mapsto \nu_i)^*], h, r) \models \Psi$
$(s,h,r) \models \mu$	$\equiv$	$(s,h) \models \mu$
$(s,h,r) \models \texttt{RC}\langle a_l, a_u \rangle$	$\equiv$	$(s,h) \models r_l = a_l \wedge r_u = a_u$ where $r{=}(r_l, r_u)$
$(s,h,r) \models \rho_1 \blacktriangleright \rho_2$	$\equiv$	$\forall r' \cdot$if $(s,h,r') \models \rho_1$ then $(s,h,r \ominus r') \models \rho_2$

Fig. 5. Semantics of Assertions in the Resource-Aware Logic

The resource capacity (a_l, a_u) is considered larger (or more general) than (b_l, b_u) if $a_l \leq b_l$ and $b_u \leq a_u$. The intuition is that under this condition, any execution which guarantees the capacity (b_l, b_u) also guarantees the capacity (a_l, a_u). Based on this observation, $(0, \infty)$ is the largest resource capacity. In fact, it indicates an unconstrained resource consumption.

In order to properly define an operational semantics in terms of the proposed program state model, we also need to be able to express resource consumption. To this end we define a splitting operation over the resource capacity. We will say that a capacity (a_l, a_u) can be split into capacities (b_l, b_u) and (c_l, c_u), written $(a_l, a_u) \ominus (b_l, b_u) = (c_l, c_u)$, if whenever an execution that guarantees the capacity (b_l, b_u) starts from a state with the capacity (a_l, a_u) then the remaining capacity is (c_l, c_u). In other words, the executions allowed by (a_l, a_u) can be decomposed into executions required by (b_l, b_u) followed by executions required by (c_l, c_u).

Definition 3 (Resource Capacity Splitting) *Given resource capacities* (a_l, a_u),(b_l, b_u) *with* $b_u \leq a_u$ *and* $a_l + b_u \leq a_u + b_l$ *then* $(a_l, a_u) \ominus (b_l, b_u) = (c_l, c_u)$ *where*

$$c_l = \texttt{min}\{x_l \in \mathbb{N}^\infty \mid x_l + b_l \geq a_l\} \text{ and } c_u = \texttt{max}\{x_u \in \mathbb{N}^\infty \mid x_u + b_u \leq a_u\}.$$

From Defn. 3, (c_l, c_u) is the *largest* resource consumption allowed for any execution following executions satisfying (b_l, b_u) such that the overall resource consumption is described by (a_l, a_u). Under this interpretation it follows naturally that when $b_u > a_u$ the splitting operation is undefined as c_u does not exist. In addition, when $a_l + b_u > a_u + b_l$, the splitting operation is also undefined as it would lead to $c_l > c_u$.

3.2 Assertion Language and Semantics for a Resource-Aware Logic

To support resource reasoning, we extend a minimalistic assertion language with two resource assertions, as shown in Fig. 4. We use v and v^* for variables and sequences of variables, $f(v^*)$ for functions from variables to $\mathbb{N}^\infty$, μ and Φ to represent resource-free formulas and ρ for resource assertions.

The resource assertion ρ ranges over (i) *atomic* resource assertions $\texttt{RC}\langle a_l, a_u \rangle$, where a_l, a_u are functions from variables to $\mathbb{N}^\infty$; and (ii) *splitting* resource assertions $\rho_1 \blacktriangleright \rho_2$, which holds for states that allow executions to be split into two execution fragments, on which ρ_1 and ρ_2 hold respectively.

$$
\begin{array}{l}
\Psi ::= \bigvee(\exists v^* \cdot \mu \wedge \rho)^* \\
\Phi ::= \bigvee(\exists v^* \cdot \mu)^* \\
\rho \ ::= \mathtt{RC}\langle a_l, a_u\rangle \mid \rho_1 \blacktriangleright \rho_2 \\
a \ ::= f(v^*)
\end{array}
$$

Fig. 4. The Assertion Language

We concisely list in Fig. 5 the semantic model for the assertion language. We observe that the usual semantics of the logical connectives, *e.g.*, conjunctions and disjunctions, lifts naturally over resource assertions. The semantics of the resource-free assertions is straightforward: a resource-free formula μ holds for all states (s, h, r) such that $(s, h) \models \mu$ with respect to the semantics of the corresponding underlying logic.

We point out that we have chosen to model the $\mathtt{RC}\langle a_l, a_u\rangle$ assertion as a precise predicate. That is, a program state σ satisfies a resource constraint ρ if the resource capacity in σ is equal to the evaluation, in the context of σ, of the upper and lower functions associated with ρ. This modeling relation ensures that the resource assertion ρ is *precise* with regards to the resource capacity, where $(s, h, r) \models \rho$ does not imply $(s, h, r') \models \rho$ whenever r' is larger than r, *i.e.*, $r' \geq_c r$. Consequently, $\mathtt{RC}\langle a_l, a_u\rangle \vdash \mathtt{RC}\langle b_l, b_u\rangle$ iff $(s, h) \models a_l{=}b_l \wedge a_u{=}b_u$. Additionally, $\mathtt{RC}\langle a_l, a_u\rangle \wedge \mathtt{RC}\langle b_l, b_u\rangle \equiv \mathtt{RC}\langle a_l, a_u\rangle$ iff $a_l{=}b_l \wedge a_u{=}b_u$; otherwise, $\mathtt{RC}\langle a_l, a_u\rangle \wedge \mathtt{RC}\langle b_l, b_u\rangle \equiv \mathtt{false}$.

To provide a precise modular resource reasoning, we lift the semantic split operation into a resource splitting assertion $\rho_1 \blacktriangleright \rho_2$. This enables our proof construction to follow the same style of other resource manipulating logics, such as separation logic. The intuition behind the splitting resource assertions is that $\rho_1 \blacktriangleright \rho_2$ holds for any program state from which it is possible to consume as many resources as ρ_1 requires and end in a state that satisfies ρ_2. Or equivalently, $\rho_1 \blacktriangleright \rho_2$ holds for all states whose resource capacity can be split into two portions, such that the resulting capacities satisfy ρ_1 and ρ_2, respectively. In addition, we can use $\blacktriangleright$ to add a resource capacity ρ_1 into the current available resource capacity ρ, resulting in $\rho \blacktriangleright \rho_1$. The semantics of $\rho_1 \blacktriangleright \rho_2$ is also given in Fig. 5.

3.3 Resource-Enhanced Entailment with Frame Inference

Based on the semantics of resource assertions and the standard definition of the logical entailment relation (*i.e.*, $\Psi_1 \vdash \Psi_2$ iff $\forall \sigma \cdot$ if $\sigma \models \Psi_1$ then $\sigma \models \Psi_2$), it is possible to define an entailment for resource constraints of the form $\rho \vdash \rho_1 \blacktriangleright \rho_2$ as follows:

Lemma 1 (Resource Entailments). *Given resource assertions ρ, ρ_1 and ρ_2, $\rho \vdash \rho_1 \blacktriangleright \rho_2$ iff $\forall s, h, r, r_1 \cdot$ if $(s, h, r) \models \rho$ and $(s, h, r_1) \models \rho_1$ then $(s, h, r \ominus r_1) \models \rho_2$.*

Proof. We have

$$
\begin{array}{ll}
& \rho \vdash \rho_1 \blacktriangleright \rho_2 \\
\equiv & \forall s, h, r \cdot \text{ if } (s, h, r) \models \rho \text{ then } (s, h, r) \models \rho_1 \blacktriangleright \rho_2 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(Defn. of logical entailment)} \\
\equiv & \forall s, h, r \cdot \text{ if } (s, h, r) \models \rho \text{ then } \forall r_1 \cdot \text{ if } (s, h, r_1) \models \rho_1 \text{ then } (s, h, r \ominus r_1) \models \rho_2 \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{(Semantics of } \rho_1 \blacktriangleright \rho_2\text{)} \\
\equiv & \forall s, h, r, r_1 \cdot \text{ if } (s, h, r) \models \rho \text{ and } (s, h, r_1) \models \rho_1 \text{ then } (s, h, r \ominus r_1) \models \rho_2 \ \square
\end{array}
$$

It follows that given $\ominus_f$, a lifting of resource capacity splitting to functions, then:

$$
\begin{array}{ll}
\mathit{Prog} ::= \mathit{tdecl}^*\ \ \mathit{meth}^* & \Upsilon ::= \texttt{requires}\ \Psi\ \Upsilon \mid \texttt{ensures}\ \Psi \mid \texttt{ensures}\ \Phi \\
\mathit{tdecl} ::= \texttt{data}\ c\ \{(\mathit{type}\ v)^*\} & e ::= \texttt{null} \mid k \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \\
\mathit{type} ::= c \mid \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \texttt{void} & \quad\ \texttt{new}\ c(v^*) \mid e_1; e_2 \mid \mathit{type}\ v; e \mid mn(v^*) \mid \\
\mathit{meth} ::= \mathit{type}\ mn((\texttt{[ref]}\ \mathit{type}\ v)^*)\ \Upsilon\ \{e\} & \quad\ \texttt{return}\ v \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

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

Fig. 6. A Core Imperative Language with Specifications

$$\frac{(\rho_l^2, \rho_u^2) = (\rho_l, \rho_u) \ominus_f (\rho_l^1, \rho_u^1)}{\texttt{RC}\langle \rho_l, \rho_u \rangle \vdash \texttt{RC}\langle \rho_l^1, \rho_u^1 \rangle \blacktriangleright \texttt{RC}\langle \rho_l^2, \rho_u^2 \rangle}$$

Entailments of the form $\rho \vdash \rho_1 \blacktriangleright \rho_2$ are of particular interest in the context of program verification as they naturally encode the restriction imposed at a method call and the remaining restriction after the execution of this method. For the proposed resource logic, we construct a general entailment system with frame inference by merging the entailment of resource constraints presented earlier with the entailment system corresponding to the underlying logic. Let the underlying entailment system be of the general form $\Psi \vdash \Phi \rightsquigarrow \Phi_r$ denoting that Ψ implies Φ with frame Φ_r. In sub-structural logics such as separation logic, the frame captures any residual state that is not required by the entailment. In pure logics where the program states are not changed, the frame is simply the antecedent of the entailment.

To support logics with disjunctions, the entailment system firstly deconstructs disjunctive antecedents (*e.g.*, using the rule $\boxed{\textbf{ENT}-\textbf{DISJ}-\textbf{LHS}}$) and consequents until formulas of the form $\mu \wedge \rho$ with a single resource constraint[6] are encountered in both sides of the sub-entailments. The judgment system then applies the rule $\boxed{\textbf{ENT}-\textbf{CONJ}}$ that is slightly changed to handle resource constraints by splitting an entailment into two parts, namely *logical* part and *resource* part. The logical goal is solved by the entailment system $\mu_a \vdash \mu_c \rightsquigarrow \Phi_r$ of the underlying logic. The resource goal is solved by using the resource entailment rules presented above. The solving process for the resource part leverages the entailment outcome Φ_r from the underlying logic, which is simply added to the antecedent of the resource entailment, to check the condition stated in Defn. 3 for the resource capacity splitting operation to be defined. -2mm

$$\frac{\begin{array}{c}\Psi\ =\ \bigvee \exists v_i^* \cdot (\mu_i \wedge \rho_i) \\ \forall i \cdot\ (\mu_i \wedge \rho_i) \vdash \Phi \rightsquigarrow \Psi_r^i\end{array}}{\Psi \vdash \Phi \rightsquigarrow \bigvee \exists v_i^* \cdot \Psi_r^i}\ [\textbf{ENT}-\textbf{DISJ}-\textbf{LHS}] \qquad \frac{\begin{array}{c}\mu_a \vdash \mu_c \rightsquigarrow \Phi_r \\ \mu_a \wedge \Phi_r \wedge \rho_a \vdash \rho_c \blacktriangleright \rho_r\end{array}}{\mu_a \wedge \rho_a \vdash \mu_c \wedge \rho_c \rightsquigarrow (\Phi_r \wedge \rho_r)}\ [\textbf{ENT}-\textbf{CONJ}]$$

3.4 Hoare Logic for Resource Verification

Language. We provide a core strict imperative language with usual constructs, such as type declarations, method declarations, method calls, assignments, etc. in Fig. 6 to facilitate the verification for multiple front-end imperative languages. For simplicity,

[6] A conjunction of resource constraints can be simplified to either a single resource constraint or

we choose a core language without while-loop constructs and assume a preprocessing step that applies an automatic translation into tail-recursive methods with reference-type parameters (declared by the keyword `ref`).

The pre and post conditions of a method are specified by the `requires` and `ensures` keywords, followed by logic formulas in the assertion language in Fig. 4. Resource-related assertions always appear in the method preconditions to denote resource requirements imposed on the caller for its execution. In contrast, resource assertions in the postconditions denote unspent/generated fuel returned to the caller, so that these assertions may not appear in the postconditions, depending on the analyzed resource. For example, as execution length (*i.e.*, a temporal resource) can only be consumed, it is safe and convenient to assume that the method consumes all the initially required resource; thus we can avoid the need for execution length related assertions in postconditions.

Hoare Logic. We observe that the resource consumption of each program statement is dependent on the tracked resource. As a result, the resource-aware Hoare logic needs to be adapted accordingly for each resource type. In terms of termination and non-termination reasoning, we are interested in the execution length as the tracked resource capacity. In the next section, we will construct a specific Hoare logic to reason about this resource.

4 (Non-)Termination Proofs via Resource Reasoning

For termination and non-termination reasoning, we have proposed three temporal constraints to capture: guaranteed termination $\texttt{Term}\ X$, guaranteed non-termination `Loop` and possible non-termination `MayLoop`, where X is a ranking function built from program variables. First, we define these constraints as resource capacity assertions, using the more general `RC` predicate. Next, we leverage the resource logic in Sec. 3, specialized in execution capacity, to construct a logic for termination and non-termination reasoning.

A resource-based definition for the proposed temporal constraints is as follows:

Definition 4 (Temporal Constraints) *Temporal constraints are resource assertions over program execution lengths, such that* $\texttt{Term}\ X \equiv \texttt{RC}\langle 0_f, \varpi\rangle$, $\texttt{Loop} \equiv \texttt{RC}\langle \infty_f, \infty_f\rangle$ *and* $\texttt{MayLoop} \equiv \texttt{RC}\langle 0_f, \infty_f\rangle$ *where* 0_f *and* ∞_f *denote the constant functions always returning* 0 *respectively* ∞. ϖ *is a function of program variables to naturals, imposing a finite upper bound on the execution length of a terminating program.*

Using the definition of resource entailments in Lemma 1, we formalize the set of valid entailments for temporal constraints below:

$$\texttt{MayLoop} \vdash \texttt{MayLoop} \blacktriangleright \texttt{MayLoop} \qquad \texttt{Loop} \vdash \texttt{Term}\ X \blacktriangleright \texttt{Loop}$$

$$\texttt{MayLoop} \vdash \texttt{Term}\ X \blacktriangleright \texttt{MayLoop} \qquad \texttt{Loop} \vdash \texttt{Loop} \blacktriangleright \texttt{MayLoop}$$

$$\texttt{MayLoop} \vdash \texttt{Loop} \blacktriangleright \texttt{MayLoop}$$

$$\texttt{Loop} \vdash \texttt{MayLoop} \blacktriangleright \texttt{Loop}$$

$$\frac{\mu \Longrightarrow Y \leq_d X}{\mu \wedge \texttt{Term}\ X \vdash \texttt{Term}\ Y \blacktriangleright \texttt{Term}\ X -_d Y}$$

`false` as discussed in Sec. 3.2.

where $\leq_d$ and $-_d$ are the ordering and the subtraction operation on the domain of the termination measures X and Y, respectively. All other decomposition attempts, such as $\texttt{Term}\ X \vdash \texttt{MayLoop} \blacktriangleright \_$ and $\texttt{Term}\ X \vdash \texttt{Loop} \blacktriangleright \_$, describe unfeasible splits. Thus in those cases, the entailment fails and an error is signaled.

4.1 From Termination Measures to Execution Capacity's Finite Upper Bounds

In Defn. 4, as X denotes a termination measure, a bounded function that decreases across recursive method calls, the resource upper bound ϖ must also follow. Thus, the mapping function from X to ϖ must be an *order-embedding* denoted by $\texttt{embed}(X)$. In our approach, the termination measure X is a list of arithmetic formulas over naturals since this formulation is simpler to write than a single but more complex termination measure and it can be used for a wider range of programs. In general, an order-embedding of lists of unbounded elements requires *ordinals*. However, transfinite ordinals are not suitable to model finite computational resources denoted by $\texttt{Term}\ X$.

By a *co-inductive* argument that every execution of a terminating method only computes finitely many different values, it follows that every non-negative element of a lexicographic termination measure applied to states of the corresponding call tree is upper-bounded. We then show that there always exists an order-embedding $\mathcal{L}$ from the codomain of a termination measure (*i.e.*, tuples of bounded naturals) to naturals, such that $\texttt{embed}(X) = \mathcal{L} \circ X$.

Lemma 2. *If the termination of a program can be proven by a given lexicographic termination measure, then for each call tree τ of the program, every element of the termination measure applied to the program states corresponding to the nodes in the call tree τ is bounded.*

Proof. As the program can be proven to terminate by the measure $[X_n, X_{n-1}, \ldots, X_0]$, the corresponding evaluation call tree is finite. The reason is that if the call tree is infinite then by König's lemma [29], there is an infinite evaluation path, which means that the program is non-terminating.

Let S_i be the set of evaluating values of X_i and N be the finite number of the tree's nodes. Then, for all i, the cardinality $|S_i| \leq N$ or S_i is finite. As a result, the set $S = \bigcup S_i$ is also finite. The maximum value k of S is the upper bound of every element of the given termination measure. □

If every element x_i, where $0 \leq i \leq n-1$, of a lexicographic termination measure $[x_n, x_{n-1}, \ldots, x_0]$ corresponding to a given call tree τ is bounded by a constant k, we can use the base $b = k{+}1$ to construct a possible order-embedding function $\mathcal{D}([x_n, x_{n-1}, \ldots, x_0]) = x_n * b^n + x_{n-1} * b^{n-1} + \ldots + x_0$. The function $\mathcal{D}$ preserves the order of the given measure along every trace of τ, as stated by Lemma 3.

Lemma 3. *For all $x_n, \ldots, x_0, y_n, \ldots, y_0 \in \mathbb{N}$ such that $\forall i \in \{0..n-1\} \cdot x_i, y_i < b$, $[x_n, \ldots, x_0] >_l [y_n, \ldots, y_0]$ iff $\mathcal{D}([x_n, \ldots, x_0]) > \mathcal{D}([y_n, \ldots, y_0])$, where $>_l$ is the lexicographic ordering.*

Proof. ($\Rightarrow$) From the premise, we have $\exists i{\in}\{0..n\} \cdot x_n{=}y_n \wedge \ldots \wedge x_{i+1}{=}y_{i+1} \wedge x_i{>}y_i$. Consequently, $x_i{-}y_i{\geq}1$. Moreover, because $\forall i \cdot 0{\leq}x_i, y_i{<}b$, we also have $1{-}b \leq x_i{-}y_i \leq b{-}1$. Let consider

$$\begin{aligned}&\mathcal{D}([x_n, x_{n-1}, \ldots, x_0]) - \mathcal{D}([y_n, y_{n-1}, \ldots, y_0])\\ &= (x_i - y_i) * b^i + (x_{i-1} - y_{i-1}) * b^{i-1} + \ldots + (x_0 - y_0)\\ &\geq b^i + (1-b) * (b^{i-1} + \ldots + 1) = b^i + (1 - b^i) = 1 > 0\end{aligned}$$

Thus, $\mathcal{D}([x_n, \ldots, x_0]) > \mathcal{D}([y_n, \ldots, y_0])$.

($\Leftarrow$) By contradiction, assume that $[x_n, \ldots, x_0] <_l [y_n, \ldots, y_0]$. Similarly to the above proof, we have $\mathcal{D}([y_n, \ldots, y_0]) > \mathcal{D}([x_n, \ldots, x_0])$, which is a contradiction. Moreover, if $\forall i{\cdot}x_i{=}y_i$ then $\mathcal{D}([x_n, \ldots, x_0]){=}\mathcal{D}([y_n, \ldots, y_0])$. As a result, $[x_n, \ldots, x_0] >_l [y_n, \ldots, y_0]$. □

In general, such a bounded constant k for a call tree τ can be determined by a function $\mathcal{K}$ of initial values of the call tree's variables. For example, the constant k for the loop in Fig. 2(b) is

$$k = \mathcal{K}(x_0, y_0) = \begin{cases} \texttt{max}(1 - x_0, y_0 - \frac{x_0^2}{2} + \frac{x_0}{2} + 1) & \text{, if } x_0 \leq 1 \\ x_0 - y_0 & \text{, if } x_0 > 1 \end{cases}$$

which is the maximum value of the expression $r{=}\{1{-}x, \text{if } x \leq 1;\ x{-}y, \text{if } x > 1\}$, applied to the execution trace starting with symbolic initial values x_0 and y_0. The corresponding order-embedding of the loop is $\mathcal{D}([m, n]) = m * (k + 1) + n$.

For a loop, the order-embedding $\mathcal{D}$ would be enough to ensure the sufficiency of execution capacity because the loop execution has only one trace. In order to give a proper estimate of the execution capacity for more complex recursion patterns, especially when the termination measures are based on the depth of recursion, we propose using a more refined embedding for a call tree, that is $\mathcal{L} = \begin{cases} \mathcal{D} & ,\mathcal{N} \leq 1 \\ \mathcal{N}^{\mathcal{D}} & ,\mathcal{N} > 1 \end{cases}$, where $\mathcal{N}$ is the maximum number of children for each node of the call tree.

Therefore, given the termination measure X of a terminating program, there always exists an order-embedding $\mathcal{L}$ from the codomain of X to naturals. The function $\mathcal{L}$ can be constructed from initial values of program variables and the call trees corresponding to these initial values. As a result, $\texttt{embed}(X) = \mathcal{L}{\circ}X$ is a function from program variables to naturals, which describes an upper bound on the number of method calls taken by any execution of the program.

4.2 Termination and Non-Termination Verification

Here we elaborate on the construction of both termination and non-termination proofs based on Defn. 4 and the verification framework in Fig. 7 for tracking execution length as resource. Although execution length can be tracked at various levels of granularities, we choose to track it only at method calls (*i.e.*, as the total number of method calls) in order to simplify the verification rules and the operational semantics. In Fig. 7, we only outline the Hoare logic rules for the method call and the return statements, which are especially relevant to the verification of execution lengths as they encode the resource consumption. The Hoare rules for other constructs are standard because they do not interact with the resource of interest.

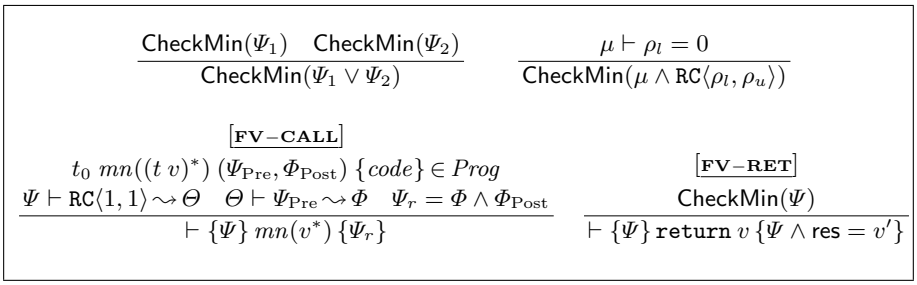

Fig. 7. Hoare Verification Rules: Method Call and Return

As a standard preprocessing step, we check that all predicate invariants are satisfied, including the invariants of resource constraints: the resource assertion $\mathtt{RC}\langle \rho_l, \rho_u \rangle$ in precondition Ψ_{Pre} is consistent if $0 \leq \rho_l \leq \rho_u$, that is, for each disjunct $\mu \wedge \mathtt{RC}\langle \rho_l, \rho_u \rangle$ of Ψ_{Pre} it follows that $\mu \vdash \rho_u \geq \rho_l \wedge \rho_l \geq 0$. We observe that the invariant check on $\mathtt{Term}\ X$ requires that every element of X be non-negative to ensure a non-negative upper-bound $\mathcal{L} \circ X$, so that the execution capacity satisfies the invariant $0 \leq 0_f \leq \mathcal{L} \circ X$.

In the method call rule $[\mathbf{FV\text{-}CALL}]$, the available execution capacity is first decreased by one step, denoted by $\mathtt{RC}\langle 1,1\rangle$, to account the cost of method call, followed by a check that the callee's requirements are met. This check is translated into an entailment for proving the method precondition. Finally, the poststate after this method call is computed. With the help of the resource-enhanced entailment system introduced in Sec. 3.3, both logical and resource proving are combined into one entailment, resulting in a standard-looking Hoare rule for method call.

In addition, specifically for temporal constraints, two entailments $\Psi \vdash \mathtt{RC}\langle 1,1\rangle \rightsquigarrow \Theta$ and $\Theta \vdash \Psi_{\mathrm{Pre}} \rightsquigarrow \Phi$ can be combined into $\Psi \vdash \Psi_{\mathrm{Pre}} \rightsquigarrow \Phi$ by using a new entailment $\vdash_t$ for temporal constraints.

Definition 5 (Unit Reduction Temporal Entailments) *Given temporal constraints θ, θ_1 and θ_2, $\theta \vdash_t \theta_1 \blacktriangleright \theta_2$ iff $\forall s, h, r \cdot$ if $(s, h, r) \models \theta$ then $(s, h, r \ominus (1,1)) \models \theta_1 \blacktriangleright \theta_2$.*

Therefore, if θ is $\mathtt{Loop}$ or $\mathtt{MayLoop}$ then $\theta \vdash_t \theta_1 \blacktriangleright \theta_2$ iff $\theta \vdash \theta_1 \blacktriangleright \theta_2$. If θ is $\mathtt{Term}\ X$ then $\mu \wedge \mathtt{Term}\ X \vdash_t \mathtt{Term}\ Y \blacktriangleright \mathtt{Term}\ ((X -_d 1_d) -_d Y)$ if $\mu \Longrightarrow Y <_d X$, where 1_d is the unit of termination measures' domain. Basically, the check $Y <_d X$ is equivalent to the check that termination measures are decreasing across recursive method calls in the traditional termination proof. By introducing the temporal entailment $\vdash_t$, we obtain a resource-based temporal logic which is related to only the temporal constraints and thus the underlying resource reasoning becomes implicit.

In the method return rule $[\mathbf{FV\text{-}RET}]$, the CheckMin predicate, which is also defined in Fig. 7, ensures that the specified minimum computation resource has been completely consumed when the method returns. Note that if the method does not terminate, the minimum guaranteed execution length is always satisfied since the actual return point is never reached. For temporal constraints, CheckMin holds for any $\mathtt{Term}\ X$ and $\mathtt{MayLoop}$ as the lower bounds in their execution capacities are always 0. In non-termination cases, $\mathsf{CheckMin}(\mu \wedge \mathtt{Loop})$ only holds when μ is unsatisfiable. This check ensures that a return statement cannot be executed/reachable from a state satisfying $\mathtt{Loop}$.

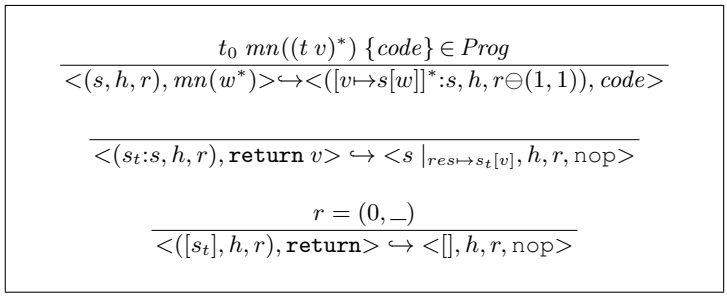

Fig. 8. Key Rules in Operational Semantics

4.3 Soundness

Our goal here is to prove the soundness of our resource-aware Hoare logic for execution lengths. First, we outline an operational semantics for the verified strict imperative language. Second, we define a Hoare triple with respect to this operational semantics and prove the soundness of our Hoare rules, *i.e.*, the operational semantics would get stuck on executions starting in states that falsify the resource assertions.

Operational semantics. We have modified a standard small-step operational semantics to incorporate the execution capacity. In Fig. 8, we list only the method call and return steps; the other steps do not interact with the execution capacity in any interesting way. As mentioned previously, our core language does not have loops. Therefore, execution capacity is only consumed at method calls.

The formulation of the method call step ensures that *at least one* execution step is still allowed by the current execution capacity, via the capacity subtraction $r \ominus (1,1)$, corresponding to the first entailment in the verification rule $[\textsc{fv-call}]$ in Fig. 7. As a result, the semantics will not allow (*e.g.*, eventually get stuck on) executions which requires more resource than the available resource upper bound in the initial states. For example, the semantics will not allow infinite executions from states in which the capacity has finite values.

The return operational rule ensures that executions do not finish if the resource lower bound has not been consumed all. That is, the operational semantics prohibits the return step if the call stack has height 1 and the execution capacity has a non zero lower bound, which would equate with a return from the outermost method before all the required steps have been taken.

Hoare Triples. We define the Hoare triple in a continuation-passing style as in Appel and Blazy [4]. A *configuration* is a pair of code k and state σ. We say a configuration is *safe*, written $\mathsf{safe}(k, \sigma)$, if all reachable states are safely halted or can continue to step:

$$\begin{aligned}\mathsf{safe}(k,\sigma) \ \equiv\ & \forall k', \sigma' \cdot <\sigma, k> \hookrightarrow^* <(s', h', r'), k'> \rightarrow \\ & ((k' = \texttt{nop} \wedge s' = [] \wedge r' = (0, \_)) \ \vee\ \exists \sigma'', k'' \cdot <\sigma', k'> \hookrightarrow <\sigma'', k''>)\end{aligned}$$

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

$$\mathsf{guards}(P, k) \;\equiv\; \forall \sigma \cdot\; \sigma \models P \;\rightarrow\; \mathsf{safe}(k, \sigma)$$

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

$$\{\Psi\}c\{\Phi\} \;\equiv\; \forall k \cdot\; \mathsf{guards}(\Phi, k) \;\rightarrow\; \mathsf{guards}(\Psi, c; k)$$

Note that we dramatically simplified Appel and Blazy's Hoare tuple to include just enough detail to indicate how the temporal assertions fit into the setup without overwhelming the presentation. We conclude by stating the key soundness theorems.

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

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

In addition, by guaranteeing that the Hoare tuple $\{\Psi\}c\{\Phi\}$ holds, the safety theorem also implies that the postcondition holds after the execution of the code c. More precisely, this style of Hoare tuple implies the expected soundness property for any decidable postcondition.

Proposition 1. *If* $\vdash \{\Psi\}\, c\, \{\Phi\}$ *then* Φ *holds after the execution of the code* c.

Proof. To show that if the Hoare tuple $\{\Psi\}c\{\Phi\}$ holds then the postcondition Φ holds, we design the continuation k as a "tester" program that tests the resulting state and gets stuck if the test fails, otherwise does nothing. For example, consider a postcondition $\Phi = x{>}3$, we can use a continuation k:

$$k = \mathtt{if}\ (\mathtt{x} > 3)\ \mathtt{then}\ \mathtt{skip}\ \mathtt{else}\ \mathtt{get\_stuck}$$

such that k will be safe iff the state of the machine after c's execution satisfies Φ. Thus, we know Φ guards k. We can feed that fact into our Hoare tuple to get Ψ guards $c; k$. Therefore we know that either: (i) c does not terminate, or (ii) c does terminate, and the resulting state is enough to make k safe, which implies (by k's construction) that Φ holds after c terminates. □

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

Proof. In [4], it is proven sound a set of Hoare rules very similar to ours for a language that has many of the same features, *e.g.*, load/ store/ assignment/ conditional, making the proofs of these features very similar. □

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

Proof. The proof of the return rule is standard, except in the case of returning from a method requiring the resource assertion $\mathtt{RC}\langle l, u\rangle$ whose the lower bound l is larger than the actual execution length of the method. In this case, operational semantics must get stuck. The Hoare rule for return requires that the CheckMin predicate holds, meaning that return is not executed with any such precondition. That is, the Hoare rule prohibits the execution of a return from a program state with $r = (l, \_)$ where $l > 0$, which

describes a superset of the states in which the operational semantics would block when executing a return step. Specifically, a proper Hoare derivation guarantees that non-terminating code never returns since the lower bound $l = \infty$ has never been consumed all.

The proof for the method call rule hinges on the proof that the precondition guarantees that there exists an execution capacity with a smaller upper bound that suffices for the callee. The resource-enhanced entailment from Sec. 3.3 and the Hoare rule for method call guarantee exactly this, meaning that a proper Hoare derivation guarantees that all function calls requires smaller upper bounds in their execution capacities than the available resource in the current program state. Hence, this guarantees the upper bound requirement of the resource assertion is never violated. □

We have used an operational semantics enriched with execution counters to show that a proper Hoare derivation guarantees that the operational semantics never blocks in accordance with the resource specifications. However, the execution counters do not have a counterpart in a "real machine" as modelled by a standard operational semantics. Below we will outline one such standard, erased semantics and show that our enriched semantics is a strict subset of the erased semantics. Thus the soundness results for our resource logic with regards to the enriched semantics simply carry to the erased semantics.

$$\frac{t_0\ mn((t\ v)^*)\ \{code\} \in Prog}{<(s,h), mn(w^*)> \hookrightarrow <([v{\mapsto}s[w]]^*{:}s, h), code>}$$

$$\frac{}{<(s_t{:}s, h), \texttt{return}\ v> \hookrightarrow <(s\ |_{res\mapsto s_t[v]}, h), \texttt{nop}>}$$

Theorem 4 (Erasure). *The set of executions allowed by the enriched operational semantics is a subset of the set of executions allowed by the erased operational semantics.*

Proof. Each rule in the enriched operational semantics directly corresponds to a rule in the erased operational semantics that has precisely a subset of its premises. Since the enriched state never affects the erased state (except for perhaps making the machine get stuck more often), any execution (sequence of operational steps) in the enriched semantics corresponds directly to an execution in the erased semantics. □

4.4 Flow-Insensitive Temporal Logic

Observe that the current formulation of the temporal logic with temporal constraints is *flow-sensitive* since the entailment $\theta \vdash_t \theta_1 \blacktriangleright \theta_2$ might return a residue θ_2 distinct from θ. However, with the following observations, we can formalize a *flow-insensitive* version of the temporal logic and provide a further abstraction on the resource-based framework presented so far.

First, it is possible to refine the granularity of the termination and non-termination verification by tracking only execution lengths of (mutually) recursive method calls. Second, using König's lemma [29], it is sufficient to inspect individual execution traces in the call tree for deciding just termination or non-termination, instead of tracking the total execution length of all traces in the call tree. That is, a program terminates iff every execution trace is finite; otherwise, the program is non-terminating.

Based on these observations, the tracked resource will be abstracted to capture the execution capacity required for the longest trace in the call tree, instead of the execution capacity required for the remaining program. With this, the resource (for the longest trace allowed) remains unchanged after each splitting operation, which determines the residue resource needed for subsequent method calls. Thus, for every method, we endeavor to provide a single abstract resource that is sufficient for executing a given method call and also its remaining code sequences.

By using this abstraction, we can obtain a formulation on temporal entailment that ensures $\theta \vdash_t \theta_1 \blacktriangleright \theta$ whereby the temporal constraint in residue is always identical to the one in the antecedent. Therefore, we can omit the definition for the operator $-_d$. Moreover, the finite upper bound ϖ used for the definition of $\texttt{Term}\ X$ in Defn. 4 can be determined as $\varpi = \mathcal{D} \circ X$, instead of the larger $\mathcal{L} \circ X$. As a result, without any change to the Hoare rules, during a method's verification, the same initial resource capacity is used for the verification of call traces and thus facilitating a simpler verification procedure for temporal constraint. As a direct outcome of this abstraction, the temporal assertions `Loop`, `MayLoop` and $\texttt{Term}\ X$ are now *flow-insensitive*, and therefore closer to the pure logic form, as opposed to the sub-structural form of resource logics. Note that flow-insensitive label applies to only the temporal constraints. In general, program states (*e.g.*, denoted by separation logic as the underlying logic) remain flow-sensitive since they might be changed due to changes on heap state and program variables.

5 Experiments

We have implemented the proposed termination logic into an automated verification system, called *HipTNT*. The integration of the termination logic into an existing system allows us to utilize the infrastructure that has been developed for some richer specification logics, such as separation logic, beyond a simple first-order logic. Consequently, we are able to *specify* and *verify* both termination and non-termination properties, in addition to correctness properties for a much wider class of programs, including heap-manipulating programs. In this system, the final proof obligations are automatically discharged by off-the-shelf provers, such as Z3 [17]. The expressivity of our new integrated logic is shown in the following experimental results, in which the lexicographic order is needed for about 25% of our experimental programs.

5.1 Numerical Programs

The verification system was evaluated using a benchmark of over 200 small numerical programs selected from a variety of sources: (i) from the literature, such as [14,12], (ii) from benchmarks used by other systems (that are AProVE [21], Invel [38] and Pasta [19]) and (iii) some realistic programs, such as the Microsoft Zune's clock driver that has a leap-year non-termination bug. Most of the methods in these benchmark programs contain either terminating or non-terminating code fragments, expressed in (mutual) recursive calls or (nested) loops. To construct these benchmarks we added the novel termination specifications to the original examples from the analysis tools for termination and non-termination. We have chosen these benchmarks in order to show

Benchmarks	**Programs**	`Term`	`Loop`	`MayLoop`	**PC**(s)	**TC**(s)	**Overhead** (%)
Invel	59	137	81	12	14.88	15.96	6.77
AProVE	124	534	120	8	15.73	17.21	8.60
Pasta	44	219	10	3	4.95	5.79	14.51
Others	48	194	32	22	7.35	8.78	16.29
Totals/(%)	**275**	**1084** (79.0%)	**243** (17.7%)	**45** (3.3%)	**42.91**	**47.74**	10.12%

Fig. 9. Termination Verification for Numerical Programs

the usability and practicality of our approach. A comparison with these tools would be of less relevance as our proposal focuses on verifying the given specifications rather than infer them.

To express the programs' behavior more precisely and concisely, we integrated the proposed logic into an enhanced mechanism for structured specification with automatic case analysis [20]. The more expressive specification language can be formally described by the following grammar rule:

$$\mathsf{Y} ::= \texttt{case}\ \{\pi_1 \Rightarrow \mathsf{Y}_1; \ldots; \pi_n \Rightarrow \mathsf{Y}_n\} \mid \texttt{requires}\ \Psi\ \mathsf{Y} \mid \texttt{ensures}\ \Phi$$

It allows the decomposition of a program's behavior into multiple disjointed scenarios (guarded by π_i) for easier comprehension. This decomposition also helps with verification performance, as it helps to reduce disjunctive form and to avoid repeated proving of common sub-formula. For example, the new structured specification of the `sumE` method in Sec. 2, can be defined as follows:

```
int sumE (int n)
  case { isEvenNat(n) ⇒ requires Term[n] ensures true;
         ¬isEvenNat(n) ⇒ requires Loop ensures false; }
```

Fig. 9 summarizes the characteristics and the verification times for a benchmark of numerical programs. Columns 3-5 describe the number of preconditions that have been specified and successfully verified as *terminating*, *non-terminating* or *unknown*, respectively. As hoped for, the number of preconditions annotated by `MayLoop` occupies the smallest fragment (about 3%) of the total number of preconditions. Such `MayLoop` constraints were only used in some unavoidable scenarios as discussed in Sec. 2. In contrast, the `Term` constraints (with the given measures) are in the majority because most of the methods are expected to be terminating, except for the Invel benchmark which focuses on mostly non-terminating programs.

Our verification system can perform both correctness and termination proofs. Column 7 (**TC**) gives the total timings (in seconds) needed to perform both termination and correctness proofs for all the programs in each row, while column 6 (**PC**) gives the timings needed for just correctness proofs. The difference in the two timings represents the small overheads needed for termination and non-termination reasoning.

5.2 Heap-manipulating Programs

As illustrated in Fig. 10, we have also conducted termination reasoning on our own benchmark of heap-based programs using various data structures with a small over-

Programs	**LOC**	**Procedures**	`Term`	`Loop`	`MayLoop`	**PC**(s)	**TC**(s)	**Overhead** (%)
AVL	390	13	18	0	0	13.89	14.66	5.25
Linked List (LL)	135	13	13	0	0	0.28	0.29	3.45
Sorted LL	480	13	15	0	0	1.33	1.38	3.62
Circular LL	80	4	4	4	0	1.04	1.18	11.86
Doubly LL	174	11	12	0	0	0.41	0.46	10.87
Complete	112	6	7	0	0	2.58	3.53	26.91
Heap Tree	214	5	6	0	0	14.82	15.12	1.98
BST	165	6	6	0	0	0.93	1.04	10.58
Perfect Tree	83	5	5	1	0	0.32	0.33	3.03
Red-Black Tree	556	19	25	0	0	6.22	6.40	2.81
BigNat	235	18	18	0	0	15.13	15.42	1.88
Totals/ (%)	**2624**	**114**	**129** (96.3%)	**5** (3.7%)	**0**	**56.95**	**59.81**	4.78%

Fig. 10. Termination Verification for Heap-manipulating Programs

head. The modular structure of the resource reasoning framework in Sec. 3 facilitates the embedding of temporal constraints into a richer specification mechanism based on separation logic, automatically extending it to proving termination or non-termination properties over heap-manipulating programs. The temporal entailment judgment in the [ENT−CONJ] rule can leverage the power of the separation logic entailment engine to discharge the temporal constraints in a heap-related entailment.

For example, consider the following entailment, which might be encountered when verifying a method call with heap arguments (*e.g.*, the length method of linked lists),

$$\mathtt{ll}(\mathtt{x},\mathtt{n}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{Term}[\mathtt{n}] \vdash \mathtt{x}{\mapsto}\mathtt{node}(\_,\mathtt{y}){*}\mathtt{ll}(\mathtt{y},\mathtt{n}_1){\wedge}\mathtt{Term}[\mathtt{n}_1]$$

The entailment prover for separation logic can infer the constraint $\mathtt{n}_1{=}\mathtt{n}{-}1$ (*e.g.*, by the unfolding mechanism and explicit instantiation mechanism introduced in [36]) when checking the spatial part of the entailment, which is a necessary condition to ensure the validity of the eventual temporal entailment judgment

$$\mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{n}_1{=}\mathtt{n}{-}1 \wedge \mathtt{Term}[\mathtt{n}] \vdash_t \mathtt{Term}[\mathtt{n}_1] \blacktriangleright \mathtt{Term}[0]$$

Due to the tight integration with the underlying logic, this task of specifying and verifying the termination properties was easy even though some of the programs use non-trivial data structures (*e.g.*, Red-Black and AVL-trees), or non-linear constraints (*e.g.*, the BigNat program, which implements infinite precision natural numbers (by linked lists) with procedures for some arithmetic operations, in addition to a fast multiplication method based on the Karatsuba algorithm).

We have successfully determined that none of the above 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 a perfect tree would go into an infinite loop if a negative number was given as its height. Furthermore, during the verification 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.

Programs	**LOC**	**Proc.**	Term	Loop	MayLoop	**PC**(s)	**TC**(s)	**Overhead** (%)
TPDB Benchmark								
Shuffle	20	2	2	0	0	0.23	0.26	11.54
LessLeavesRec	22	2	2	0	0	0.30	0.36	16.67
Alternate	23	2	2	0	0	0.37	0.39	5.13
SortCount	32	3	6	0	0	3.06	3.45	11.30
UnionFind	39	5	8	0	0	0.51	0.53	3.92
DivTernary	55	9	12	0	0	0.77	0.87	11.49
WorkingSignals	126	17	23	0	0	8.74	9.50	8.00
MinusUserDefined	21	2	10	0	0	0.30	0.36	16.67
MultiLasso	14	1	3	1	1	0.12	0.13	7.69
Totals/(%)	**352**	**43**	**68** (97.14%)	**1** (1.43%)	**1** (1.43%)	**14.40**	**15.85**	9.15%

Fig. 11. Termination Verification for the SIR/Siemens and TPDB Benchmark

The partial correctness proof did not detect this problem. It was later corrected into a terminating method, a courtesy of our newly integrated feature.

5.3 SIR/Siemens and Termination Problems Data Base (TPDB) Benchmarks

Moreover, the termination verification has also been done on some medium programs taken from the SIR/Siemens industrial test suite [18] and selective problems from the Termination Competition [34], as shown in Fig. 11. Beside data structures, some programs in the SIR/Siemens benchmark also use arrays in their implementation (*e.g.*, *tcas* and *replace* programs). For the Termination Competition's problems, we select problems in the Java Bytecode categories that cannot be automatically proved by the competitors, such as AProVE and Julia.

However, the termination of all procedures in these benchmarks can be verified successfully with a small overhead (about 5%) by our system. Note that for the *printtokens* programs of the SIR/Siemens benchmark, some of their methods required a practical assumption that the size of input files was finite for their termination; otherwise they might not terminate as indicated by failures of the termination verification.

6 Related Work and Conclusion

There exists a rich body of related works on automatic analysis for termination [31,9,15], non-termination [23,38,11], and both [21]. However, they consider termination and non-termination reasoning as distinct from functional correctness reasoning. Therefore, these works cannot leverage the result of functional correctness analysis to conduct more intricate (non-)termination reasoning. Recently, Brockschmidt *et al.* [10] propose a cooperation between safety and termination analysis to find sufficient supporting invariants for the construction of termination arguments but not considering non-termination. Chen *et al.* [13] introduce a similar approach for proving only non-termination. Our proposal complements these works since our aim is to construct a logic

where termination and non-termination properties are directly integrated into specification logics, and thus utilize the available infrastructure on functional correctness proofs. We have achieved this, and have also successfully evaluated its applicability on a wide range of programs, covering both numerical and heap-based programs.

Related to resource verification, [5] introduces a resource logic for a low-level language. While this logic avoids the need of auxiliary counters, it redefines the semantic model of the underlying logic to track the resource consumption via logical assertions, making the proposal harder to retrofit to other logics. Moreover, this logic only targets partial correctness, so that it does not take into account infinite resource consumption.

There are some works that are based on the well-foundedness of inductive definitions of heap predicates [8,12] or user-defined quantitative functions over data structures [22] to prove termination of heap-manipulating programs. On one hand, they do not require any explicit ranking function. On the other hand, these approaches might have problems with programs like the Karatsuba multiplication method, in which the arguments of the recursive calls are not substructures of the input lists. In addition, the automated tools, such as AProVE and COSTA, cannot prove the termination of this method. In contrast, our approach is more flexible as it allows explicit termination measures, that are possibly non-linear, for proving programs' termination. These termination measures can be constructed from not only the heap structures but also the values of the data structures' elements. For example, we use the actual value of the natural presented by a linked list to bound the execution of the Karatsuba method. Moreover, we also allow non-termination to be specified and verified for these programs. We believe that relatively complex examples, such as the Karatsuba method, highlight the benefits of our approach, which trades a lower level of automation but gains additional power.

The comparison of our approach with the other specification languages, *i.e.*. Dafny [32], JML [30], etc., has been discussed in Sec. 1. Another closely related work to ours is that of Nakata and Uustalu [35]. In this work, a Hoare logic for reasoning about non-termination of simple While programs (without method calls) was introduced. The logic is based on a trace-based semantics, in which the infiniteness of non-terminating traces is defined by coinduction. However, induction is still needed to define the finiteness of traces. In contrast, with resources, we can unify the semantics of the proposed termination and non-termination temporal constraints and allow the Hoare logic for functional correctness to be enhanced for termination and non-termination reasoning with minor changes. Moreover, our logic allows interprocedural verification in a modular fashion.

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 combine the two areas more closely together, through a tightly coupled union. Our unique contribution is to embed both termination and non-termination reasoning directly into specification logics, and to do so with the help of temporal entailment. We also show how its properties can be captured by a resource logic based on execution capacity, and how it could be abstracted into a flow-insensitive temporal logic. We believe this approach would have benefits. Its expressiveness is immediately 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 program analysis. In particular we believe that a possible future avenue for investigation is to use

the safety specifications as a basis for termination specification inference. Last, but not least, it has placed termination and non-termination reasoning as a first-class concept, much like what was originally envisioned by Hoare's logic for total correctness.

References

1. A. Abel. Type-based termination of generic programs. *SCP*, 74(8), 2009.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *SSM*, 4, 2005.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *FMCO*, 2008.
4. A. W. Appel and S. Blazy. Separation logic for small-step C minor. In *TPHOLs*, 2007.
5. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *TCS*, 389(3), 2007.
6. R. Atkey. Amortised resource analysis with separation logic. *LMCS*, 7(2), 2011.
7. P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI/ISO C Specification Language Version 1.8. *URL http://frama-c.com/acsl.html*, 2013.
8. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic Termination Proofs for Programs with Shape-Shifting Heaps. In *CAV*, 2006.
9. A. R. Bradley, Z. Manna, and H. B. Sipma. The Polyranking Principle. In *ICALP*, 2005.
10. M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *CAV*, 2013.
11. M. Brockschmidt, T. Strder, C. Otto, and J. Giesl. Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In *FoVeOOS*, 2012.
12. J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, 2008.
13. H.-Y. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn. Proving nontermination via safety. In *TACAS*, 2014.
14. B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving Conditional Termination. In *CAV*, 2008.
15. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
16. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *SEFM*, 2012.
17. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
18. H. Do, S. G. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. In *ESE*, volume 10, 2005.
19. S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE*, 2009.
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, P. Schneider-Kamp, and R. Thiemann. Automatic termination proofs in the dependency pair framework. In *IJCAR*, 2006.
22. S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.
23. A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R.-G. Xu. Proving nontermination. In *POPL*, 2008.
24. E. C. R. Hehner. Termination is timing. In *MPC*, 1989.
25. E. C. R. Hehner. Specifications, programs, and total correctness. *SCP*, 34(3), 1999.

26. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
27. J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In *ESOP*, 2010.
28. C. B. Jones. Balancing expressiveness in formal approaches to concurrency. 2013.
29. S. Kleene. *Mathematical logic*. Wiley, 1967.
30. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
31. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, 2001.
32. R. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, 2010.
33. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, 2010.
34. C. Marché and H. Zantema. The termination competition. In *RTA*, 2007.
35. K. Nakata and T. Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. In *ESOP*, 2010.
36. H. H. Nguyen, C. David, S.-C. Qin, and W.-N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, 2007.
37. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.
38. H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *TAP*, 2008.
39. H. Xi. Dependent Types for Program Termination Verification. In *LICS*, 2001.