

Automated Inference and Testing of Separation Logic Specifications with Magic Wands

ANONYMOUS AUTHOR(S)

A fundamental challenge in program verification is devising specifications of a program's intermediate states, e.g., loop invariants, that are sufficient to verify the program's top-level specification. This is particularly challenging for heap-manipulating programs equipped with rich specifications in separation logic, where the decidability of entailment checking is restricted to certain fragments. In this paper, we propose a technique that combines LLMs and testing to automatically infer invariants which use *magic wands*, i.e., separating implication, to naturally capture functional correctness properties of loops iterating over heap-allocated data structures. Key to our approach is a novel run-time testing technique for CN-style separation logic specifications extended with magic wands. We evaluate our approach on a set of programs operating over rich data structures, including doubly linked lists and binary trees. Our experiments demonstrate that our testing technique is able to effectively discard invalid invariants proposed by an LLM, enabling the end-to-end pipeline to effectively propose correct invariants in a majority of cases, across a range of models and reasoning levels.

1 Introduction

Program verification fundamentally relies on some user-provided specification of the desired (or undesired) behaviors of a target program. Importantly, these specifications are typically not limited to a characterization of the program's input-output behaviors: a successful verification usually depends on additional specifications, such as loop invariants, which describe intermediate program states. These auxiliary specifications are often quite complex, and asking users to manually supply them imposes a significant hurdle to the adoption of program verification. As a consequence, the question of how to automatically infer intermediate specifications has received significant attention from the programming languages and formal methods communities [15, 30, 34, 37, 40, 46].

Alongside these efforts, the community has developed a number of new specification languages which are capable of capturing semantically rich properties of real-world programs. Perhaps the most significant of these is separation logic [31], which has enabled modular reasoning about heap-manipulating programs written in mainstream languages including C [11] and Rust [16]. The key novelties of separation logic are its *separating conjunction* ($*$) and *separating implication*, i.e., *magic wand* ($\multimap$) operators, which enable local reasoning about a program's behavior to be soundly generalized to larger contexts. The additional expressivity of separation logic comes at a cost, however, as the validity of separation logic formulas is undecidable in general. To address this limitation, most automated verifiers based on separation logic either only permit specifications in restricted decidable fragments that exclude magic wands, or require significant user guidance in the form of additional program annotations. This is unfortunate, as many properties of intermediate program states are naturally captured by magic wands.

As an example, consider the simple `addN` function shown in Fig. 1, which increments every element of a singly linked list by the integer pointed to by its first argument. This behavior is encoded in the pre- and post-conditions in the comments on lines 1–2 and 16. Both assertions use the separating conjunction, $*$, to stipulate that `addN`'s arguments point to disjoint locations in the heap, ensuring that the value pointed to by `p` is not overwritten by the loop on lines 10–13. A successful verification of `addN` requires a loop invariant like the one in the comment spanning lines 5–9. The given invariant combines separating conjunction and magic wand, using the former to capture the portion of the heap that has not yet been modified (line 7), and the latter to describe the part

```

50 1  /*  $\forall e \in l, \text{minI32} \leq e + n \leq \text{maxI32},$ 
51 2   $p \mapsto n * \text{lrep}(l, h) */$ 
52 3  struct sll* addN(int* p, struct sll* h) {
53 4      struct sll *current = h;
54 5      /*  $\exists ls: \text{list } Z, \text{haddr: address}, \text{haddr} == \text{current},$ 
55 6       $\forall e \in ls, \text{minI32} \leq e + n \leq \text{maxI32},$ 
56 7       $p \mapsto n * \text{lrep}(ls, \text{haddr}) *$ 
57 8       $\text{lrep}(\text{map } (\lambda x. x + n) ls, \text{haddr}) *$ 
58 9       $\text{lrep}(\text{map } (\lambda x. x + n) l, h) */$ 
59 10 while (current != NULL) {
60 11     current->key = current->key + *p;
61 12     current = current->next;
62 13 }
63 14 return h;
64 15 }
65 16 /*  $p \mapsto n * \text{lrep}(\text{map } (\lambda x. x + n) l, h) */$ 

```

```

struct sll
{
    int key;
    struct sll *next;
};

(l : list Z, h : address)
lrep(l, h) :=
    l = []  $\rightarrow (h = \text{nil}), \text{emp}$ 
    l = hd :: tl  $\rightarrow \text{minI32} \leq \text{hd} \leq \text{maxI32},$ 
     $\exists ht, h \mapsto (\text{hd}, ht) * \text{lrep}(tl, ht)$ 

```

Fig. 1. C function that increments the elements of a singly linked list, alongside the struct sll definition and the lrep predicate

already traversed, capturing the shape of the final heap (line 9) after all future changes have been made (line 8). Importantly, this invariant is expressed using the same predicates as addN's user-supplied top-level specification. While there has been recent work on inferring separation logic specifications [14, 18, 21–23, 28, 29, 37], to the best of our knowledge, none of these techniques can infer invariants that describe functional correctness properties using magic wands like the one shown in this example.

This paper proposes a specification inference technique for loop invariants with magic wands that combines recent improvements in the capabilities of LLMs to propose plausible invariants [23, 41, 42] and property-based testing of separation logic specifications [1, 2, 4]. At a high level, this procedure implements a CEGIS-style [36] inference loop, relying on an LLM to propose a candidate invariant and then relying on property-based testing to weed out invalid candidates. Our key technical contribution is a novel technique for testing whether a given heap state satisfies a fragment of separation logic with magic wands. Similar to other recent works [4], we restrict these formulas to ensure efficient checking, relying on shared pointers to connect the current heap with a hypothetical heap state that will exist in the future. Our experiments demonstrate that our testing technique is able to effectively discard invalid invariants proposed by an LLM, enabling the end-to-end pipeline to eventually propose correct invariants in a majority of cases, across a range of models and reasoning levels.

This paper is organized as follows: we begin with a review of separating implication, illustrating its ability to express invariants for loops that iterate over heap-allocated structures. From there, we present our high-level loop invariant inference procedure, followed by a detailed accounting of our specification language and our corresponding testing algorithm. We next present an evaluation of SMEAGOL, a prototype implementation of this technique, before concluding with a discussion of related work and future directions.

2 Background: Why the Wand?

Intuitively, a separating conjunction of the form $P * Q$ is satisfied by a heap σ that can be partitioned into two disjoint subheaps, $\sigma = \sigma_1 \uplus \sigma_2$, one satisfying P , $\sigma_1 \models P$, and the other satisfying Q , $\sigma_2 \models Q$. When combined with inductively defined predicates, such

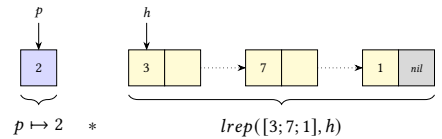


Fig. 2. A heap satisfying the precondition of addN, with $n = 2$ and $l = [3; 7; 1]$.

```

99 1  /*  $\forall e \in l, \text{minI32} \leq e + n \leq \text{maxI32},$ 
100 2    $p \mapsto n * \text{lrep}(l, h) */$ 
101 3  struct sll* addN(int* p, struct sll* h) {
102 4   struct sll *current = h;
103 5   /*  $\exists ls: \text{list } Z, \text{haddr}: \text{address},$ 
104 6     haddr == current,
105 7      $\exists l1: \text{list } Z, l == l1 ++ ls,$ 
106 8      $\forall e \in ls, \text{minI32} \leq e + n \leq \text{maxI32},$ 
107 9      $p \mapsto n * \text{lrep}(ls, \text{haddr}) *$ 
108 10    lseg(map ( $\lambda x. x + n$ ) l1, h, haddr) */
109 11  while (current != NULL) { ... }

```

$(l : \text{list } Z, h_1, h_2 : \text{address})$
 $\text{lseg}(l, h_1, h_2) \doteq$
 $l = [] \rightarrow (h_1 = h_2), \text{emp}$
 $l = \text{hd} :: \text{tl} \rightarrow \text{minI32} \leq \text{hd} \leq \text{maxI32},$
 $\exists ht, h_1 \mapsto (\text{hd}, ht) * \text{lseg}(\text{tl}, ht, h_2)$

Fig. 4. A variant of addN whose loop invariant uses only separating conjunction, alongside the lseg predicate it requires. The additions over the wand-based invariant from Fig. 1 are highlighted.

as lrep in Fig. 1, this operator allows us to succinctly describe the shape of portions of the heap in terms of algebraic datatypes, like lists: the precondition of addN is satisfied by a heap like the one shown in Fig. 2, for example.

The separating implication, $P * Q$, in contrast, describes a heap σ that, when combined with any (disjoint) hypothetical heap satisfying its antecedent P , $\sigma_1 \models P$, results in a heap satisfying its consequent Q , $\sigma_1 \uplus \sigma \models Q$. When used in a loop invariant, the antecedent of a magic wand can thus naturally capture any changes to the heap that will eventually occur, and its consequent captures the heap that results when the loop is finished. Fig. 3 shows the heap that results after a single iteration of the loop in addN. Observe that the green portion of the heap satisfies the wand: combining it with any heap satisfying $\text{lrep}([9; 3], \text{haddr})$ yields a heap satisfying $\text{lrep}([5; 9; 3], h)$.

While it is possible to define an alternative loop invariant for addN that uses only separating conjunction, doing so requires the use of an additional shape predicate that captures the portion of the heap that has been traversed so far. Fig. 4 illustrates this approach using the lseg predicate. This invariant requires an additional ghost variable $l1$ that models the portion of the original list traversed so far, as well as a proposition relating it to the original list.

While other linear data structures, e.g., doubly linked lists, require analogous additions (Appendix A of the supplementary material), nonlinear structures can require significantly more. Consider the heap in Fig. 5, which holds a binary tree t rooted at root , in the middle of a loop that traverses the tree using the pointer r . The subtree rooted at r (yellow), which we model as t_2 , is the portion of the tree still to be explored; the rest of the tree (green) has already been processed. As with addN, a magic wand for a binary tree can elegantly capture the heap using only the predicate from the top-level specification:

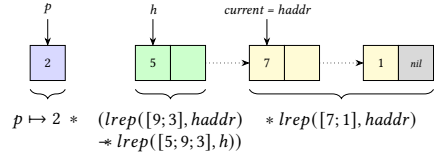


Fig. 3. The heap of Fig. 2 after one iteration of the loop in addN, where $ls = [7; 1]$.

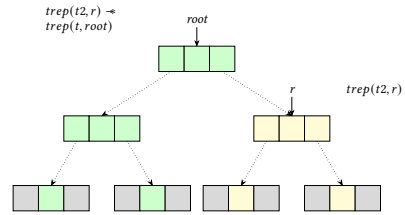


Fig. 5. A heap holding a binary tree, where root points to the root of the tree and r points to a subtree encountered by a loop traversing this structure.

$$\exists t_2 : \text{tree}, \text{trep}(t_2, r) * (\text{trep}(t_2, r) \multimap \text{trep}(t, \text{root}))$$

```

148 Inductive tree :=
149   | E : tree
150   | Hole : tree
151   | T : tree → key → value → tree → tree.
152
153 (t : tree, p : address)
154 trep(t, p) :=
155   t = E → (p = nil), emp
156   t = Hole → False
157   t = T lt k v rt → minI32 ≤ k ≤ maxI32 ∧
158     minI32 ≤ v ≤ maxI32,
159     ∃ pl, ∃ pr, p ↦ (k, v, pl, pr) *
160     tseg(lt, pl) * trep(rt, pr)
161
162 (t : tree, curr, p : address)
163 tseg(t, curr, p) :=
164   t = E → (p = nil), emp
165   t = Hole → (p = curr), emp
166   t = T lt k v rt → minI32 ≤ k ≤ maxI32 ∧
167     minI32 ≤ v ≤ maxI32,
168     ∃ pl, ∃ pr, p ↦ (k, v, pl, pr) *
169     tseg(lt, curr, pl) * tseg(rt, curr, pr)
170
171 (t1, t2 : tree)
172 union(t1, t2) :=
173   t1 = E → E
174   t1 = Hole → t2
175   t1 = T lt k v rt →
176     T (union(lt, t2)) k v (union(rt, t2))

```

Fig. 6. The predicates and helper function needed to express a loop invariant for traversing a binary tree using only separating conjunction. Additions over the wand-based invariant are highlighted in purple.

Expressing the same state using only separating conjunction requires substantially more machinery:

$$\exists t_1 : \text{tree}, \exists t_2 : \text{tree}, t = \text{union}(t_1, t_2), tseg(t_1, r, \text{root}) * trep(t_2, r)$$

The auxiliary definitions used by this invariant are highlighted in purple in Fig. 6. As with lists, the traversed portion of the heap requires an additional predicate, *tseg*. Unlike a list segment, however, a partially traversed tree is missing the subtree at *r*, so its model needs some way to mark where that subtree belongs. This invariant therefore requires extending the *tree* datatype used to model the heap with a new constructor, *Hole*. This constructor is used to record the pointer from *r*'s parent to *r*. Since a complete tree in memory never contains a hole, *trep* must in turn map *Hole* to *False*. Finally, relating the two models back to *t* requires an additional ghost variable *t*₁ and a *union* function that plugs *t*₂ into the hole in *t*₁.

In general, the auxiliary predicates, constructors, functions, and ghost variables required by invariants defined using only separating conjunction grow with the complexity of the underlying data structure. Importantly, each of these components is an additional definition that a user must supply or which an inference procedure must invent. In contrast, wand-based invariants can be expressed using only the predicates already present in a function's top-level specification. SMEAGOL exploits this observation, combining an LLM with run-time testing to infer and validate these simpler invariants.

3 Approach

We begin this section by introducing the separation logic fragment SMEAGOL is equipped with, the data structures we currently support, and our approach towards inference and testing. After that, we introduce the novel syntax and execution semantics of the wand fragment to test specifications with wand at run time. We conclude with a complexity analysis of our testing algorithm.

3.1 Overview

We require our programs and specifications to be deterministic. Magic wands may occur in both invariants and post-conditions, but not in the precondition. We support the following Separation Logic assertions:

$$\tau := \text{emp} \mid x \mapsto y \mid lrep(l, h) \mid dllrep(l, h, \text{preva}) \mid trep(t, p) \quad (1)$$

$$\phi := \tau \mid \phi * \phi \mid \phi \multimap \phi \mid \text{if } c \text{ then } \phi \text{ else } \phi \quad (2)$$

Fig. 7. The syntax of SL assertions considered. The inductive predicates $lrep(l, h)$, $dllrep(l, h, preva)$ and $trep(t, p)$ represent a singly linked list l with head h , a doubly linked list l with head h and previous address $preva$, and a binary tree rooted at r . The mathematical layout of both singly and doubly linked list is a standard nil and cons layout (Fig. 1). For binary trees we define an inductive datatype for both the leaf and non-leaf case). c is a deterministic boolean function.

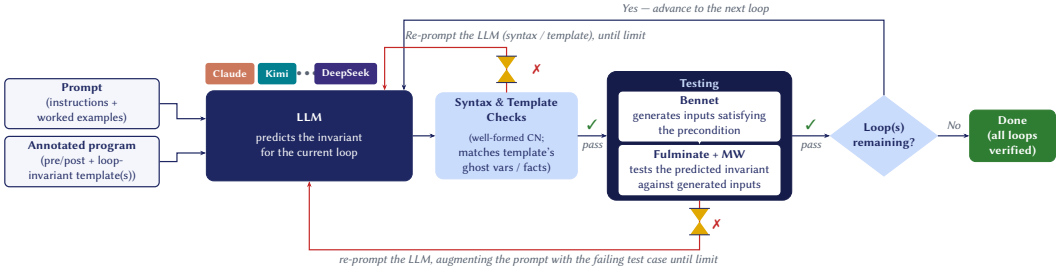


Fig. 8. Overview of SMEAGOL

We do not support nested wands explicitly associated left in this version. Further, we require the wands that are in their default order (right associative, or explicitly grouped right) be normalized, or uncurried to a single wand — i.e., a specification of the form $\phi_1 \multimap \phi_2 \multimap \phi_3 \cdots \multimap \phi_n$ is to be rewritten as $(\phi_1 * \phi_2 * \cdots * \phi_{n-1}) \multimap \phi_n$.

Fig. 8 depicts the high-level workflow of our inference pipeline, which has two segments: generation and testing. Our system takes two inputs: first, the C program, annotated with datatypes, predicates, pre- and post-conditions and the loop invariant templates. Second, the number of times an LLM of your choice can be re-prompted.

Algorithm 1 formalizes the overview, with the first input being the tuple: $(P, Pre, Post, hyp, template)$ and the second input being the *limit*. The output is either a fully-tested function, or the set of loops (in top-down order) we were able to successfully test. The algorithm stops at the first failed loop. Algorithm 1 processes the loop templates of P in top-down order in a CEGIS-style loop [36]. For every loop, the LLM is prompted with the instructions (Figure S3 of the supplementary material) to generate an invariant. The instructions include a template, along with the detailed syntax and a set of examples. The LLM then predicts the invariant (inv) along with additional functions and predicates (h_{extra}) utilized by inv that are not in hyp . inv is then analyzed for the following violations:

- All the ghost variables of the template are utilized
- All the propositions in the template are utilized
- No additional proposition is inferred
- No additional recursive predicates are proposed
- The wand(s) proposed are normalized

If any of the above checks are violated, we re-prompt (Figure S4 of the supplementary material) with the violation message appended to the set of previous violations (if any) for the template under consideration. If the template checks pass, we annotate inv , h_{extra} in P and check for compilation errors. On failure, the LLM is re-prompted with the failure message appended. The testing algorithm is invoked after the annotated program passes both template and compile checks.

Algorithm 1: Loop Invariant Inference

```

Procedure INFERINV(P, Pre, Post, hyp, template, limit)
  Inputs      : P: candidate C function
                Pre, Post: target pre- and post-conditions
                hyp: Set of predicates and functions occurring in Pre, Post and template
                template: The set of loop invariant templates for all loops in P
  Output      : Test Success or Partial Success
  foreach t ∈ templates do
    curPrompt ← initializePrompt(P, Pre, Post, t)
    countlim ← 0
    error ← ∅
    while countlim < limit do
      (inv, hextra) ← GenNxtStep(curPrompt)
      switch AnalyzeLoop(inv, P, Pre, Post, hyp, t) do
        case SYNTAX VIOLATION do
          error ← error ∪ {violation reason}
          curPrompt ← nextStepsPrompt(P, Pre, Post, hyp ∪ hextra, t, error)
          countlim ← countlim + 1
        case TESTING do
          σ ← GenTests(P, Pre)
          result ← TestLoop(σ, inv, P, Pre, Post, hyp ∪ hextra)
          if result = ∅ then
            break
          else
            error ← error ∪ {result}
            curPrompt ← nextStepsPrompt(P, Pre, Post, t, error)
            countlim ← countlim + 1
      if countlim = lim then return (Limit Exceeded, partially filled P)
  return Success

```

The first step towards testing the invariant is generating a set of tests satisfying the pre-condition *Pre*. Since *Pre* does not contain the wand, we directly query Bennet/Darcy, the state-of-the-art PBT generator for SL assertions expressed with $*$, to generate a set of tests σ that model the memory layout satisfying *Pre*. After the tests are generated, we call our testing algorithm (Algorithm 2) with σ , the invariant *inv*, the function *P*, the pre- and post-conditions *Pre*, *Post* and the set of predicates and functions $\text{hyp} \cup h_{\text{extra}}$. The invariant *inv* is a sequence of heap fragments separated by the $*$ operator. A component is either wand-free, that is, it has only separating conjunction and is a memory footprint that does not contain the magic wand, or it is a wand $A_{wa} * A_{wb}$, where A_{wa} is the antecedent and A_{wb} the consequent. Both A_{wa} and A_{wb} can have ownership of multiple cells separated by the $*$ operator.

Algorithms 2 and 3 constitute the run-time testing framework of SMEAGOL. For heap fragments containing magic wand, we take inspiration from Viper’s package inference algorithm [12] and adapt it to run-time testing of loop invariants and post-conditions with our WAND algorithm. Components that only have separating conjunction are handled by the function CONSUME, which is the testing algorithm from Fulminate implemented in CN [4]. (Section 4 contains a full description of how our algorithm interacts with Fulminate.) Algorithm 2 tests the specification for every test case $\sigma_i \in \sigma$, and returns the first failed test case. If all tests pass, it returns $\emptyset$, and we move to the next loop in *P*.

For each test σ_i , Algorithm 2 initializes σ_c and σ_w , the concrete layouts of the wand-free and wand components which combined together form σ_i , to the empty set, and goes through the components of *inv* from left to right.

Algorithm 2: Run-time testing of heap specifications

```

Procedure TestLoop( $\sigma, inv, P, Pre, Post, hyp$ )
Inputs      :  $\sigma$ : heap layouts of test cases satisfying Pre
               inv: the proposed loop invariant
               P: function, Pre, Post: Pre (Post) condition
               hyp: hypothesis space
Output      : Failed Test Case
foreach  $\sigma_i \in \sigma$  do
   $\sigma_c \leftarrow \emptyset$ ;
   $\sigma_w \leftarrow \emptyset$                                 /* Layouts of the wand-free components and of the wands */
  foreach component I  $\in inv$  do
    if I has only * (no wand) then
       $\sigma_a \leftarrow \text{CONSUME}(I, \sigma_i, P, Pre, Post, hyp)$                                 /* Try to take ownership of I */
      if  $\sigma_a \not\subseteq I$  or  $\sigma_a \cap (\sigma_c \uplus \sigma_w) \neq \emptyset$  then return  $\sigma_i$ 
       $\sigma_c \leftarrow \sigma_c \uplus \sigma_a$ 
    else
       $\sigma_n \leftarrow \text{WAND}(I, \sigma_i, P, Pre, Post, hyp, \sigma_c)$                                 /*  $I = A_{wa} * A_{wb}$  */
      if  $\sigma_n \not\subseteq I$  or  $\sigma_n \cap (\sigma_c \uplus \sigma_w) \neq \emptyset$  then return  $\sigma_i$ 
       $\sigma_w \leftarrow \sigma_w \uplus \sigma_n$ 
  if  $\sigma_c \uplus \sigma_w = \sigma_i$  then
    return  $\emptyset$ 
  else
    return  $\sigma_i$ 

```

Algorithm 3: Testing one wand, called by Algorithm 2

```

Procedure Wand( $A_{wa} * A_{wb}, \sigma_i, P, Pre, Post, hyp, \sigma_c$ )
   $\sigma_{wa} \leftarrow \text{POSTULATE}(\sigma_i, P, hyp, Pre, Post, \sigma_c, A_{wa})$  /* Peek: no ownership; fails on a bad read or a clash with  $\sigma_{wa}$  */
  if  $\sigma_{wa} \not\subseteq A_{wa}$  then return fail
   $\sigma_{wb} \leftarrow \text{BORROWINNER}(P, hyp, Pre, Post, \sigma_c, \sigma_{wa}, A_{wb})$  /* Stop at an antecedent address, splice its value */
  if  $\sigma_{wb} \not\subseteq (\sigma_{wa}, A_{wb})$  then return fail
   $\sigma_{wf} \leftarrow \text{BORROWOUTER}(P, hyp, Pre, Post, \sigma_c, \sigma_i, \sigma_{wb}, A_{wb})$  /* Cells B needs beyond those A supplies */
  return  $\sigma_{wf}$  /*  $\sigma_{wf}$  would contain the False heap state on failure */

```

Given $\sigma_i, hyp, P, Pre, Post$ and a wand-free component *I*, CONSUME determines whether the footprint is valid and memory can be allocated. If this fails, or the footprint overlaps the layouts found so far, we immediately return σ_i , otherwise, the footprint is added to σ_c — the set of all footprints defined by the wand-free components.

Every other component has the shape $A_{wa} * A_{wb}$ and is passed to Algorithm 3, which postulates the antecedent A_{wa} first — i.e., from $\sigma_i, P, Pre, Post, hyp$, variables used by σ_c , and the specification A_{wa} which is a function of the above parameters, a temporary heap layout (or, postulated layout) σ_{wa} is constructed that satisfies A_{wa} .

If the heap construction fails then failure is returned, and the failed test layout σ_i is returned to Algorithm 1. On success we move to satisfy the layout corresponding to the consequent A_{wb} . This is carried out in two borrow phases, BORROWINNER and BORROWOUTER. The first step, BORROWINNER attempts to construct part of the heap layout by borrowing from the postulated σ_{wa} . On a successful inner borrow, we proceed to satisfy the remaining layout for the consequent A_{wb} by borrowing from σ_i .

The remaining layout borrowed is the wand footprint σ_{wf} . If either phases fail, σ_i is ultimately returned to Algorithm 1. Otherwise the valid σ_{wf} is returned to Algorithm 2 and the next component gets tested. After all components have been processed, Algorithm 2 checks whether the disjoint union of σ_c and σ_w equals the total heap layout σ_i . If true, success is reported, by returning the empty set, otherwise the failed test case σ_i is reported back to Algorithm 1.

Workflow. A step-by-step workflow of [Algorithm 2](#) and [Algorithm 3](#) on the function of [Fig. 1](#), with a concrete test case and a wrong invariant that the testing rejects, is in Appendix C of the supplementary material.

4 SMEAGOL's Specification Language

To this point, we have used the traditional syntax of separation logic to describe our approach, but SMEAGOL uses an extended version of CN for its specifications. This section describes the concrete syntax of that language, and presents a dynamic semantics that describes how this language is used for testing invariants.

4.1 Syntax

We write our specifications in CN and in this section we present the fragment. [Fig. 9](#) presents portion of the CN fragment that SMEAGOL reads and writes. Following Fulminate [\[4\]](#), its first part is the surface syntax of Fulminate (top-level definitions and C functions, here extended with loop invariants), and its second part is the core specification language of Fulminate, to which we add a single construct, wand.

A CN specification is a sequence of *conditions*, each a resource, a constraint, or a let binding. A *resource* condition take $id = p(t_1, \dots, t_n)$ asserts ownership of the memory footprint of the predicate p at the given address and binds id to the mathematical value that p returns. $RW\langle T \rangle(t)$ (Fulminate's RW) is the points-to of a single cell of type T , i.e., $x \mapsto y$ where T is the type of y .

A *constraint* t is a boolean fact over the bound variables. In our benchmarks T is one of `i32`, `u32`, `boolean` or the C structs `s11`, `d11` and `tree`.

A function is specified by a *requires* clause (its precondition) and an *ensures* clause (its postcondition), each a sequence of conditions. The precondition takes the resources that the caller hands over, and the postcondition returns the resources back to the callee.

Specification *functions* are pure, first-order and possibly recursive, and state pure facts such as `AddOne(Lhead)`. *Predicates* take ownership of a set of resources that are defined possibly recursively over the shape of the data structures, and the datatype encoding the corresponding data structure is returned. Thus, the body of a predicate always ends in a return.

Conditions are sequenced with the `*` operator, which is expressed by the keyword *take*. A specification is read left to right, with every name in scope and a wand must come after the takes that bind the names it mentions.

The ghost variables correspond to the names bound by *take*, and the conditional *if c then ϕ else ϕ* in [Fig. 7](#) is only available inside a predicate body, not directly in the annotation of the invariant. CN supports verification, but we add a wand syntax in CN only for the purpose of testing. `wand( $spec_a$ ,  $spec_b$ )`; is resource a condition whose two components are sequences of conditions. We require every *take* in $spec_a$ and $spec_b$ to explicitly specify the underlying layout, which is then used for postulation and borrow.

Mapping to CN — Example. The specification of the function in [Fig. 1](#) is written with $\exists$, $*$, whereas CN's syntax does not include either of these symbols.

[Fig. 10](#) shows a simplified version of the function (adds 1 instead of n) and its annotated specifications in CN. The list l becomes the CN datatype `List`, and `lrep( $l$ ,  $h$ )` becomes `Lrep( $h$ )`, so the precondition `lrep( $l$ ,  $h$ )` becomes the clause `take Lhead = Lrep( $h$ )`; with `Lhead` playing the role of l .

The function `(map( $\lambda x.x + 1$ )l)` becomes the CN function `AddOne(Lhead)`, and the postcondition becomes `take Lreturn = Lrep(return)`; `Lreturn == AddOne(Lhead)`; . In the invariant, the star $C_2 := lrep(ls, haddr)$ is the single clause `take Lcurr = Lrep(curr)`; : the existential

```

393 c_function_def      := cty f(cty1 id1, ..., ctyn idn) /*@ requires conditions ensures conditions' @ * / block
394 loop                := while(e) /*@ inv conditions @ * / block
395 function_spec       := /*@ requires spec1 ensures ret.spec2 @ * /
396 spec                := return t | p(t1, ..., ti) | assert(t); spec | let id = t; spec | take id = spec; spec'
397                    | if(t) {spec1} else {spec2} | wand(speca, specb); spec | ·
398 speca, specb        := take id = spec; spec*; id == t; spec*
399 p                   := RW(cty) | id
400 spec_predicate_def := predicate bty p(bty1 id1, ..., btyk idk) {spec}
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441

```

(3)

Fig. 9. The fragment of CN used for the functions, loop invariants and predicates of the paper, with wand added (highlighted); the complete grammar is in Figure S5 of the supplementary material. p ranges over resource predicates (Lrep, Trep, and so on) and RW, t over pure terms, e over C expressions and *block* over C blocks, which may contain a *loop*. $\cdot$ is the empty spec, and $spec^*$ is zero or more specs. The specs inside a wand are wand-free, and a wand does not occur in $spec_1$ of a *function_spec*.

$\exists ls$ is the name Lcurr bound by the take, and $\exists haddr$ needs no extra name because the address is just the program variable curr and CN's take semantics bind the addresses to their underlying values. The bound $\forall e \in ls. minI32 \leq e + 1 \leq maxI32$ of the template becomes the pure clause allSafeToIncrement(Lcurr);. The wand $C_1 := lrep(map \lambda ls, haddr) \multimap lrep(map \lambda l, h)$ becomes wand(*antecedent*, *consequent*);: the antecedent take Lcurr2 = Lrep(curr); Lcurr2 == AddOne(Lcurr); names the suffix after the loop has processed it, and the consequent take Lh1 = Lrep(h); Lh1 == AddOne(Lhead); states that the whole list from h is then the incremented original list. The equations Lcurr2 == . . . and Lh1 == . . . are the layout of the wand's sides: they give SMEAGOL the pure value that each side's take must have.

```

419 1 struct slist* addOne(struct slist* h)
420 2 /*@ requires take Lhead = Lrep(h);
421 3     allSafeToIncrement(Lhead);
422 4     ensures take Lreturn = Lrep(return);
423 5     Lreturn == AddOne(Lhead); @*/
424 6 {
425 7     struct slist *curr = h;
426 8     while (curr != NULL)
427 9     /*@ inv take Lcurr = Lrep(curr);
428 10         allSafeToIncrement(Lcurr);
429 11         wand(take Lcurr2 = Lrep(curr);
430 12             Lcurr2 == AddOne(Lcurr);,
431 13             take Lh1 = Lrep(h);
432 14             Lh1 == AddOne(Lhead));; @*/
433 15 {
434 16     curr->head = curr->head + 1;
435 17     curr = curr->tail;
436 18 }
437 19 return h;
438 20 }
439
440
441

```

Fig. 10. A simplified version of the function in Fig. 1 (adds 1 instead of n) with its contract and loop invariant in CN, over the definitions of Figure S6 of the supplementary material. The star C_2 is take Lcurr = Lrep(curr); and the wand C_1 is the wand($\cdot$. . .) clause.

4.2 Dynamic Semantics for testing the Wand

This section extends the dynamic semantics of the specification language of Fulminate [4] with the wand construct in Fig. 9. Algorithm 3 implements these semantics and we refer to its steps while describing the rules. We present a brief description of the judgement Fulminate uses and how we extend it to support the wand construct. Appendix E of the supplementary material follows the environment of Fulminate step by step on a concrete test. The judgement of Fulminate is $P, E_g, H, incdec, sd \vdash \langle spec, V, R_1, R_2, G \rangle \longrightarrow \langle spec', V', R'_1, R'_2, G' \rangle$ where P is the program, which supplies the predicates, pure functions and functions under test. E_g maps each global variable to its address. H is the heap, a finite partial map from addresses to values; a specification only reads it, so it never changes. sd is the stack depth of the function whose specification runs, and $incdec$ says whether ownership is moving up the stack on a function call (a requires) or down, on a function return (an ensures). V is the pure-value environment from identifiers to the R-values. let and take add bindings to the top of V , a predicate call pushes a mapping of its formal parameters and return pops it. R_1 and R_2 are the set of ambient resources: R_1 contain the set of addresses currently used by the function under test, and R_2 contain the ones not used yet and are to be framed. G is the ghost ownership map from each address to the stack depth that owns it.

We first formalize the judgements corresponding to the wand, then briefly describe the components. After that we walk through the execution semantics with a running example.

$$\begin{aligned} \Gamma_{\text{pos}} \vdash \langle spec, V, R_{1\text{pos}}, R_{2\text{pos}}, posmap \rangle &\longrightarrow_{\text{pos}} \langle spec', V', R'_{1\text{pos}}, R'_{2\text{pos}}, posmap' \rangle, \\ \Gamma_{\text{bor}} \vdash \langle spec, V, G \rangle &\longrightarrow_{\text{bor}} \langle spec', V', G' \rangle, \end{aligned}$$

where

$$\begin{aligned} \Gamma_{\text{pos}} &= P, E_g, H, sd, R_1, G, \\ \Gamma_{\text{bor}} &= P, E_g, H, dec, sd, R_1, posmap. \end{aligned}$$

The footprint is inferred in two stages, first by postulating the antecedent (updating their ambient resources in $R_{1\text{pos}}, R_{2\text{pos}}$) and then borrowing from the antecedent resources, and/or R_1 to satisfy the consequent. Fulminate processes a loop invariant by decrementing the stack depth sd of the cells visited in R_1 to efficiently check for memory overlap, then retag them after testing the loop. Since the wand construct may define a cell n that has already been consumed and tagged to $sd - 1$, we require checking ownership at both $G(n) \in \{sd, sd - 1\}$. Borrowing ultimately infers the footprint, and therefore retags the footprint resources in G to $sd - 1$. The WAND rule defined below is the entry point of this construct. The conclusion of WAND keeps V , R_1 and R_2 , so the bindings made inside the wand aren't visible to the other components, and the ambient resources from R_1 in the inferred footprint are tagged at a lower depth and recorded in the updated G .

The antecedent A and the consequent B are run by two judgments, one for each mode, whose components are only the ones that the mode uses. Both update the pure-value environment V , as variables of A and B share the same scope.

Everything on the left of the turnstile is read-only. In mode pos, G is read-only, so the postulate phase never retags a cell; in mode bor, $posmap$ is the result of the postulate phase, and dec is the $incdec$ of an ensures, because the consequent describes memory that is given back.

In pos, V holds the special value $stop_ptr$, a boolean to indicate the starting address of a postulated memory predicate. $R_{2\text{pos}}$ is the set of ambient resources that are available for postulating; it is initialized to R_1 . $R_{1\text{pos}}$ is set of addresses that the antecedent has postulated, which are moved from $R_{2\text{pos}}$; it is empty at the beginning. $posmap$ is a partial map from a pair (id, n) to a value, where id is a take of A and n is its stop pointer, the first address that the take owns. The value $posmap(id, n)$ is the layout of the data structure that the take defines at n in the postulated heap.

The stop pointer is not known before the first cell is owned, so the second component is initialized with the placeholder *dummy*. It is empty at the start. Two notations are used for the maps. $(M, k \mapsto w)$ is the map M in which k is bound to w . $M \oplus \{k \mapsto w\}$ is the same update for *posmap* and V , except that it adds the binding only if k is not bound in M yet. $M \setminus \{k \mapsto w\}$ removes the binding.

We follow one test case of the loop invariant of `insertEnd` from here on, and state after a suitable rule application, what resources are updated and how Algorithms 2 and 3 are connected to the rules. The program P inserts a node holding the integer *value* at the end of the list. The full annotated version along with a main function is in Appendix E of the supplementary material. We attempt to infer the wand footprint of the invariant

```
take Lcurr = Listrep(curr);
wand(take Lcurr1 = Listrep(curr); Lcurr1 == insEnd(Lcurr, value); ,
      take Lh = Listrep(h); Lh == insEnd(Lhead, value); )
```

where *Lhead* is the list that the precondition took at h . Let a test heap be: $a_1 \mapsto (1, a_2)$, $a_2 \mapsto (2, a_3)$ and $a_3 \mapsto (3, \text{nil})$, with $h = a_1$ and *value* = 4. We test the invariant when $\text{curr} = a_3$, so *Lhead* = $[1, 2, 3]$ and *Lcurr* = $[3]$. The precondition owns every cell reachable from h at the stack depth *sd* of `insertEnd`, so $G(a_1) = G(a_2) = G(a_3) = \text{sd}$ and $R_1 = \text{RW}(a_1) * \text{RW}(a_2) * \text{RW}(a_3)$.

The first component, `take Lcurr = Listrep(curr)`, is wand-free, so Algorithm 2 calls CONSUME, which walks the list at *curr*: it reads a_3 , binds *Lcurr* to $[3]$ and retags the cell, $G(a_3) = \text{sd} - 1$, while a_1 and a_2 stay at *sd*. The second component is a wand, so the algorithm calls WAND instead (the **else** branch of Algorithm 2). The rules below define what that call does, and after each group of rules we say what it does on this test.

WAND has one hypothesis for each mode. The first postulates A in mode *pos*: updates $(V, \text{stop_ptr} \mapsto \text{true})$, and assigns $R_{1\text{pos}} = \text{emp}$, $R_{2\text{pos}} = R_1$ and *posmap* to be empty. It returns the extended environment V' , the postulated cells $R_{1\text{pos}}$ and the binding of the stop address with its contents in *posmap*. The second processes the consequent B in mode *bor* from V' , with the *posmap* that the first returned. It returns the ownership map G' where the cells of the footprint are tagged at $\text{sd} - 1$ which were tagged at *sd* in G . The conclusion keeps V , R_1 and R_2 and replaces G by G' ; the rest, V' , V'' , $R_{1\text{pos}}$, $R_{2\text{pos}}$ and *posmap*, is dropped. The consequent keeps walking through the cells that A postulated, including the cells below a stop, and evaluates any asserts on them. A postulated cell that B never reaches is dropped with *posmap*, because A is allowed to postulate more memory than B uses. The first hypothesis is the call of POSTULATE in Algorithm 3: it runs A in mode *pos* from $R_{1\text{pos}} = \text{emp}$, $R_{2\text{pos}} = R_1$, V extended with $\text{stop_ptr} \mapsto \text{true}$, and an empty *posmap*. The second hypothesis is BORROWINNER and BORROWOUTER: it runs B in mode *bor*, and the cells it tags in the process constitute the footprint σ_{wf} that WAND returns.

$$\frac{\Gamma_{\text{pos}} \vdash \langle A, (V, \text{stop_ptr} \mapsto \text{true}), \text{emp}, R_1, \emptyset \rangle \xrightarrow{*}_{\text{pos}} \langle \cdot, V', R_{1\text{pos}}, R_{2\text{pos}}, \text{posmap} \rangle \quad \Gamma_{\text{bor}} \vdash \langle B, V', G \rangle \xrightarrow{*}_{\text{bor}} \langle \cdot, V'', G' \rangle}{P, E_g, H, \text{indec}, \text{sd} \vdash \langle \text{wand}(A, B); \text{spec}, V, R_1, R_2, G \rangle \xrightarrow{*} \langle \text{spec}, V, R_1, R_2, G' \rangle} \text{WAND}$$

Mode pos: The Antecedent. A take of A written in the annotation, `take id = spec; spec1`, and pinned by an equation `id == e` in *spec₁*, which the rule WAND_TAKE_POS consumes. It removes the equation from *spec₁*, i.e., $\text{spec}'_1 = \text{spec}_1 \setminus (\text{id} == e)$, so the take is not pinned twice and binds *id* to the result of evaluating e in V . The $(\text{id}, \text{dummy}) \mapsto \llbracket e \rrbracket$ is added to *posmap*: the stop pointer of the take is not known yet, hence set to *dummy*. The value is pinned by the evaluating of e , which is a pure function in P . The hypothesis then steps *spec* to *spec'* under the updated V' and

$posmap'$, and the conclusion steps the take to take $id = spec'$ followed by $spec'_1$. In the example, $spec$ is $Listrep(curr)$ and $spec_1$ is $Lcurr1 == insEnd(Lcurr, value)$: the rule binds $Lcurr1$ to $insEnd([3], 4) = [3, 4]$, adds $(Lcurr1, dummy) \mapsto [3, 4]$ to $posmap$ and sets $stop_ptr$ to true.

$$\frac{\begin{array}{c} (id == e) \in spec_1 \quad spec'_1 = spec_1 \setminus (id == e) \\ V' = (V \oplus \{id \mapsto \llbracket e \rrbracket\}, stop_ptr \mapsto true) \quad posmap' = posmap \oplus \{(id, dummy) \mapsto \llbracket e \rrbracket\} \\ \Gamma_{pos} \vdash \langle spec, V', R_{1pos}, R_{2pos}, posmap' \rangle \longrightarrow_{pos} \langle spec', V'', R'_{1pos}, R'_{2pos}, posmap'' \rangle \end{array}}{\Gamma_{pos} \vdash \langle take\ id = spec; spec_1, V, R_{1pos}, R_{2pos}, posmap \rangle \longrightarrow_{pos} \langle take\ id = spec'; spec'_1, V'', R'_{1pos}, R'_{2pos}, posmap'' \rangle} \text{WAND_TAKE_POS}$$

When $spec$ is a call of a predicate, the rule $WAND_PRED_POS$ is fired, which jumps from the call straight to the body of p and pushes the map of the formal parameters on V . In our working example, the call updates V to $p \mapsto a_3$, and inlines the body of $Listrep$: $take\ H = RW<slist>(p); take\ T = Listrep(H.tail); return\ Cons\{Head : H.head, Tail : T\}$.

$$\frac{P(p) = \text{predicate } bty\ p(bty_1\ id_1, \dots, bty_k\ id_k)\{spec\} \quad V' = (id_1 \mapsto v_1, \dots, id_k \mapsto v_k) :: V}{\Gamma_{pos} \vdash \langle p(v_1, \dots, v_k), V, R_{1pos}, R_{2pos}, posmap \rangle \longrightarrow_{pos} \langle spec, V', R_{1pos}, R_{2pos}, posmap \rangle} \text{WAND_PRED_POS}$$

$WAND_ASSERT_POS$ checks every assert that the walk reaches, must be true.

$$\frac{\text{runtime_check}(v = true)}{\Gamma_{pos} \vdash \langle \text{assert}(v); spec, V, R_{1pos}, R_{2pos}, posmap \rangle \longrightarrow_{pos} \langle spec, V, R_{1pos}, R_{2pos}, posmap \rangle} \text{WAND_ASSERT_POS}$$

When a cell ownership primitive RW is matched, either $WAND_RW_POS_STOP$, or $WAND_RW_POS$ is fired. Both postulate the cell n if it is tagged in $G(n) \in \{sd, sd - 1\}$ and available in R_{2pos} . Then n is removed from R_{2pos} and added to R_{1pos} . The rule $WAND_RW_POS_STOP$ is fired if $stop_ptr$ is true, and n is the first cell that the take owns. The rule replaces $dummy$ in the most recently added entry in $posmap$ with n , and updates $stop_ptr$ in V to false, so that the cells below it, and the calls that the body makes, do not record stops. w is returned as it is the underlying value pinned by the evaluation that was recorded in $posmap$.

$$\frac{\begin{array}{c} \text{runtime_check_imp}(G(n) \in \{sd, sd - 1\}) \quad R_{2pos} = R'_{2pos} * RW(n) \\ R'_{1pos} = R_{1pos} * RW(n) \quad (stop_ptr \mapsto true) \in V \quad (id, dummy) \mapsto w \in posmap \\ posmap' = (posmap \setminus \{(id, dummy) \mapsto w\}) \oplus \{(id, n) \mapsto w\} \\ V' = \cdot :: (V, stop_ptr \mapsto false) \end{array}}{\Gamma_{pos} \vdash \langle RW(T)(n), V, R_{1pos}, R_{2pos}, posmap \rangle \longrightarrow_{pos} \langle return\ w, V', R'_{1pos}, R'_{2pos}, posmap' \rangle} \text{WAND_RW_POS_STOP}$$

If $stop_ptr$ is false, $WAND_RW_POS$ is fired. $posmap$ does not change, and the contents at $H(n)$ are returned.

In the example, $RW(a_3)$ passes the ownership check, since $G(a_3) = sd - 1$, and moves from R_{2pos} to R_{1pos} . Since $stop_ptr$ is true, $WAND_RW_POS_STOP$ applies: the entry becomes $(Lcurr1, a_3) \mapsto [3, 4]$, $stop_ptr$ becomes false, and $[3, 4]$ is returned.

$$\frac{\begin{array}{c} \text{runtime_check_imp}(G(n) \in \{sd, sd - 1\}) \quad R_{2pos} = R'_{2pos} * RW(n) \\ R'_{1pos} = R_{1pos} * RW(n) \quad (stop_ptr \mapsto false) \in V \quad H(n) = n' \quad V' = \cdot :: V \end{array}}{\Gamma_{pos} \vdash \langle RW(T)(n), V, R_{1pos}, R_{2pos}, posmap \rangle \longrightarrow_{pos} \langle return\ n', V', R'_{1pos}, R'_{2pos}, posmap \rangle} \text{WAND_RW_POS}$$

WAND_RETURN_POS follows from the previous rules. It pops the first element from V which marks the end of the predicate, as predicates end with return, and binds id to the returned value. For a *stop_ptr* cell, the postulated contents are bound to the return by WAND_RW_POS_STOP, which are then updated in V corresponding to the id .

$$\frac{V = V_f :: V_0 \quad V' = V_0 \oplus \{id \mapsto v\}}{\Gamma_{\text{pos}} \vdash \langle \text{take } id = \text{return } v; \text{spec}_1, V, R_{1\text{pos}}, R_{2\text{pos}}, \text{posmap} \rangle \longrightarrow_{\text{pos}} \langle \text{spec}_1, V', R_{1\text{pos}}, R_{2\text{pos}}, \text{posmap} \rangle} \text{WAND_RETURN_POS}$$

Mode bor: The consequent B. After the postulation is complete, we try to construct a layout for B in the bor mode. We split the rules for the take of B . It can be a call to a predicate or a primitive RW. A call $\text{take } id = p(v_1, \dots, v_k)$; spec_1 fires the rule WAND_PRED_BOR that jumps to the body of p , with the formal parameters pushed on V , similar to WAND_PRED_POS. In the running example, B is $\text{take } Lh = \text{Listrep}(h)$, which WAND_PRED_BOR steps into the body of Listrep on a_1 ; the calls on a_2 and a_3 in the body are stepped into in the same way.

$$\frac{P(p) = \text{predicate } \text{bty } p(\text{bty}_{i_1} id_1, \dots, \text{bty}_{i_k} id_k) \{ \text{spec} \} \quad V' = (id_1 \mapsto v_1, \dots, id_k \mapsto v_k) :: V}{\Gamma_{\text{bor}} \vdash \langle \text{take } id = p(v_1, \dots, v_k); \text{spec}_1, V, G \rangle \longrightarrow_{\text{bor}} \langle \text{take } id = \text{spec}; \text{spec}_1, V', G \rangle} \text{WAND_PRED_BOR}$$

A take $\text{take } id = \text{RW}(T)(n)$; spec_1 has two cases. If posmap has an entry $(id_a, n) \mapsto n'$ then WAND_RW_POST_BOR is fired. The cell is one that A postulated and n' is the contents that A recorded for it; the cell is not retagged, since it belongs to the postulated memory and hence not part of the footprint.

$$\frac{(id_a, n) \mapsto n' \in \text{posmap} \quad V' = \cdot :: V}{\Gamma_{\text{bor}} \vdash \langle \text{take } id = \text{RW}(T)(n); \text{spec}_1, V, G \rangle \longrightarrow_{\text{bor}} \langle \text{take } id = \text{return } n'; \text{spec}_1, V', G \rangle} \text{WAND_RW_POST_BOR}$$

Otherwise WAND_RW_BOR is invoked. The premise $R'_1 = R_1 * \text{RW}(n)$ records the cell as used. The ghost mapping is checked at sd , the contents are those of the heap, $H(n) = n'$, and the cell is retagged to $sd - 1$. In both cases, the next step is a return: $\text{take } id = \text{return } n'$, followed by additional conditions (spec_1) and an empty map pushed on V that the return rule will pop. The cells that the rules retag in G are exactly the footprint of the wand that Algorithm 3 returns. In the example, a_1 is not in posmap , so WAND_RW_BOR reads $H(a_1) = (1, a_2)$ and retags it, $G'(a_1) = sd - 1$; the call on a_2 does the same, $G'(a_2) = sd - 1$. The call on a_3 finds $(Lcurr1, a_3) \mapsto [3, 4]$ in posmap , so WAND_RW_POST_BOR applies: a_3 is not retagged, and since posmap associated the whole list $[3, 4]$ with corresponding to a_3 , the value of the take at a_3 is $[3, 4]$ and not the list $[3]$ that the real cell holds.

$$\frac{\begin{array}{c} R'_1 = R_1 * \text{RW}(n) \\ \text{runtime_check_imp}(G(n) = sd) \quad H(n) = n' \quad G' = G[n \mapsto \text{dec}(sd)] \quad V' = \cdot :: V \end{array}}{\Gamma_{\text{bor}} \vdash \langle \text{take } id = \text{RW}(T)(n); \text{spec}_1, V, G \rangle \longrightarrow_{\text{bor}} \langle \text{take } id = \text{return } n'; \text{spec}_1, V', G' \rangle} \text{WAND_RW_BOR}$$

WAND_ASSERT_BOR checks every assert, on addresses and values, as WAND_ASSERT_POS does. The walk of B does not stop at the structure that A postulated: it keeps recursing through the rest of every body, so the asserts of the predicates of B are evaluated on the postulated cells too. Additional rules for the let construct are omitted here for brevity but are consistent with the let rule of Fulminate with the judgements catered to the two modes of wand.

$$\frac{\text{runtime_check}(v = \text{true})}{\Gamma_{\text{bor}} \vdash \langle \text{assert}(v); \text{spec}, V, G \rangle \longrightarrow_{\text{bor}} \langle \text{spec}, V, G \rangle} \text{WAND_ASSERT_BOR}$$

WAND_RETURN_BOR pops the first element of V and binds id with $\oplus$, so a value that is already bound is kept. In the example, the returns build $\text{Cons}(2, [3, 4]) = [2, 3, 4]$ for the call on a_2 and $\text{Cons}(1, [2, 3, 4]) = [1, 2, 3, 4]$ for the call on a_1 , which binds Lh .

$$\frac{V = V_f :: V_0 \quad V' = V_0 \oplus \{id \mapsto v\}}{\Gamma_{\text{bor}} \vdash \langle \text{take } id = \text{return } v; \text{spec}_1, V, G \rangle \longrightarrow_{\text{bor}} \langle \text{spec}_1, V', G \rangle} \text{WAND_RETURN_BOR}$$

The equation $id == e$ of B is a check (WAND_CHECK_BOR): the value of id in V is compared with $\llbracket e \rrbracket$. If they differ, no rule applies and the test fails. Every take binds its own identifier in V , so the equations of several takes of B are checked independently. In the example, $Lh == \text{insEnd}(Lhead, \text{value})$ compares $[1, 2, 3, 4]$ with $\text{insEnd}([1, 2, 3], 4) = [1, 2, 3, 4]$, so it holds. Had the antecedent been the wrong one like: $Lcurr1 == Lcurr$ (does not include the value to be inserted at the end), the entry would be $(Lcurr1, a_3) \mapsto [3]$, Lh would be $[1, 2, 3]$, and the check would fail. The conclusion of WAND keeps V , R_1 and R_2 and replaces G by G' , which differs from G at a_1 and a_2 . The footprint of the wand is $\sigma_{wf} = \{a_1 \mapsto (1, a_2), a_2 \mapsto (2, a_3)\}$; back in Algorithm 2, it is disjoint from $\sigma_c = \{a_3 \mapsto (3, \text{nil})\}$, and $\sigma_c \uplus \sigma_w = \sigma_i$, so the test passes.

$$\frac{(id \mapsto v) \in V \quad \text{runtime_check}(v = \llbracket e \rrbracket)}{\Gamma_{\text{bor}} \vdash \langle id == e; \text{spec}, V, G \rangle \longrightarrow_{\text{bor}} \langle \text{spec}, V, G \rangle} \text{WAND_CHECK_BOR}$$

Complexity. We analyze the complexity of Algorithm 2 which calls Algorithm 3 as a routine.

Let $N = |\sigma_i|$ be the number of cells of the test heap. Every layout built by Algorithm 2 and Algorithm 3 is a subset of σ_i , so it has at most N cells. Each cell is read once and added to a layout after a clash check against the cells already in it.

For a test case, CONSUME reads the cells of the wand-free components of inv into σ_c . The clash check is done by checking the stack depth sd and there will be at most N cells scanned, bounded by $O(N)$. A wand w is tested by Algorithm 3: POSTULATE builds σ_{wa} with k_a cells, BORROWINNER builds σ_{wb} with k_b cells. The clash check proceeds by checking every cell consumed so far, as we can have resources owned at depths sd and $sd - 1$, so a wand costs $O(k_a^2 + k_b^2) \in O(N^2)$, since both k_a and k_b can have a maximum of N cells. The disjointness test of Algorithm 2 with σ_{wf} is also done by looking at the stack depth, and hence is upper bounded by $O(N)$. Therefore, Algorithm 2 is upper bounded by $O(|\sigma|N^2)$, where $|\sigma|$ is the number of test cases.

5 Evaluation

Our evaluation investigates three key research questions about our approach:

- **RQ1:** How effective is SMEAGOL? Is it able to infer correct loop invariants for a diverse set of benchmarks corresponding to three data structures across different LLMs?
- **RQ2:** How useful is SMEAGOL's testing framework? How often is it able to detect semantically incorrect invariants?
- **RQ3:** How does the testing framework perform with respect to large test cases?

All of our experiments were carried out on an Apple M2 Max Macbook Pro with 32GB RAM with custom API keys for the different LLMs. The LLMs used are Opus (Anthropic's highest reasoning model) [19], Haiku, Sonnet, Kimi K3 (Open Weight) [39] and DeepSeek [44].

All models are re-prompted to a maximum of 10 times per loop other than the results in Table S9 of the supplementary material for which we report difference in results by changing the number of re-prompts from 10 through 20 and 40. All the tests in RQ1 and RQ2 are generated with the default setting which are 100 tests per function invocation and the size of tests ranging from 1 through 25.

5.1 Benchmark Construction

To evaluate our approach, we developed a suite of specifications for a set of programs of varying complexity for the data structures — Singly Linked Lists, Doubly Linked Lists and Binary Trees. Our suite includes benchmarks with two loops (`addTwo`, `addListThroughDll`, and `bstDelete`), and a benchmark with a nested loop (`addListCount`). Two benchmarks (`bstReturnNode` and `returnSumNode`) have magic wand in the postcondition. Magic wand naturally occurs in the postcondition if a node that may occur in the middle of the data structure is returned: i.e., not the first or last node of a list, and not a root or leaf node of a binary tree.

The above-mentioned benchmarks (excluding `bstDelete`) have been constructed by the authors. Others are sampled from related work [22, 23] on heap specification inference. Two benchmarks (`reverseList` and the first loop of `addListCount`) do not require the magic wand. All the other loop invariants in our benchmarks require the wand connective.

5.2 Invariant Template

As a part of our inference procedure, we provide an invariant template to the LLM. The contents of all the pointer variables that change in the loop are captured in the template by the set of ghost variables g . The names of these variables (id) are associated with their values and type by the CN take syntax: $take\ id = RW\langle T \rangle(n)$, which maps to $\exists id, n \mapsto id$ in standard separation logic.

By including this information in the template, we explicitly bind each changing variable to its value, thereby preventing information loss. We further provide additional propositional properties Φ on these variables, such as the range of values the contents of the variable can support (e.g. `allSafeToIncrement(Lcurr)` function in Fig. 10). These are additional properties that would be necessary to formulate a proof of the invariant.

The template does not include the function that describes the underlying contents of the data structure. As an example, a valid template for the loop invariant in Fig. 10 includes only `Lcurr` and `allSafeToIncrement(Lcurr)`. The entire memory layout and the contents of the data structure are left to be inferred. The template (g, Φ) captures all essential information to constrain the correctly inferred invariants to the set of reachable states of the loop. Algorithm 3 further rejects an invariant if the postulation of an antecedent fails, hence not allowing trivially true wands that fill in vacuous memory. We hypothesize that these restrictions limit the space of overapproximate invariants. This is in contrast to related works like Sling [22] which does not work with a template and allows inferring extraneous ghost variables that often result in an overapproximate invariant. We have not formalized this claim, and it is in scope for future work.

5.3 RQ1: Effectiveness of SMEAGOL

Table 1 presents the results of SMEAGOL for every benchmark with the best performing seed of each LLM. We ran experiments for each LLM with three seeds to control for their inherent randomness. The complete results of these tests can be found in Appendix F of the supplementary material. Those results also contain token consumption and cost per benchmark averaged across seeds.

At the time of writing the paper, Opus is Claude's best reasoning model, and we configured its reasoning effort at the highest level. We also experimented with Claude's lower-tier models Sonnet and Haiku, and the open-weight models Kimi K3 and DeepSeek v4-pro. Where not prohibited by token budget and time constraints, we ran models with the default (i.e. maximum) reasoning level. Due to constraints, we ran Kimi with the reasoning set to the lowest level.

Opus is the best performing model for our set of benchmarks, and passes all invariants across all seeds. Timing information in Table 1 is reported with respect to Opus.

Table 1. Best-performing seed per model with at most 10 reprompts per loop (Opus 36, Kimi 41, Sonnet 48, DeepSeek 68, Haiku 57; best = most benchmarks solved, ties broken by fewer reprompts). Trials = 1 + reprompts (at most 11). For benchmarks with a single loop the entry is the number of trials (or × if it stayed unresolved); for benchmarks with two loops it is a tuple (trials for loop 1, trials for loop 2), where × marks a loop that stayed unresolved and N/A a second loop that was not reached because loop 1 failed. Opus is shown in dark green. Total time (inference and testing) and testing time are for Opus seed 36 only.

Benchmark	Opus	Kimi	Sonnet	DeepSeek	Haiku	Total Time (s)	Testing Time (s)
addConditional	1	1	1	1	1	6	0.8
addInRange	1	2	1	1	×	24	0.8
addListCount	(1, 1)	(1, 3)	(1, 1)	(1, 1)	(×, N/A)	92	1.9
addListThroughi	1	1	1	2	1	19	0.6
addOne	1	1	1	1	1	5	0.7
addTwo	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(10, 1)	87	1.4
appendList	1	2	1	1	×	17	0.7
checkTwoLists	1	1	1	1	4	28	0.6
copyList	2	5	1	4	×	225	0.8
copyListFilter	1	2	2	1	×	180	0.7
deleteEndleast2	1	1	1	1	1	24	0.7
findList	1	1	1	1	1	18	0.7
insertEnd	1	1	1	1	1	3	0.6
repeatInsertEnd	2	4	1	3	×	192	0.8
returnSumNode	1	1	1	1	5	24	0.6
reverseList	1	1	1	1	×	19	0.6
addListThroughiDll	(1, 1)	(1, 10)	(1, 1)	(1, 3)	(×, N/A)	312	1.3
dllAddOneWand	1	1	1	1	3	5	0.7
dllCopy	2	5	2	6	×	207	0.9
bstAddRight	1	1	1	2	1	10	2.3
bstDelete	(1, 1)	(1, 2)	(×, N/A)	(×, N/A)	(×, N/A)	206	13.4
bstDeleteMin	1	5	1	2	3	93	3.2
bstFind	1	1	1	1	6	26	2.3
bstReturnNode	1	1	1	1	1	10	2.2
findMinKey	1	1	1	1	1	19	2.5
findMinValue	1	1	1	1	1	16	2.6
insertBSTNoCheck	1	1	1	1	4	23	2.0

We manually inspected the passing invariants proposed by each LLM in Table 1 and found that all the passing invariants are correct and strong (not overapproximate). LLMs (DeepSeek and Haiku) inferred *recursive predicates*, which violate the given prompt rules, only three times across all seeds. Our benchmarks did not require a left nested wand, and no LLM proposed a nested wand for any invariant.

From the timing data, we observe that most of the benchmarks take less than a second to test, owing to the efficiency of the testing algorithm. `bstDelete`, with multiple loops and two annotated auxiliary functions whose specifications must also be tested, required the most time to test.

We observe that Opus proposed passing invariants for 24 benchmarks in the first try and required only 1 re-prompt for each of the remaining benchmarks. Kimi also performed well and passed every benchmark, but required more re-prompts than the other high-end models. Sonnet required the least number of re-prompts for the cases where it was successful, and was the only LLM to propose a passing invariant for `copyList` in the first try, however it failed to propose a

correct invariant for `bstDelete`. Both DeepSeek and Sonnet pass all but `bstDelete`, for which neither managed to propose a correct invariant for the first loop.

Haiku, the base model of Claude, performed poorly in comparison to the others. It solved only 17 out of 27 benchmarks, failing several benchmarks that most of the other models in our tests did not need re-prompting for. Since Haiku’s reasoning strength is the lowest of the LLMs considered in our experiments, we ran another set of experiments with the second-best-performing seed of Haiku (Seed 58), increasing the number of re-prompts to 20 and 40. We present the results of that experiment and accompanying discussion in Table S9 of the supplementary material.

For all cases, we observe that the different analysis units of SMEAGOL work together to guide the LLM towards the correct invariant. The higher-reasoning models tend to use the feedback provided by SMEAGOL to repair the proposed invariants the most quickly. Furthermore, all models were able to correctly follow the novel syntax of the template, and we observed only 4 syntax errors for all LLMs across all seeds.

5.3.1 Extra Predicates and Pure Functions Proposed. We require the LLM to *not* propose any *recursive* predicates, but additional non-recursive predicates and optionally-recursive pure functions may be required to write a complete invariant.

Functions are used in the invariant to describe the value layout of the data structure. An invariant may require additional functions which are not in the hypothesis space because they are not in the pre- or post-conditions. For example, the `addTwo` benchmark describes the layout of the output list with a *pure* function `AddTwo` that adds 2 to its input. However, the implementation uses two loops — one after the other — incrementing each element of the list by 1 in each loop. The correct invariant therefore requires an additional *pure* function that adds 1 to its input.

Invariants that use an if-then-else construct to take ownership cannot be written directly in CN’s `inv` annotations and require extra predicates. These are distinct from recursive predicates, which SMEAGOL disallows.

These functions and invariants, as described above, constitute the minimal set of additional information needed to write a correct invariant.

An expert hand-constructed ground truth invariants for all 27 benchmarks. Extra non-recursive predicates were needed for 2 benchmarks (`bstDelete` and `bstDeleteMin`). Six more benchmarks required extra functions. We analyzed the predicates and pure functions inferred by the LLMs for the successfully tested cases and observe that for 2 benchmarks out of the 19 (`addConditional` and `addListThroughIDll`), at least one of the LLMs proposed extra functions. For `addConditional`, one model (Haiku) inferred an extra length function that it did not include in the invariant. For `addListThroughIDll` the LLMs express the list contents with multiple functions, unnecessarily complicating it. This is the only benchmark in Doubly Linked List that traverses in both directions, and coming up with the most efficient function may be less intuitive than the rest.

For the 8 benchmarks, we found that the expert wrote an extra function for `addListThroughi` which was not needed, and which only Haiku included in its proposed invariant. The LLMs other than Sonnet infer more functions than necessary for `dllCopy` arguing the structure of the tail node of the copied list so far. This pattern is also observed for DeepSeek for `copyList`. Lastly, for `bstDelete` and `bstDeleteMin`, where the expert wrote a single predicate to capture the parent-child relationship, the LLMs proposed extra functions and non-recursive predicates. This effectively complicated the reasoning while maintaining correctness.

These results show that while LLMs do sometimes trivially propose unnecessary or overcomplicated predicates and functions to accomplish the tasks, in most cases they do so with the minimal set of necessary functions and predicates, even proposing a more efficient invariant in one case.

Table 2. Opus pass@ k (seed 88, no reprompting). Each cell is the probability that a sample passes, estimated as the fraction of passing samples among the first 5, 10, and all 20 independent attempts. For benchmarks marked * (two loops) a sample is a round: one attempt at loop 1 and, only if it passes, one at loop 2; a round passes only if both loops pass. Only benchmarks with probability below 1 for at least one k are shown; the other 17 benchmarks pass in all samples (probability 1 for pass@5, pass@10 and pass@20).

Benchmark	pass@5	pass@10	pass@20
addListCount *	0	0	0
addTwo *	1	1	0.95
copyList	0.2	0.1	0.1
copyListFilter	0.8	0.9	0.9
repeatInsertEnd	0	0	0
returnSumNode	1	0.9	0.95
addListThroughDll *	0.2	0.2	0.25
dllCopy	0	0	0
bstDelete *	0	0	0.05
bstDeleteMin	0.2	0.2	0.3

5.3.2 *Prompt Rule Violation(s)*. Each LLM response is syntactically evaluated with respect to the prompt constraints and only moved to the testing phase if it passes these checks. We did not observe any prompt violations committed by Opus in our first round of experiments, although one was committed in a later set of experiments and is described in a later section. For the best-performing seeds, Sonnet committed 3, Kimi had 7, DeepSeek had 8, and Haiku had 76.

The most common violation was an unused ghost variable from the template, which can lead to overapproximate or incorrect invariants. All of Kimi’s and Sonnet’s violations were of this kind. Haiku committed this kind of violation 26 times.

Both DeepSeek and Haiku gave responses with illegal characters outside of the annotations, causing CN to fail compilation. Haiku committed this violation 42 times, and DeepSeek 3 times. In once case, DeepSeek violated the prompt rules by inventing a novel proposition. Similarly, Haiku removed a provided proposition for one case, which can lead to an overapproximate or incorrect invariant.

5.3.3 *Pass@ k* . We conducted an additional set of experiments on all the benchmarks with a different Opus seed (Seed 88), where each benchmark was independently run (i.e., without any re-prompts) for k trials. Here k is 5, 10, and 20.

This experiment gauges the randomness of LLMs and assesses the frequency of first-try success. The results are presented in Table 2. Of our benchmarks, 17 pass for every trial and are not reported here. Those 17 also pass in the first try for all seeds (reported in the Supplementary Material).

Benchmarks addTwo, returnSumNode and copyListFilter pass in the first attempt, but do fail for some trials here. For addTwo and copyListFilter, these failures were violations of the prompt rules or failed the template check, with one special character and two misspelled variable names. returnSumNode also failed for one trial, with the proposed invariant violating the semantics of *. Testing successfully detected the error.

The benchmarks addListCount, repeatInsertEnd, and dllCopy failed across all trials and therefore have 0 in each pass@ k entry. This result matches our observations across all seeds for these benchmarks, which all required re-prompting. Comparing these results to the Seed 88 run revealed that these failures all shared the same cause — a CN type checking failure due to a local

and changing pointer variable marked as unchanged. After requiring a re-prompt for all seeds in our first set of experiments, Opus proposed a correct invariant for `copyList` on the first try in some trials. Manual error analysis reveal that the failed cases are also similar to the errors across all seeds.

The third category of benchmarks that fail in most attempts are: `addListThroughIDll`, `bstDelete` and `bstDeleteMin`. Most of the failures of `bstDelete` and `bstDeleteMin` occur due to using incorrect names, a prompt-level failure. This is also consistent with our observations in the analysis of all seeds for Opus. `addListThroughIDll` has the opposite result – i.e., every failure passes prompt-level checks and is eliminated in testing.

To conclude, the results of `pass@k` are similar to the runs across all seeds for most benchmarks – the ones that required re-prompting, also failed here for most trials. The errors that occur are also similar. These results demonstrate the usefulness of SMEAGOL. Even strong reasoning models sometimes propose incorrect invariants, which must be eliminated with syntactic and semantic checks.

5.4 RQ2: Efficacy of the Testing Algorithm

The run-time testing algorithm is the component of SMEAGOL that decides whether a proposed invariant is semantically correct. An invariant can be well-formed, pass the prompt checks and be accepted by the CN front end, and still be wrong: its wand can build the wrong list, its antecedent can describe cells that it does not own, or it can silently stop describing a part of the heap that the loop owns. None of these are caught by a parser or a prompt-level syntactic check of the invariant, so the testing algorithm is the only barrier between a plausible-looking invariant and an accepted one. Tests also produce the concrete counterexample that the next prompt is built from. This RQ asks how well it does that job.

Setup. From our experiments to evaluate RQ1, we gathered 475 invariants that were rejected on semantic grounds. This includes every semantically incorrect invariant from the three seeds each of Opus, Sonnet, Haiku, Kimi and DeepSeek, and the two additional Haiku runs of Seed 58 with 20 and 40 re-prompts, leaving out the attempts that were rejected for syntax errors or for formatting or prompt rule violations.

Kinds of errors caught. We divided the 475 rejected invariants into three categories, based on the reason they were rejected: value mismatches, footprint errors, and other errors. Table S10 of the supplementary material shows how these rejected invariants are distributed among models, seeds, and reasons. It also includes the size of the failing test cases. Value mismatches are the largest group in every seed except DeepSeek 67, footprint errors appear in nine of the fifteen seeds, and one seed (Opus 36) has no rejection at all.

The testing algorithm caught six kinds of semantic error. The first, and by far the most common with 344 of the 475 rejections (72%), is the *value mismatch*: the value that the consequent of the wand constructs from the antecedent does not match the behavior of the program. This can occur, for example, when the antecedent is defined relative to the wrong list or the transformation function that the LLM wrote does not relate the two lists correctly. The second kind is the *uncovered precondition cell* (35 rejections): a cell that the function owns at the checkpoint is unaccounted for, claimed neither by any ordinary take nor by the footprint of any wand. The result is that the invariant silently stops describing part of the memory it started with. The third and fourth kinds are due to overlapping footprints: the real footprint of the consequent overlaps a resource that the invariant takes outside the wand (21 rejections), or two parts of the antecedent claim overlapping addresses (4 rejections). The last kinds are the failures of the loop ownership check (6 rejections)

and of an assertion in a predicate or directly in the annotation (65 rejections), for example an ownership error at a call that the invariant does not cover, or a violated bound proposition.

The first four kinds are errors that are specific to wands, which are found by the testing algorithm contributed by this paper; they account for 404 of the 475 rejections (85%). We further note that the size of the counter examples are small, with the heaps having 2.13 cells on average and the largest one has 12 cells, which results in quick testing time, as observed in our experiments for RQ1 and RQ3.

Discussion. We present two observations that show why the testing algorithm is the key component of SMEAGOL. First, it drives the re-prompting observed in Section 5.3. The 475 counterexamples become the feedback of the next prompt. This information allows the stronger models to repair their proposed invariants in subsequent attempts, acting as a useful signal for the LLM to direct invariant inference. Second, the weaker models rely on this testing information. Across all trials, Haiku received the most feedback from testing, producing 156 rejected invariants in its three seeds and 210 more in the longer runs of Seed 58, against 3 for Opus, 29 for Sonnet, 61 for Kimi and 16 for DeepSeek. Haiku was only to pass only 14 of our 27 benchmarks in its first attempt across all seeds and trials, and never produced a correct invariant for another 8. For the remaining 5 invariants, Haiku was able to produce correct invariants only as a result of testing. Of the benchmarks Haiku passed in the first attempt, it only did so consistently for 7, relying on testing at least some of the time for another 7.

5.5 RQ3: SMEAGOL's testing effectiveness

A key contribution of this work is the implementation of a novel run-time testing algorithm that evaluates whether a given data structure satisfies a loop invariant containing magic wand connectives. Here we evaluate how that testing algorithm performs at scale, on data structures of differing sizes. In its current setting our algorithm is unlikely to encounter large data structures that substantially impact performance. Indeed, in Table S10 of the supplementary material, the maximum number of heap cells of any of our failing test cases was 12. This performance evaluation is an attempt to be thorough and forward-looking toward potential future applications for run-time testing of wand-containing propositions.

5.5.1 Experimental Setup. For each benchmark in our dataset, we implemented a constant function — the Fulminate mechanism for tests without generators — to randomly (under a fixed seed) generate a data structure of a fixed size and provide it as input to the function under test. We chose to write simple generators for the data structures ourselves in order to minimize the overhead incurred by Bennet/Darcy (the CN PBT generator), in an attempt to isolate a single piece of the system. Each benchmark was annotated with our handwritten "ground truth" loop invariants, rather than invariants generated by an LLM.

We measured the run time of each benchmark using hyperfine, a command line interface tool for evaluating performance. We ran each benchmark with hyperfine ten times for a given size and reported the average run time. Each benchmark was tested with data structures with 1, 10, 100, and 1000 entries.

5.5.2 Performance Results and Analysis. Table 3 reports a relevant subset of the results of our performance tests: a representative example for each data structure, an outlier representative of a set of outliers in our dataset, a meaningful example that does not include magic wand, and the average performance for each data structure. The complete set of results can be found in Table S3 of the supplementary material. Because it is difficult to fully isolate our testing algorithm from

Table 3. Performance testing results, average of ten trials for each benchmark and size. Full results can be found in Table S3 of the supplementary material. Benchmarks that take two data structures as input are labeled with *. Benchmarks that include a precondition on the size of the data structure were run at their smallest acceptable size n , and are annotated in the table, as (n) .

Benchmark	Time @ Size 1 (s)	Time @ Size 10 (s)	Time @ Size 100 (s)	Time @ Size 1000 (s)
appendList*	0.257	0.305	0.876	26.696
findList	0.241	0.263	0.570	9.925
reverseList	0.236	0.259	0.515	10.488
sllist average	0.251	0.274	0.568	11.221
dllCopy	0.271	0.306	0.637	12.078
dllist average	0.264	0.298	0.640	12.912
bstDeleteMin ⁽²⁾	0.264	0.267	0.321	0.833
bstFind	0.257	0.268	0.309	0.837
insertBSTNoCheck	0.268	0.281	0.323	0.860
tree average	0.268	0.273	0.322	0.849

the context it inhabits, these results can be interpreted as upper bounds for performance. In particular, these measured performance times include compilation, because SMEAGOL's unit testing mechanism runs this kind of test as an accessory to the compilation process.

There are two primary kinds of outlier in our data. One of these kinds is represented in Table 3 by the appendList result set. Some of the benchmark programs in our dataset took as input two separate lists. For these programs, we chose to generate two lists of the given size (i.e. two lists, each of size 10, for 20 rather than 10 total elements). As a consequence of this choice, the measured performance for this kind of benchmark diverged from the otherwise narrow range of results for our list benchmarks.

The other kind of outlier has only a single example in our dataset, represented by the reverseList result set. The correct invariant for reverseList does not include a magic wand connective, and so does not invoke the wand testing algorithm. These results serve as a meaningful baseline for the overhead of testing large data structures with SMEAGOL.

The performance data gathered seem to support our complexity analysis of our testing algorithm. Exploratory testing (i.e. for one trial per size of one benchmark at larger sizes not covered in our dataset) seems to result in run times that show only small variance from quadratic lines of best fit generated from our data; however, we have not completed a full statistical analysis of our performance testing data. A graph plotting the average run times observed in our testing, fitted with a quadratic curve, based on the complexity evaluation of our testing algorithm, can be found in of the supplementary material.

6 Threats to Validity

We cannot guarantee the absence of a **data leak** — that is, we cannot guarantee that the models that we used in our experiments did not use data gathered from us to propose correct invariants. We mitigated this risk by not providing the target invariants or complete programs from our benchmarks in any prompts. However, magic wand data is relatively scarce, and SMEAGOL has a novel syntax to output invariants. We explicitly state in the prompt not to consult relevant sources while inferring the invariants and particularly conduct experiments on an open-weight model for transparency.

7 Related Work

7.1 Magic wand Verifier Support

There are several Automated Theorem Provers that support magic wand, including Viper [25], Silicon [33], Prusti [3] and VerCors [5]. An automated verifier needs package and apply annotations to reason about wand-containing formulas. Inferring these annotations and placing them at the appropriate locations is non-trivial.

Blom and Huisman [6] demonstrated a mapping of logical formulae containing magic wand to formulae without it, which resulted in heavy annotation overhead, which are non-trivial to infer. Silicon does some heuristic-driven automated inference of annotations, but the procedure is not complete and requires user assistance. Prusti is a static verifier for Rust that largely automates these additional annotations by translating the specification and code into Viper. It has some support for magic wands through a novel pledge-style specification to capture the future state. Separation Logic tools such as Iris [20] and VST [11] that rely on interactive theorem provers support magic wands, along with some established lemmas to reason about them.

7.2 Testing of Heap Specifications

Separation Logic specifications have also recently been the subject of research on testing. Fulminate [1, 4] is a run-time testing tool for CN [27] that converts a C + CN program into C code with execution statements corresponding to the specification that can be subject to a test suite or a debugger of your choice. Bennet [2] is a property-based testing tool, for testing preconditions in CN-annotated C programs, that maps CN predicates and specifications into a generator DSL, with further optimizations to improve performance. Their results demonstrate that the generators are general-purpose as they can be adapted to different C programs and are effective at catching bugs in preconditions. There has also been recent work in run-time verification of specifications based on an implicit dynamic frames (IDF) logic [35] — a variant of separation logic — to support the gradual verification of imperative programs [13, 43, 47]. This work offers tighter integration between static verification and run-time testing compared to Fulminate and stronger guarantees than testing (i.e. the code is always being monitored for specification violations), but only works on a simple IDF logic with *write-only* permissions, missing key features such as fractional permissions [7], iterated separating conjunctions [4, 26], and magic wands.

7.3 Specification Inference of the Heap

Inferring the heap portion of a specification as well as the recursive predicates needed to write these specifications has been the subject of decades of research effort. [17, 45] are some of the works on inferring the recursive predicates needed to represent the unbounded data structures for heap-manipulating programs. These predicates are then used in the heap specifications. [45] also presented results of the LLM-generated predicates, which were fairly accurate. Research on inferring heap specifications has ranged over preconditions, post-conditions, and loop invariants. However, research on specification inference has so far focused primarily on the separating conjunction operator. [24] mapped code and heap specifications into a numerical abstraction, to reason about properties like the height of a tree or length of a list. This mapping eliminated the need to query a Separation Logic-style verifier to reason about numeric properties. Their work deals with lists and trees, where the traversed fragment of the tree is loosely represented with *true*. [37] presents a bi-abduction style specification inference on Refined C [32] programs; however, their approach is designed to work on straight-line programs. [10] presents a bi-abductive reasoning framework to infer the anti-frame and frame for specifications. They evaluate on linked lists, and

their framework is primarily applied to infer pre- and post-conditions. [29] is an abductive approach to automatically refine heap specifications; however, the refined specifications correspond mainly to the pure fragment and not the spatial fragment. [28] employs bi-abduction along with join and widen operation to abstract over the shape predicates and infer loop invariants. Their approach infers both the spatial and pure part of the specification, but is limited to the separating conjunction operator. Sling [22] is a Daikon-[15] style dynamic analysis tool for spatial predicates, that infers pre- and post-conditions and loop invariants from traces obtained through appropriate debug points. Their benchmarks cater to a variety of data structures like arrays, singly- and doubly- linked lists, binary search trees, and AVL trees. Their tree benchmarks are mostly recursive. The only work we are aware that infer magic wands is [21]. This is an abductive reasoning framework integrated in Viper which infers the memory predicates as well as the layout, with support for automatic inference of the package and apply annotations. However, the inference is limited to memory safety and not on the entire layout of the heap. SMEAGOL is equipped to test the fully-expressed layout which is known to be undecidable, and gives us a well-tested wand which can be verified in the backend.

7.4 Learning based Specification Inference of Heap Predicates

Learning-based works have primarily focused on inferring numeric invariants ([38, 41, 42]).

We discuss three notable works in this domain that have employed a learning-based approach. [8] present a data-driven framework to automatically verify memory safety properties of heap-manipulating programs. Their novelty is in the machine-learning model that learns specifications from program states. [23] presents a benchmark of heap-manipulating programs along with a neurosymbolic model to generate and prove loop invariants in Separation Logic. This is the first work to deploy a fine-tuned LLM that is trained by repeated unfolding of the shape predicates, which then aids in the loop invariant prediction. Their benchmarks also consist of data structures supported by Sling, with tree programs featuring loops. Since they only support the separating conjunction operator, their tree programs are especially cumbersome to verify, with extraneous predicates, non-trivial existential variables, and a parent pointer in the code-level definition which may not be desirable. Lastly [9] queries an LLM to generate memory safety specifications which are then tested through Bennet and Fulminate. The scope of this work is limited to memory safety and also do not support the wand connective as CN is not shipped with it.

8 Conclusion

We have presented SMEAGOL, a novel generation and testing framework involving magic wands across different data structures. The evaluation demonstrates SMEAGOL's efficacy in both inference and testing, and paves the way towards writing simpler invariants that rely on a minimal set of predicates. At present SMEAGOL does not support wand in the precondition as the generator is equipped to handle the $*$ operator only. The scope of this work is limited to testing only as CN does not have the support for wand in its verification pipeline. In future we'd like to extend this to support a verification pipeline to both automatic and interactive verifiers that translates the program and annotated specifications and invariants to the verifier language, so that inferred invariants can also be verified.

9 Data-Availability Statement

The code, benchmarks and, ground truth invariants are attached in the supplementary materials of the submission. The ground truth invariants can be tested by running the Docker image provided. To test the inference script, a personal API key needs to be configured for Anthropic, OpenAI, DeepSeek, and Kimi.

References

- [1] Zain K Aamer, Rini Banerjee, Hiroyuki Katsura, David Kaloper-Meršinjak, Dimitrios J. Economou, Kayvan Memarian, Dhruv Makwana, Neel Krishnaswami, Benjamin C. Pierce, Christopher Pulte, and Peter Sewell. 2026. Code-Specify-Test-Debug-Prove: Flexibly Integrating Separation Logic Specification into Conventional Workflows. *Proc. ACM Program. Lang.* 10, PLDI, Article 200 (June 2026), 27 pages. doi:10.1145/3808278
- [2] Zain K Aamer and Benjamin C Pierce. 2025. Bennet: Randomized Specification Testing for Heap-Manipulating Programs. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025), 3924–3953.
- [3] Vytautas Astrauskas, Aurel Bily, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J Summers. 2022. The prusti project: Formal verification for rust. In *NASA Formal Methods Symposium*. Springer, 88–108.
- [4] Rini Banerjee, Kayvan Memarian, Dhruv Makwana, Christopher Pulte, Neel Krishnaswami, and Peter Sewell. 2025. Fulminate: Testing CN Separation-Logic Specifications in C. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 1260–1292.
- [5] Stefan Blom and Marieke Huisman. 2014. The VerCors tool for verification of concurrent programs. In *International Symposium on Formal Methods*. Springer, 127–131.
- [6] Stefan Blom and Marieke Huisman. 2015. Witnessing the elimination of magic wands. *International Journal on Software Tools for Technology Transfer* 17, 6 (2015), 757–781.
- [7] John Boyland. 2003. Checking interference with fractional permissions. In *International Static Analysis Symposium*. Springer, 55–72.
- [8] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 66–87.
- [9] Twain Byrnes and Mike Dodds. 2026. CNnotator: LLM-Guided Memory Safety Annotation Synthesis. In *Proceedings of the 1st Workshop on Code Translation, Transformation, and Modernization (ReCode '26)*. Association for Computing Machinery, New York, NY, USA, 24–29. doi:10.1145/3786180.3788313
- [10] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 289–300.
- [11] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. doi:10.1007/S10817-018-9457-5
- [12] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J Summers. 2022. Sound automation of magic wands. In *International Conference on Computer Aided Verification*. Springer, 130–151.
- [13] Jenna DiVincenzo, Ian McCormack, Conrad Zimmerman, Hemant Gouni, Jacob Gorenburg, Jan-Paul Ramos-Dávila, Mona Zhang, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2025. Gradual C0: Symbolic Execution for Gradual Verification. *ACM Trans. Program. Lang. Syst.* 46, 4, Article 14 (Jan. 2025), 57 pages. doi:10.1145/3704808
- [14] Jérôme Dohrau, Alexander J Summers, Caterina Urban, Severin Münster, and Peter Müller. 2018. Permission inference for array programs. In *International Conference on Computer Aided Verification*. Springer, 55–74.
- [15] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
- [16] Lennard Gäher, Vincent Lafeychine, Sascha Kehrli, Avraham Shinnar, Wojciech Ozga, Guernsey Hunt, and Derek Dreyer. 2026. Bringing Foundational Verification to Real-World Rust Code. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 352 (Oct. 2026), 27 pages. doi:10.1145/3839484
- [17] Bolei Guo, Neil Vachharajani, and David I August. 2007. Shape analysis with inductive recursion synthesis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 256–265.
- [18] Guanhua He, Shengchao Qin, Wei-Ngan Chin, and Florin Craciun. 2013. Automated specification discovery via user-defined predicates. In *International Conference on Formal Engineering Methods*. Springer, 397–414.
- [19] Saffron Huang, Bryan Seethor, Esin Durmus, Kunal Handa, Miles McCain, Michael Stern, and Deep Ganguli. 2025. How AI is transforming work at Anthropic. *Anthropic. December* 2 (2025).
- [20] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.
- [21] Nicolas Klose and Peter Müller. 2026. Abductive Inference of Separation Logic Specifications with Isorecursive User-Defined Predicates and Magic Wands. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 356 (Oct. 2026), 28 pages. doi:10.1145/3839488
- [22] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and*

- Implementation. 788–801.
- [23] Chang Liu, Xiwei Wu, Yuan Feng, Qinxiang Cao, and Junchi Yan. 2024. Towards general loop invariant generation: a benchmark of programs with memory manipulation. *Advances in Neural Information Processing Systems* 37 (2024), 129120–129145.
- [24] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2010. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 211–222.
- [25] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2015. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*. Springer, 41–62.
- [26] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Automatic verification of iterated separating conjunctions using symbolic execution. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17–23, 2016, Proceedings, Part I* 28. Springer, 405–425.
- [27] Christopher Pulte, Dhruv C Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 2023. CN: Verifying systems C code with separation-logic refinement types. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1–32.
- [28] Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, and Xin Chen. 2013. Loop invariant synthesis in a combined abstract domain. *Journal of Symbolic Computation* 50 (2013), 386–408.
- [29] Shengchao Qin, Chenguang Luo, Wei-Ngan Chin, and Guanhua He. 2011. Automatically refining partial specifications for program verification. In *International Symposium on Formal Methods*. Springer, 369–385.
- [30] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. *ACM SIGPLAN Notices* 42, 6 (2007), 123–134.
- [31] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74.
- [32] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 158–174.
- [33] Malte Schwerhoff and Alexander J Summers. 2015. Lightweight Support for Magic Wands in an Automatic Verifier (Artifact). *Dagstuhl Artifacts Series* 1, 1 (2015), 10–1.
- [34] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2inv: A deep learning framework for program verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* 32. Springer, 151–164.
- [35] Jan Smans, Bart Jacobs, and Frank Piessens. 2012. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 1 (2012), 1–58.
- [36] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph. D. Dissertation. University of California, Berkeley.
- [37] Simon Spies, Lennard Gähler, Michael Sammler, and Derek Dreyer. 2024. Quiver: Guided Abductive Inference of Separation Logic Specifications in Coq. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 889–913.
- [38] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark Barrett, David Dill, Xiaokang Qiu, and Shuvendu K Lahiri. 2025. ClassInvGen: Class invariant synthesis using large language models. In *International Symposium on AI Verification*. Springer, 64–96.
- [39] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, Jianfeng Cai, Xinyuan Cai, Peizhou Cao, Yuxuan Cao, Ziwei Chai, Y Charles, et al. 2026. Kimi k3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653* (2026).
- [40] Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. 2013. Bi-abduction with pure properties for specification inference. In *Asian Symposium on Programming Languages and Systems*. Springer, 107–123.
- [41] Anjiang Wei, Tarun Suresh, Tianran Sun, Haoze Wu, Ke Wang, and Alex Aiken. 2025. InvBench: Can LLMs Accelerate Program Verification with Invariant Synthesis? *arXiv preprint arXiv:2509.21629* (2025).
- [42] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *Computer Aided Verification, Arie Gurfinkel and Vijay Ganesh (Eds.)*. Springer Nature Switzerland, Cham, 302–328.
- [43] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.
- [44] Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348* (2026).

[45] Ziyi Yang and Ilya Sergey. 2025. Inductive Synthesis of Inductive Heap Predicates. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 169–195.

[46] Zhe Zhou, Robert Dickerson, Benjamin Delaware, and Suresh Jagannathan. 2021. Data-driven abductive inference of library specifications. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29.

[47] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound gradual verification with symbolic execution. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2547–2576.