# Oracle-Guided Heap Invariant Synthesis


## Abstract

We consider the problem of verifying safety properties of heap-manipulating programs. A central challenge in such verification is to infer auxiliary invariants on the heap that enable one to prove the goal property. Inferring such invariants is tricky for current software verifiers due to the limitations of deductive proof engines for existing heap logics to automatically extract them during proof search. In this paper, we propose an alternative oracle-guided approach for heap invariant synthesis. Our approach reduces the verification problem to one of inductive synthesis, where the safety verification problem is reduced to one of synthesizing a heap separator pattern from positive and negative examples of heaps. This reduction makes our approach modular by offloading the invariant synthesis step to an external oracle. This oracle can be implemented in various ways, with varying degrees of expressiveness and automation. We present a theoretical analysis of this oracle-guided approach. We also show that the oracle-guided approach is effective in practice via an evaluation on benchmarks from recent software verification competitions, where we can outperform the state of the art verifiers.


## 1. Introduction

## 2. Overview

In this section, we give an informal overview of the approach that we propose in this paper. In §2.1, we introduce an example program `alt_list`, based on a benchmark in the SV-COMP [1] benchmark suite, that updates its heap using low-level memory operations. In §2.2, we review a class of heap patterns that represent sets of heaps of unbounded size. In §2.3, we present a class of proof structures that use the heap patterns introduced in §2.2 to represent program invariants in order to prove that low-level heap programs, such as `alt_list`, satisfy their desired assertions. In §2.4, we define a class of learning games of program heaps and heap patterns. In §2.5, we describe how a program verifier can reduce the problem of constructing a valid proof of program safety to winning a set of the learning games described in §2.4.

### 2.1 An example low-level heap program

Figure 1 contains the source code for a program `alt_list` written in a C-like, low-level language. For the states of `alt_list`, let a list be *alternating* if (1) the data field of the head of the list is equal to the value stored in Boolean variable `d` and (2) the data fields in successive cells of the list store alternating Boolean values. `alt_list` (1) iteratively constructs an alternating list of non-deterministic length and then (2) checks that the constructed list is indeed alternating.

```
1  void alt_list() {
2    bool d = TRUE;
3    List l = cons(d, NIL);
4    // LOOP-CONS: build a list with alternating Boolean values.
5    while (non_det()) {
6      d = !d;
7      l = cons(d, l);
8    }
9    // LOOP-CHK: check that the Boolean values alternate.
10   while (l != NIL) {
11     assert(l->data == d);
12     d = !d;
13     l = l->next;
14   }
15   return;
16 }
```

Figure 1: `alt_list`: a simplified version of an SV-COMP benchmark program that (1) constructs a list with cells that store alternating Boolean values and (2) checks that the Boolean values in successive cells alternate.

`alt_list` initializes the list stored in `l` to consist of a single cell whose value is equal to `d` and whose successor cell is `NIL` (the function `cons` takes as input a Boolean value $d$ and a list cell $l$ and returns a new list cell whose data field stores $d$ and whose `next` field stores $l$). `alt_list` then non-deterministically chooses whether to execute `LOOP-CONS`, prepends a new cell to the list stored in `l` (line 5). If `alt_list` chooses to execute `LOOP-CONS`, then it negates the value stored in Boolean variable `data` (line 6), and constructs a new list cell whose next cell is the cell stored in `l` and whose data value is stored in `data` (line 7).

After exiting `LOOP-CONS`, `alt_list` executes `LOOP-CHK` to iteratively check that the values in successive cells of `l` store alternating Boolean values. In each iteration, `alt_list` checks if `l` stores `NIL` (line 10). If not, then `alt_list` checks that the data value of `l` is equal to `data` (line 11), and if so, inverts the value stored in `data` (line 12) and updates `l` to store its successor (line 13). Otherwise, if `l` stores `NIL`, then `alt_list` returns successfully (line 15).

The problem that we address in this paper is to determine if a given low-level heap program, such as `alt_list`, always satisfies each of its assertions, such as the assertion on line 11, which checks that the list stored in `l` is alternating.

### 2.2 Representing sets of heaps with graph patterns

In this work, we propose a verifier, PROVEIT, that represents sets of heaps as graph *patterns*, which are directly analogous to three-valued structures introduced in previous work on shape analysis [18]. In this section, we review graph patterns as they have been presented in previous work, in particular how they are used to represent sets of program heaps.




, .





http://dx.doi.org/10.1145/

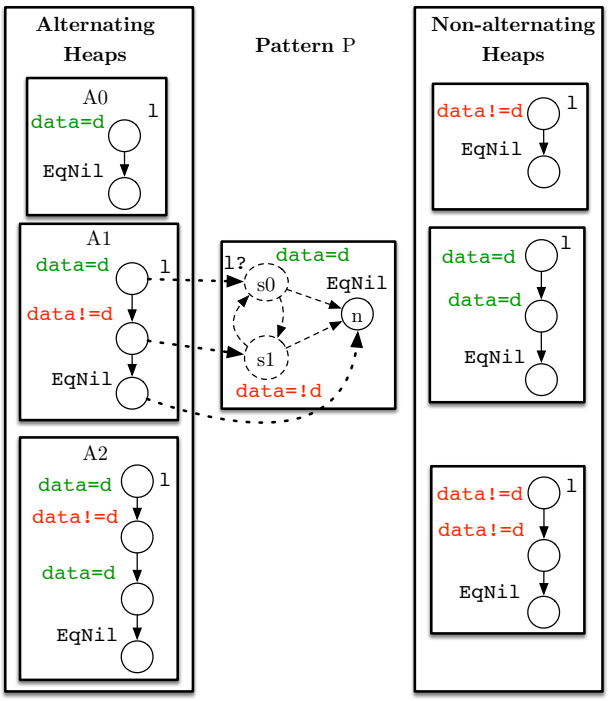


Figure 2: Sets of alternating heaps and non-alternating heaps of `alt_list` (§2.1), depicted as graphs, and a heap pattern that matches each of the alternating heaps and none of the non-alternating heaps. The labels of each node $n$ are written adjacent to the node in a heap graph are written adjacent to $n$; some label values are colored for clarity. Each edge is implicitly labeled with the field name `next`. Each summary node of the pattern $P$ is dashed, each indefinite node label is followed by a question mark, and each indefinite edge with an indefinite label is dashed. The dash-dotted lines from heap $A_1$ to $P$ depict a matching from $A_1$ to $P$.

In our approach, each program heap is modeled as a labeled graph, in which each node models a heap cell, and is labeled with facts about its corresponding heap cell, such as variables in which the cell is stored. Each edge in the heap graph is labeled with a field name; such labeled edges model which fields of cells point to other cells.

**Example 1.** *Figure 2 depicts distinct sets of graphs of heaps in* `alt_list` *states that (1) are alternating and (2) are not alternating (for now, ignore the pattern $P$ with dashed nodes and edges in the center of Figure 2). The alternating heaps depicted consist of the alternating heaps with one to three non-nil cells. The non-alternating heaps depicted consist of three non-alternating heaps with one to two non-nil cells.*

Each pattern is a graph in which nodes and edges are labeled from the space of annotations as the labels on the nodes and edges of heap graphs. However, a pattern graph may also contain *summary nodes* and *indefinite labelings*. A heap graph $H$ is *matched* by a pattern graph $P$ if there is a mapping $h$ from the nodes of $H$ to the nodes of $P$ such that:

1. If multiple nodes of $H$ are matched by a node $n$ of $P$, then $n$ is a summary node.
2. For each node $n$ of $H$, if $n$ is labeled with label $L$, then $h(n)$ is either definitely or indefinitely labeled with $L$. If $n$ is not labeled with label $L$, then $h(n)$ is either definitely not labeled or indefinitely labeled with $L$.
3. For all nodes $m$ and $n$ of $H$, if there is an edge from $m$ to $n$ labeled with field `f`, then there is an edge from $h(m)$ to $h(n)$ either definitely or indefinitely labeled with `f`. If there is no edge from $m$ to $n$ labeled with `f`, then either there is no edge from $h(m)$ to $h(n)$ labeled with `f` or there is an edge from $h(m)$ to $h(n)$ indefinitely labeled with `f`.

**Example 2.** *The heap pattern $P$ depicted in Figure 2 matches exactly the alternating heaps of* `alt_list`*. A matching from the second alternating heap $A_1$ in Figure 2 to $P$ is depicted with dotted edges from the nodes of the heap to the nodes of the pattern. Because $P$ matches each of the alternating heaps and none of the non-alternating heaps in Figure 2, we say that $P$* distinguishes *the alternating and non-alternating heaps.*

### 2.3 A proof structure for low-level heap programs

`alt_list` demonstrates that proving that some programs satisfy each of their assertions, which are defined purely over local variables, may still sometimes require a verifier to find invariants over the entire structure of the program heap. In particular, in order to prove that `alt_list` always satisfies its assertion at a line 12, a verifier must prove that at line 5, the heap always holds an alternating list.

PROVEIT attempts to construct proofs that are represented as a partial prefix-tree (i.e., an *unwinding tree*) of the program's control paths [15]. Each node in the tree models an occurrence of a control location in a program path, and each edge in the tree models a step of execution of the program over a sequence of non-branching instructions. Each node $n$ is thus identified by a sequence of instructions $\texttt{Instrs}_n$ that the program must execute to reach the node, and is annotated with an *invariant*, represented as a heap pattern (§2.2) that is satisfied by all states reached after the program executes $\texttt{Instrs}_n$.

An unwinding tree $T$ is a proof that a given error location $\texttt{L}_E$ is unreachable in any run of a program $P$ if:

1. The initial heap of $P$ satisfies the invariant at the root of $T$.
2. For each edge $(m, n)$ in $T$, the invariant at $m$ and the semantics of the instructions modeled by the edge $(m, n)$ imply the invariant at $n$.
3. Each node in $T$ that models $\texttt{L}_E$ is annotated with an invariant that is not satisfied by any program state.
4. Each leaf $l$ that models control location `L` of $T$ either models a terminal control location of $P$, or is *covered* by another node of $T$ that models `L`, and is annotated with a weaker invariant than the invariant of $l$.

If a leaf $l$ is covered by node $n$, the intuitively the proof tree need not be further expanded from $l$, because any proof tree rooted at $n$, which is a proof of safety for all paths with prefix $\texttt{Instrs}_n$, is a valid proof of safety for all paths with prefix $\texttt{Instrs}_l$.

**Example 3.** *Figure 3 depicts a prefix tree $T$ of an unwinding tree that proves that* `alt_list` *always satisfies the assertion at line 12. $T$ models runs of* `alt_list` *that execute* `LOOP-CONS` *most three times. Nodes* 0, 2, 4, *and* 6 *model states of* `alt_list` *when control is at the loop head, and nodes* 1, 3, *and* 5 *model states of* `alt_list` *when control exits* `LOOP-CONS`*. Each edge of $T$ is annotated with the sequence of instructions in* `alt_list` *that it models. Each node $n$ of $T$ is annotated with a heap pattern (§2.2) that over-approximates the set of heaps reachable by executing $\texttt{Instrs}_n$. Node* 6 *is covered by node* 4 *because the pattern at node* 6 *is entailed by the pattern at node* 4*. Thus, tree $T$ can be expanded into a complete tree that proves the safety of $P$ by expanding $T$ from only leaf nodes* 1, 3, *and* 5*, not from leaf node* 6*.*

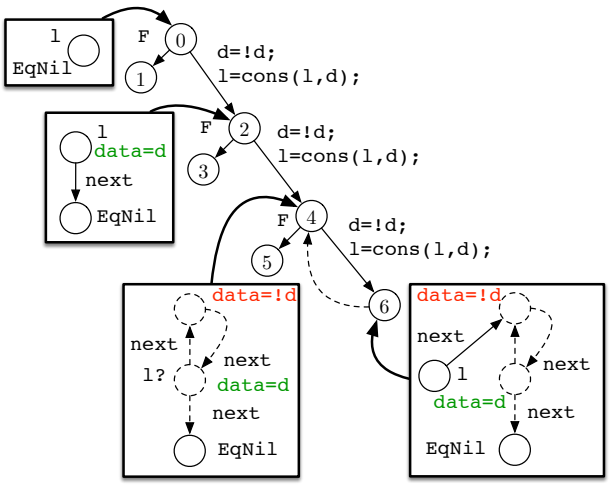


Figure 3: The prefix of an unwinding tree $T$ that proves that no run of `alt_list` violates its assertion. Edge tree edge is annotated with the instruction sequence that it simulates, and each node is annotated with its pattern invariant.

### 2.4 Learning heap patterns as a game

BH: this section needs to be updated to match the new formulation

The key observation behind the design of PROVEIT is that any pair of disjoint sets of heaps $H^+$ and $H^-$ defines a natural game, in which the objective for one player is to learn a pattern that distinguishes $H^+$ from $H^-$. In particular, let an *inductive pattern synthesizer* be an oracle that takes as input a finite set of positive heaps $H^+$ and negative heaps $H^-$, and returns a heap pattern that is matched by each heap $H^+$ and no heap in $H^-$. The synthesizer's goal in the game is to synthesize a heap pattern that distinguishes $H^+$ and $H^-$, without direct access to $H^+$ and $H^-$. The state of the game consists of finite subsets of *revealed* $H_0^+ \subseteq H^+$ and revealed $H^- \subseteq H^-$, and a pattern $P$ generated by the synthesizer that distinguishes $H_0^+$ from $H_0^-$. In each play, if the inductive pattern synthesizer has not won, then the pattern verifier reveals either a heap in $H^+$ not matched $P$ or a heap in $H^-$ that is matched by $P$. The inductive pattern synthesizer then generates a new pattern that must match all revealed patterns in $H^+$ and no revealed patterns in $H^-$.

**Example 4.** *For* `alt_list`*, the reachable heaps and heaps that lead to an assertion violation from line* $G_5$ *(i.e., the alternating heaps and non-alternating heaps) define a pattern-synthesis game* $G_5$*. One valid play of the game with an inductive pattern synthesizer* $O_5$ *from a game state consisting of no revealed positive heaps and the non-alternating heaps in Figure 2 as revealed negative heaps is as follows: (1)* $O_5$ *plays the pattern that annotates node* 0 *in Figure 3; (2) the pattern verifier reveals heap* $A_1$ *in Figure 2; (3)* $O_5$ *plays the pattern that annotates node* 2 *in Figure 3; (4) the pattern verifier reveals heap* $A_2$ *in Figure 2; (5)* $O_5$ *plays the pattern that annotates node* 4 *in Figure 3 (i.e., pattern* $P$ *in Figure 2), and wins the game.*

### 2.5 From game strategies to program proofs

The main result of our work is that while the problem of verifying heap-manipulating programs is undecidable, it can be reduced to winning a finite set of pattern-synthesis games. The primary difficulty in constructing an unwinding tree $T$ that proves the safety of a program is in inferring patterns for the nodes of $T$ that are (1) sufficiently strong enough to prove that each path of $T$ to an error node is infeasible but (2) sufficiently weak that they can cover the patterns of sufficiently many leaf nodes to bound the set of program paths that must be modeled. For the example of `alt_list`, the pattern on node 4 of the unwinding tree in Figure 3 satisfies both of these criteria; however, in general the problem of inferring sufficient patterns is undecidable, and to date, there are not general-purpose analyses that can infer such invariants for practical programs.

In settings where it is not critical to infer properties of the program's entire heap, it is sufficient to infer invariants for an unwinding tree by obtaining an *interpolant* [15] for each node $n$ of two formulas describing (1) states reachable by executing the instruction sequence of $n$ from the beginning of the program and (2) states from which $n$ reaches an error. However, interpolation-based approaches must infer interpolants as formulas in a theory for which the problem of constructing an interpolant is decidable, typically the combined theory of linear arithmetic, uninterpreted functions, and arrays (LIUFA). Unfortunately, formulas in such theories cannot naturally describe sets of heaps with arbitrarily many cells, such as the set of alternating heaps of `alt_list` (§2.1).

In this work, we explore under what conditions a verifier can efficiently learn heap patterns that are sufficient invariants for proving the correctness of a program, under the assumption that the verifier can query an oracle that can efficiently win a class of the heap-learning games described in §2.4.

**Example 5.** *Suppose that* PROVEIT *has constructed the unwinding tree depicted in Figure 3, and must choose a pattern with which to annotate node* 4*, which models line 5 of* `alt_list`*.* PROVEIT *could choose a pattern* $P$ *that only distinguishes between alternating and non-alternating lists of length less than or equal to two. However,* $P$ *is, intuitively too strong an invariant in that it cannot cover any valid annotation of node* 6*.*

*Alternatively, if* PROVEIT *plays the game* $G_5$ *as pattern verifier against the pattern synthesizer* $O_5$ *(Ex. 4), and annotates node* 4 *with the winning pattern played by* $O_5$ *(as depicted in Figure 3), then* PROVEIT *can construct an unwinding tree that proves the safety of* `alt_list`*.*

The main result that we present in this paper is that for a given program $P$, if our program verifier PROVEIT has access to an inductive pattern synthesizer that wins a game defined analogously to the game $G_5$ defined for line 5 of Figure 1 (Ex. 4) for each control location of $P$, then PROVEIT verifies $P$ successfully.

## 3. Background

In this section, we define a standard language that performs low-level memory operations to update linked data structures (§3.1). We then review definitions of three-valued structures introduced in previous work [18], which we use to formulate patterns over program heaps (§3.2).

### 3.1 Language Definition

In this section, we define the syntax (§3.1.1) and semantics (§3.1.2) of our subject language LANG.

#### 3.1.1 Syntax

A LANG program is a sequence of instructions that operate on a fixed set of predicate variables and pointers to heap objects. The syntax of LANG is given in Figure 4 for fixed finite sets of control locations `Locs`, predicate variables $\texttt{Vars}_P$, heap variables $\texttt{Vars}_H$, and heap fields `Fields`. A program is a sequence of instruction, each labeled with a control location (Equation 1). An instruction either updates the program's predicate variables or heap variables (Equation 2). An instruction that updates predicate variable either stores in a predicate variable the result of a Boolean operation

$$\text{LANG} := (\texttt{Locs} : \texttt{Instrs})^* \quad (1)$$
$$\texttt{Instrs} := \text{instr}_P \mid \text{instr}_H \quad (2)$$
$$\text{instr}_P := \texttt{Vars}_P := \texttt{Vars}_P\ \texttt{Ops}_P\ \texttt{Vars}_P \quad (3)$$
$$\mid \texttt{Vars}_P := (\texttt{Vars}_H = \texttt{Vars}_H) \quad (4)$$
$$\mid \texttt{br}\ \texttt{Vars}_P, \texttt{Locs}, \texttt{Locs} \quad (5)$$
$$\text{instr}_H := \texttt{Vars}_H := \texttt{alloc}() \quad (6)$$
$$\mid \texttt{Vars}_H := \texttt{Vars}_H \quad (7)$$
$$\mid \texttt{Vars}_H := \texttt{Vars}_H \rightarrow \texttt{Fields} \quad (8)$$
$$\mid \texttt{Vars}_H \rightarrow \texttt{Fields} := \texttt{Vars}_H \quad (9)$$

Figure 4: Syntax of heap-updating programs, LANG. The spaces of control locations, predicate variables, heap variables, and fields are denoted $\texttt{Locs}$, $\texttt{Vars}_P$, $\texttt{Vars}_H$, and $\texttt{Fields}$, respectively.

(Equation 3), an equality test on heap cells (Equation 4), or branches control based on the value in a predicate variable (Equation 5). An instruction that updates heap variables either allocates a new heap cell (Equation 6), copies the heap cell from one pointer variable to another (Equation 7), loads a heap cell into a pointer variable (Equation 8), or stores a heap cell as a child of a cell in a pointer variable (Equation 9).

For each program $P \in$ LANG, the control-flow graph of $P$, $\text{cfg}_P \subseteq \texttt{Locs} \times \texttt{Instrs} \times \texttt{Locs}$, is defined in the standard way.

#### 3.1.2 Semantics

A LANG program updates a state, which consists of a control location, evaluation of predicate variables, and a heap, which is a graph with labeled edges.

**Definition 1.** *A* LANG heap *is a triple* $(C, V_H, F)$*, where*

1. *For the countable universe of heap cells, denoted* $\mathcal{C}$*,* $C \subseteq \mathcal{C}$ *is a finite set of cells that contains a distinguished cell* $\text{nil} \in \mathcal{C}$*.*
2. $V_H : \texttt{Vars}_H \rightarrow C$ *maps each heap variable to the cell that it stores. We denote the space of evaluations of heap variables as* $\text{Evals}_H = \texttt{Vars}_H \rightarrow \mathcal{C}$*.*
3. $F : C \times \texttt{Fields} \rightarrow C$ *maps each cell* $c \in C$ *and field* $\texttt{f} \in \texttt{Fields}$ *to the child of* $c$ *at* $\texttt{f}$*. We denote the space of field maps as* $\text{FieldMaps} = \mathcal{C} \times \texttt{Fields} \rightarrow \mathcal{C}$*.*
4. $\text{PredLbl} : C \rightarrow 2^{\texttt{Vars}_P}$ *maps each cell* $c \in C$ *to an assignment of Boolean values to* $\texttt{Vars}_P$*. We denote the space of predicate labeling functions as* $\text{PredLblFn} = \mathcal{C} \rightarrow 2^{|\texttt{Vars}_P|}$*.*

*We denote the space of heaps as* $\text{Heaps} = \mathcal{P}(\mathcal{C}) \times \text{Evals}_H \times \text{FieldMaps} \times \text{PredLblFn}$*, and the space of characteristic functions of languages of heaps as* $\text{Chars} = \text{Heaps} \rightarrow \mathbb{B}$*.*

*A* LANG state *is a triple* $(\texttt{L}, V_P, h)$*, where*

1. $\texttt{L} \in \texttt{Locs}$ *is a control location.*
2. $V_P : \texttt{Vars}_P \rightarrow \mathbb{B}$ *is an evaluation of predicate variables. We denote the space of evaluations of predicate variables as* $\text{Evals}_P = \texttt{Vars}_P \rightarrow \mathbb{B}$*.*
3. $h \in \text{Heaps}$ *is a heap.*

*We denote the space of states as* $Q = \texttt{Locs} \times \text{Evals}_P \times \text{Heaps}$*.*

**Example 6.** *The C-like program* `alt_list` *(§2.1) corresponds directly to a* LANG *program. An example of a reachable state of* `alt_list` *line 5 is, for the alternating heap* $A_0$ *given in §2.2, Figure 2, the state* $(\texttt{L5}, [\texttt{d} \mapsto \text{True}], A_0)$*.*

In each run, each program $P \in$ LANG updates its state by traversing control locations, executing the instruction $\texttt{i}$ at each control location to update its state according to the *program transition relation* $\rightarrow_P \subseteq Q \times Q$, where $\rightarrow_P$ is defined by a transition relation $\rightarrow_H \subseteq \text{Heaps} \times \text{instr}_H \times \text{Heaps}$ over heaps and heap-update instructions (**??**). The definition of $\rightarrow_P$ from $\rightarrow_H$ is standard, and thus we omit a complete description.

A run of a program $P$ is a sequence of states in which each pair of successive states are in $P$'s transition relation.

**Definition 2.** *For each program* $P \in$ LANG*, a* run *of* $P$ *is a sequence of states* $q_0, q_1, \ldots, q_n$ *such that for each* $0 \leq i < n$*,* $q_i \rightarrow_P q_{i+1}$*.*

### 3.2 Heap-pattern Language

A heap pattern is a labeled graph whose nodes and edges model the cells and fields of potentially-many heaps. The nodes and edges of a pattern are annotated with three-valued truth values that represent if all cells modeled by a node either definitely are, definitely are not, or may be (1) stored in a given heap variable or (2) connected by a heap field. The heap patterns are based on three-valued structures introduced in previous work on shape analysis [18].

**Definition 3.** *Let the domain of* three-valued truth values *be* $\mathbb{B}_3 = \{\text{True}, \text{False}, \text{Maybe}\}$*. A* heap pattern *is a labeled graph* $(N, \texttt{V}, E)$*, where:*

1. *For the countable universe of nodes* $\mathcal{N}$*,* $N \subseteq \mathcal{N}$ *is a finite set of nodes.*
2. $\texttt{V} : N \times \texttt{Vars}_H \rightarrow \mathbb{B}_3$ *is a* heap-variable labeling*. We denote the space of heap-variable labelings as* $\text{VarLbls} = \mathcal{N} \times \texttt{Vars}_H \rightarrow \mathbb{B}_3$*.*
3. $\texttt{P} : N \times \texttt{Vars}_P \rightarrow \mathbb{B}_3$ *is a* predicate-variable labeling*. We denote the space of predicate-variable labelings as* $\text{PredLbls} = \mathcal{N} \times \texttt{Vars}_P \rightarrow \mathbb{B}_3$*.*
4. $E : N \times \texttt{Fields} \times N \rightarrow \mathbb{B}_3$ *is a set of labeled* edges*. We denote the space of labeled edges as* $\text{LblEdges} = \mathcal{N} \times \texttt{Fields} \times \mathcal{N}$*.*

*We denote the class of all heap patterns as* $\text{Pats} = \mathcal{P}(\mathcal{N}) \times \text{VarLbls} \times \text{PredLbls} \times \text{LblEdges}$*.*

Program in a language with low-level memory updates, such as `alt_list` (§2.1), can be modeled as LANG programs if a verifier is provided with a bounded set of "relevant" predicates $R$ over heap cells. In such a case, the verifier can simply model the predicates in $R$ as heap fields. To simplify the presentation of our analysis, we assume that a separate program analysis has inferred such a set of relevant predicates, and that such predicates are already modeled as LANG fields.

**Example 7.** *Pattern* $P$ *(§2.2, Figure 2) is a heap pattern that contains two summary nodes* $s_0$ *and* $s_1$*, and one non-summary node* $n$*. For summary node* $s_0$*, predicate* `data!=d` *maps to* True *and all other predicates map to* False *For summary node* $s_1$*, predicate* `data=d` *maps to* True*, predicate* `1` *maps to* Maybe*, and all other predicates map to* False*. For concrete node* $n$*, predicate* `EqNil` *maps to* True *and all other predicates map to* False*.*

*The edges* $(s_0, s_1)$*,* $(s_0, n)$*,* $(s_1, s_0)$*, and* $(s_1, n)$ *map on field* `next` *to* Maybe*. All other edges on all other fields map to* False*.*

Each heap $h$ defines a heap pattern that represents exactly $h$.

**Definition 4.** *For each heap* $h = (C, V_H, F)$*, the* concrete pattern *of* $h$ *is* $G_h = (N_h, \texttt{V}_h, E_h)$*, where:*

1. *Each node in* $G_h$ *is a cell in* $C$*. I.e.,* $N_h = C$*.*
2. *Each heap-variable binding in* $V_H$ *defines a node labeling in* $\texttt{V}_h$*. I.e., for each node* $c \in C$ *and heap variable* $\texttt{p} \in \texttt{Vars}_H$*, if* $V_H(\texttt{p}) = c$*, then* $\texttt{V}_h(c, \texttt{p}) = \text{True}$*, and otherwise,* $\texttt{V}_h(c, \texttt{p}) = \text{False}$*.*
3. *Each field binding in* $F$ *defines an edge in* $E_h$*. I.e., for all cells* $c, c' \in C$ *and each field* $\texttt{f} \in \texttt{Fields}$*, if* $c' = F(c, \texttt{f})$*, then* $E_h(c, \texttt{f}, c') = \text{True}$*, and otherwise,* $E_h(c, \texttt{f}, c') = \text{False}$*.*

**Example 8.** *§2.2, Figure 2 contains concrete patterns for six heaps: three alternating heaps and three non-alternating heaps.*

The entailment relation over heap patterns formulates under both (1) under what conditions a heap pattern describes a heap and (2) under what conditions all of the heaps described by one heap pattern are described by another pattern.

**Definition 5.** *Let the* information-precision ordering $\sqsubseteq_3 \subseteq \mathbb{B}_3 \times \mathbb{B}_3$ *be such that* $\mathsf{True} \sqsubseteq_3 \mathsf{Maybe}$ *and* $\mathsf{False} \sqsubseteq_3 \mathsf{Maybe}$.

*For all heap patterns* $P, P' \in \mathsf{Pats}$ *with* $P = (N, \mathtt{V}, E)$ *and* $P' = (N', \mathtt{V}', E')$, $P$ entails $P'$, *denoted* $P \preceq P'$, *if there is a map* $h : N \to N'$ *such that:*

- $h$ *embeds heap-variable assignments. I.e., for each pattern node* $n \in N$, $\mathtt{V}(n) \sqsubseteq_3 \mathtt{V}'(h(n))$.
- $h$ *embeds field labelings. I.e., for all pattern nodes* $n_0, n_1 \in N$ *and fields* $\mathtt{f} \in \mathtt{Fields}$, $E(n_0, \mathtt{f}, n_1) \sqsubseteq_3 E'(h(n_0), \mathtt{f}, h(n_1))$.

For each heap $h \in \mathsf{Heaps}$ and pattern $P \in \mathsf{Pats}$, if the concrete pattern (Defn. 4) of $h$ entails $P$, then $h$ is *modeled* by $P$, which we alternatively denote as $h \preceq P$. For any two heap patterns $P_0, P_1 \in \mathsf{Pats}$ and heap $h \in \mathsf{Heaps}$, if $P_0 \models P_1$ and $h \preceq P_0$, then $h \preceq P_1$.

**Definition 6.** *In §2.2, Figure 2, each of the patterns* $A_0$, $A_1$, *and* $A_2$ *for an alternating heap entails the pattern* $P$. *A matching from* $A_1$ *to* $P$ *is depicted as dotted arrows from the nodes of* $A_1$ *to the nodes of* $P$.

# 4. Problem Statement

## 4.1 Problem Definition

The verification problem that we consider is to determine if any run of a program reaches a given control location.

**Definition 7.** *For each* LANG *program* $P \in$ LANG *and control location* $\mathtt{L} \in \mathtt{Locs}$, *a solution to* REACH$(P, \mathtt{L})$ *is a decision as to whether any run of* $P$ *contains a state whose control location is* $\mathtt{L}$.

# 5. Technical Approach

In this section, we give the design (§5.1) and key correctness properties (§5.7) of a REACH verifier, PROVEIT.

## 5.1 PROVEIT Design

In this section, we give the design of PROVEIT. For a given program $P$, PROVEIT attempts to derive inductive invariants that prove the correctness of $P$ that are represented as heap shapes, using operations given in §5.2.1. In particular, PROVEIT attempts to construct a proof represented as a partial unrolling of the control paths of $P$, annotated with heap shapes as invariants (§5.4). To construct an unwinding tree that represents a valid proof of the safety of $P$ (§5.5), PROVEIT uses an operation REFINE (§5.5.1), which refines candidate invariants of $P$.

## 5.2 Shape Domains

In this section, we define *shape domains* as an abstract space equipped with operations needed by our analysis (§5.2.1). We then describe abstractions for shape analysis presented in previous work can be viewed as shape domains (§5.2.2).

### 5.2.1 Definition of shape domains

A shape domain is a space in which each element describes a set of program heaps.

**Definition 8.** *A shape domain is a tuple* $(\mathcal{S}, \in, 0, \Omega, \mathsf{ChkSub}, \mathsf{AbsPost}, \sqsubseteq)$, *where:*

- $\mathcal{S}$ *is the space of heap* shapes.
- $\in \subseteq \mathsf{Heaps} \times \mathcal{S}$ *is the* membership *relation. For each shape* $S \in \mathcal{S}$, *we denote the set of shapes that are members of* $S$ *(i.e., the* language *of* $S$*) as* $\mathcal{L}(S)$.
- $0 \in \mathcal{S}$ *is the* empty *shape. For each heap* $h \in \mathsf{Heaps}$, $h \notin 0$.
- $\Omega \in \mathcal{S}$ *is the* universal *shape. For each heap* $h \in \mathsf{Heaps}$, $h \in \Omega$.
- *Let* $S_0, S_1 \in \mathcal{S}$ *be shapes. If for each heap* $h \in \mathsf{Heaps}$ *such that* $h \in S_0$ *it holds that* $h \in S_1$, *then* $S_0$ *is* subsumed *by* $S_1$, *denoted* $S_0 \preceq S_1$. *If* $S_0$ *subsumes* $S_1$ *and* $S_1$ *subsumes* $S_0$, *then we say that* $S_0$ *is* equivalent *to* $S_1$.
  $\mathsf{ChkSub} : \mathcal{S} \times \mathcal{S} \hookrightarrow \mathsf{Heaps} \cup \{\mathsf{IsValid}\}$ *is an effective procedure such that for all shapes* $S_0, S_1 \in \mathcal{S}$, *if* $S_0 \preceq S_1$, *then* $\mathsf{ChkSub}(S_0, S_1) = \mathsf{IsValid}$; *otherwise,* $\mathsf{ChkSub}(S_0, S_1) = h \in \mathsf{Heaps}$ *such that* $h \in S_0$ *and* $h \notin S_1$.
- $\mathsf{AbsPost} : \mathtt{Instrs} \to \mathcal{S} \to \mathcal{S}$ *maps each instruction to an abstract forward transformer over shapes. For each instruction* $\mathtt{i} \in \mathtt{Instrs}$, *heap* $h \in \mathsf{Heaps}$, *and shape* $S \in \mathcal{S}$, *if* $h \in S$, *then* $\mathsf{post}[\mathtt{i}](h) \in \mathsf{AbsPost}[\mathtt{i}](S)$.
- $\sqsubseteq \subseteq \mathcal{S} \times \mathcal{S}$ *is the* optimality *partial ordering over shapes. For all shapes* $S_0, S_1 \in \mathcal{S}$, *we denote the fact that* $S_0$ *is* more optimal *than* $S_1$ *as* $S_0 \sqsubseteq S_1$.

### 5.2.2 Instances

Previous work on shape analysis has presented several abstract domains that can be extended to form shape domains.

***Three-valued structures*** Three-valued Logic Analysis (TVLA) represents sets of program states as three-valued structures [18]. TVLA uses a domain of shapes that are three-valued structures, extended with a bottom element, which serves as an empty element. Both the membership function and subsumption procedures are defined using an embedding relation from two-valued structures, which represent memory heaps, and three-valued structures to tree-valued structures. Abstract transformers are represented as predicate transformers specified as first-order logical formulas.

While previous work on TVLA does not explicitly define optimality orderings over shapes, orderings that naturally describe the properties of interest for a shape analysis include the subsumption relation and minimality over equivalent patterns.

***Classes of graph automata*** Automata for various classes of graphs can represent sets of heaps. In particular, word automata can be naturally extended to represent heaps that are bounded tuples of lists, and tree automata can be naturally extended to represent heaps that are bounded tuples of trees. Membership and subsumption can be implemented for such domains with standard operations over automata, and abstract transforms can also be defined.

One natural optimality ordering over such shapes orders graph automata $A_0$ as more optimal than $A_1$ if $A_0$ contains fewer states than $A_1$.

### 5.2.3 Automatically-Learnable Shape Domains

A shape domain $D$ is *learnable* if there is an automatic procedure that for each language $L$ of graphs, can learn a shape in $D$ whose language is $L$ using access to only the characteristic function of $L$ and a function that validates elements in $D$ as having language $L$.

**Definition 9.** *A* learnable shape domain *is a pair* $(D, \mathsf{ShapeLearner})$, *where:*

- $D$ *is a shape domain.*
- *Let* $V : \mathcal{S} \to \{\mathsf{IsValid}\} \cup \mathbb{B} \times \mathsf{Heaps}$ *be a* validator *for* $L$. *I.e., for each shape* $S \in \mathcal{S}$, *if* $\mathcal{L}(S) = L$, *then* $V(S) = \mathsf{IsValid}$; *otherwise, if* $\mathcal{L}(S) \not\subseteq L$, *then* $V(S) = (\mathsf{True}, h)$, *where* $h \in \mathcal{L}(S)$ *and* $h \notin L$; *otherwise (if* $L \not\subseteq \mathcal{L}(S)$*), then* $V(S) = (\mathsf{False}, h)$, *where* $h \notin \mathcal{L}(S)$ *and* $h \in L$. *We denote*

*the space of all validators for $D$ as* $\mathsf{Validators}_D = \mathcal{S}_D \to \{\mathsf{IsValid}\} \cup \mathbb{B} \times \mathsf{Heaps}$.
*The* shape learner $\mathsf{ShapeLearner} : \mathsf{Chars} \times \mathsf{Validators}_D \to \mathcal{S}_D$ *is such that for each language $L \subseteq \mathsf{Heaps}$, if $\chi_L \in \mathsf{Chars}$ is a characteristic function for $L$ and $V_L \in \mathsf{Validators}_D$ is a validator for $L$, then* $\mathsf{ShapeLearner}(\chi_L, V_L)$ *is an optimal shape in $\mathcal{S}_D$ with language $L$.*

***Instances*** The shape domains of deterministic word and tree automata ordered by minimality of size can be extended to learnable shape domains, using algorithms presented for learning regular languages of words [3] and trees [6].

However, there are no known results that provide active learners for three-valued structures.

### 5.3 Learning Oracles

We will present two shape analyses, each of which attempts to prove the safety of a given program by querying an oracle for a learning problem over heaps and shapes. To define the queries answered by oracles in both classes, it will be useful to first define a relevant inference problem over shape domains.

**Definition 10.** *A* partial characterization *function is a function* $\chi : \mathsf{Heaps} \to \mathbb{B} \cup \{\mathsf{unk}\}$. *We denote the space of all partial characterization functions as* $\mathsf{PartChars} = \mathsf{Heaps} \to \mathbb{B} \cup \{\mathsf{unk}\}$.

*For shape domain $D$, a* valid refinement *of $\chi$ is a shape $S \in \mathcal{S}_D$ such that for each heap $h \in \mathsf{Heaps}$,*

- *If* $\chi(h) = \mathsf{True}$, *then* $h \in S$.
- *If* $\chi(h) = \mathsf{False}$, *then* $h \notin S$.

*A solution to the* optimal refinement inference problem $\textsc{OptRefine}(\chi, D)$ *is a shape $S \in \mathcal{S}_D$ such that:*

1. *$S$ is a valid refinement of $\chi$.*
2. *$S$ is at least as optimal as all valid refinements of $\chi$ in $D$.*

A synthesizing oracle for shape domain $D$ takes a partial characterization function $\chi$ over heaps and directly solves the optimal refinement inference problem defined by $\chi$.

**Definition 11.** *For each shape domain $D \in \mathsf{ShapeDomains}$, a* shape-synthesizing oracle *is a function $O : \mathsf{PartChars} \to D$ such that for each partial characterization function $\chi \in \mathsf{PartChars}$, $O(\chi)$ is a solution to* $\textsc{OptRefine}(\chi, D)$. *We denote the space of all synthesis oracles for $D$ as* $\mathsf{SynOracles}_D = \mathsf{PartChars} \to D$.

For shape domain $D$, a shape characterization oracles takes a partial characterization function $\chi$ and implements the characteristic function of a solution to the optimal refinement inference problem defined by $\chi$.

**Definition 12.** *For each shape domain $D \in \mathsf{ShapeDomains}$, a* characterization oracle *is a function $O : \mathsf{PartChars} \to \mathsf{Chars}$ such that for each partial characterization function $\chi \in \mathsf{PartChars}$, $O(\chi)$ is the characteristic function of a solution to* $\textsc{OptRefine}(\chi, D)$. *We denote the space of all characterization oracles for $D$ as* $\mathsf{CharOracles}_D = \mathsf{PartChars} \to \mathsf{Chars}$.

### 5.4 Unwinding Trees

ProveIt models a program as a tree of program paths annotated with heap shapes as invariants of heaps reached by all runs of the path.

**Definition 13.** *For a fixed shape domain $D$, a* shape tree *is a four-tuple $(N, E, \sigma, \gamma)$, where:*

1. *For the countable universe of tree nodes $\mathcal{N}$, $N \subseteq \mathcal{N}$ is the set of* nodes.
2. *$E \subseteq N \times N$ is the set of* edges, *where the graph $(N, E)$ is a tree. We denote the ancestor relation over nodes in $N$ as $\sqsubseteq \subseteq N \times N$.*
3. *$\sigma : N \to \mathcal{S}_D$ maps each node to an invariant represented as a heap shape (Defn. 8).*
4. *$\gamma \subseteq N \times N$ is the* cover *relation. For all nodes $m, n \in N$, if $(m, n) \in \gamma$, then $m$ is a leaf and $\sigma(m) \preceq \sigma(n)$. In such a case, we say that $m$ is* covered. *A tree is* complete *if all of its leaves are covered.*

*We denote the space of shape trees as* $T_S = \mathcal{P}(\mathcal{N}) \times (\mathcal{N} \times \mathcal{N}) \times (N \to \mathcal{S}_D) \times (\mathcal{N} \times \mathcal{N})$.

For unwinding tree $T$, we denote the components of $T$ as $N_T$, $E_T$, $\sigma_T$, and $\gamma_T$.

**Example 9.** *§2.3, Figure 3 depicts a shape tree $(N, E, \sigma, \gamma)$ that models some executions of* `alt_list` *(formulated precisely below). The nodes $N$ are depicted as circles, and the edges $E$ between nodes are depicted as solid edges. For each node $n \in N$, the shape $\sigma(n)$ is provided as an annotation. The only cover edge in $\gamma$ is depicted as a dashed edge from node 6 to node 5.*

Unwinding trees model the possible runs of programs.

**Definition 14.** *For program $P \in$ Lang, an* unwinding tree *is a pair $(T, L)$, where:*

- *$T = (N, E, \sigma, \gamma) \in T_S$ is a shape tree (Defn. 13).*
- *$L : N \to \mathtt{Locs}$ is a map from each node of $T$ to a control location such that for each uncovered node $n \in N$ and each control-flow edge $(L(n), \mathtt{i}, \mathtt{L}') \in E$, there is some tree edge $(n, n') \in E$ such that $\mathsf{post}[\mathtt{i}](\sigma(n)) \preceq \sigma(n')$. We denote the space of location maps as $\mathsf{LocMaps} = N \to \mathtt{Locs}$.*

*We denote the space of unwinding trees as $T_U = T_S \times \mathsf{LocMaps}$.*

Unwinding trees constructed by ProveIt to attempt to prove that a given error location is unreachable in a given program. In particular, for each program $P$ with unwinding tree $T = ((N, E, \sigma, \gamma), L)$ and error control location $\mathtt{L}_E \in \mathtt{Locs}$, let $T$ be *safe* if for each node $n \in N$ such that $L(n) = \mathtt{L}_E$ (i.e., each node $n$ that *models* $\mathtt{L}_E$), the $P(N)$ is not matched by any heap. If $P$ has a complete and safe unwinding tree, then $\mathtt{L}$ is not reachable in any run of $P$ (this fact is established and used in the proof of soundness of ProveIt, given in §5.7.1).

**Example 10.** *The unwinding tree $T$ depicted in §2.3, Figure 3 is an unwinding tree for $P$, with a location map from nodes 0, 2, 4, and 6 to line 5 of* `alt_list`*, and from nodes 1, 3, and 5 to line 9 of* `alt_list`*. $T$ is not complete, because only the leaf node 6 is covered.*

### 5.5 ProveIt Core

In this section, we describe the core ProveIt algorithm, given in algorithm 1. ProveIt takes as input a program $P$ and an error control location $\mathtt{L}_E$, and attempts to determine if $\mathtt{L}_E$ is unreachable in $P$. ProveIt runs a recursive procedure ProveIt′, which maintains an unwinding tree $T$ of $P$ and a *frontier* of uncovered tree nodes to analyze in order to extend $T$ to be a complete unwinding tree. If the set of frontier nodes is empty (line 1), then ProveIt determines that $T$ is a complete unwinding, and determines $\mathtt{L}_E$ is unreachable in $P$ (line 2).

If the set of frontier nodes is non-empty, then ProveIt chooses a node $n \in F$ (line 3). If $n$ does not model the error location $\mathtt{L}_E$ (line 4), then ProveIt extends $T$ with a node that models each control successor of the control location of $n$ to form a new unwinding tree $T'$, by invoking a procedure `Unwind` (line 5), adds each new node in $T_U$ to the remaining frontier nodes $F$ to construct a new frontier $F'$ (line 6), and invokes itself recursively on $T_U$ and $F'$ (line 7).

If $n$ models the error control location $\mathtt{L}_E$ (line 8), then ProveIt invokes the procedure Refine (described in §5.5.1) on $T$ and

**Input** : $P$: a program; $\mathtt{L}_E$: an error location
**Output** : A decision as to whether $\mathtt{L}_E$ is unreachable in $P$.

```
1  Procedure ProveIt ((T, L), ϵ)
2      return True
3  Procedure ProveIt′((T, L), n :: F)
4      if L(n) ≠ L_E then
5          T′ := Unwind(T, n) ;
6          F′ := F ∪ (N_T′ \ N_T) ;
7          return ProveIt′(T′, F′) ;
8      else
9          switch Refine(T, L_E) do
10             case unsafe  return unsafe ;
11             case T′
12                 T′ := UpdCover(T) ;
13                 F′ := UncoveredLeaves(T′) ;
14                 return ProveIt′(T′, F′) ;
15             end
16         endsw
17     end
18 n := FreshNode() ;
19 T0 := (({n}, ∅, [n ↦ P_Ω], ∅), [n ↦ init(P)]) ;
20 F0 := [n] ;
21 return ProveIt′(T0, F0) ;
```

**Algorithm 1:** ProveIt: the core verification algorithm. ProveIt takes as input (1) a Lang program $P$ and (2) an error control location $\mathtt{L}_E$, and determines if $\mathtt{L}_E$ is reachable in $P$.

$\mathtt{L}_E$. If Refine returns the value unsafe (line 10), then ProveIt determines that $\mathtt{L}_E$ is reachable in $P$ (line 10). Otherwise, if Refine returns an unwinding tree $T'$ whose invariants refute all paths to $n$ (line 11), then ProveIt updates the cover relation of $T$ to form an unwinding tree $T'$ by invoking a procedure UpdCover (line 12; UpdCover is described below), collects the set of uncovered leaves $F'$ in $T'$ by invoking a procedure `UncoveredLeaves` (line 13), and invokes itself recursively on $T'$ and $F'$ (line 14).

ProveIt constructs an initial unwinding tree $T_0$ consisting of a single node $n$ that models the initial control location of $P$ (line 19), constructs an initial frontier containing only $n$ (line 20), and invokes ProveIt′ on $T_0$ and $F_0$ (line 21).

***Design of* UpdCover** The procedure UpdCover takes an unwinding tree $T$ as input and updates the cover relation of $T$. In particular, UpdCover performs a pre-order traversal of $T$. At each node $n \in N_T$, UpdCover checks if there is an uncovered non-descendant $n' \in N_T$ of $n$ such that (1) $n$ and $n'$ model the same control location of $P$ and (2) the invariant of $n$ entails the invariant of $n'$. If so, UpdCover adds a cover edge from $n$ to $n'$ and does not traverse any subtree of $n$.

### 5.5.1 Refine

Refine takes as input an unwinding tree $T$ and updates the invariants of $T$ to refute any path to a node that models the error location $\mathtt{L}_E$. Refine uses a procedure `RefineTree` (line 2–line 8), which takes as input an unwinding tree $T$ and a heap shape $S_R$ and updates the invariants of $T$ to be safe inductive invariants such that $S_R$ entails the invariant of the root of the tree. `RefineTree` invokes a *invariant learner* InvLearner on $T$, the root of $T$, and $S_R$ (line 3; possible implementations of InvLearner are described in §5.6). For each child $c$ of the root of $T$ (line 4), `RefineTree` constructs the abstract post-image of $S_n$ for the instruction connecting $r$ to $c$, denoted $S_c$ (line 5), and then invokes itself recursively on $T$ restricted to $c$ and $S_c$ (line 6). `RefineTree` returns the unwinding

**Input** : $(T, L)$: an unwinding tree
**Output** : $T$, updated with safe inductive invariants.

```
1  Procedure RefineTree(T, S_R)
2      r := Root(T) ;
3      S_r := InvLearner(T, r, S_R) ;
4      foreach c ∈ Children(r) do
5          S_c := AbsPost_D[Instr(r, c)](S_r) ;
6          T′_c := RefineTree(T|_c, S_c) ;
7      end
8      return ConsUnwind(r, S_r, {T′_c}_c) ;
9  return RefineTree(T, Ω) ;
```

**Algorithm 2:** Refine: takes as input an unwinding tree $T$ and updates the invariants in $T$ to refute all paths to nodes that model the error location $\mathtt{L}_E$.

tree constructed from the root node, the shape $S_r$, and the refined children of $r$ (line 8).

Refine invokes `RefineTree` on the input unwinding tree $T$ and the universal shape $\Omega$ for $D$ (line 9).

## 5.6 Invariant learners

In this section, we define two procedures that can be used to implement InvLearner, which is used in the definition of Refine. We first define a partial characterization function (§5.6.1) and validator (§5.6.2) from a fixed unwinding tree and node, which we use in the definition of both procedures. We then define the two procedures: one queries a shape-synthesizer oracle (§5.6.3) and the other queries a shape-characterization oracle (§5.6.4). We then compare practical issues of implementing each invariant learner, in terms of requiring an end-user to serve as its corresponding oracle (§5.6.5).

### 5.6.1 A partial characterization for an unwinding tree

**Input** : A heap $h \in \mathsf{Heaps}$.
**Output** : For fixed unwinding tree $T$, node $n \in N_T$ with strongest valid invariant $S \in \mathcal{S}_D$, returns either (1) a Boolean value denoting a decision as to whether $h$ must be in the language of the invariant for $n$ or (2) the value unk.

```
1  if h ∈ S then return True ;
2  if ⋁{post*[n′, e](h) ≠↑ | L(e) = L_E, n′ →* e} then
   return False ;
3  return unk
```

**Algorithm 3:** $\chi_{T,n,S}$: defines a partial characterization function for a fixed unwinding tree $T$, node $n \in N_T$, and strongest valid invariant $S$ for $n$.

algorithm 3 contains pseudocode for an algorithm $\chi_{T,n,S}$ that, for fixed unwinding tree $T$ with node $n \in N_T$ and strongest valid invariant $S \in \mathcal{S}$ for $n$, decides whether a given heap $h \in \mathsf{Heaps}$ must be in the invariant for $n$. $\chi_{T,n,S}$ checks if $h$ is in $S$, and if so, returns True (line 1). $\chi_{T,n,S}$ then checks if, for some node $e$ that models the error location $\mathtt{L}_E$, $h$ transitions to some heap on the sequence of instructions on the control path from $n$ to $e$, and if so, returns False (line 2). Otherwise, $\chi_{T,n,S}$ returns unk (line 3).

### 5.6.2 A shape validator for an unwinding tree

algorithm 4 contains pseudocode for a procedure $V_{T,n,S}$ that, for fixed unwinding tree $T$, node $n \in N_T$, and strongest valid invariant $S \in \mathcal{S}$ for $n$, validates an input candidate invariant $C \in \mathcal{S}$. $V_{T,n,S}$ first checks that $S$ is subsumed by $C$ and, if not, returns a heap that is accepted by $S$ but is not accepted by $C$ (line 2). $V_{T,n,S}$ then

**Input** : For shape domain $D$, a candidate shape $C \in \mathcal{S}_D$.
**Output** : One of: (1) a value IsValid denoting that $C$ is a valid invariant for $n$, (2) True paired with a heap that must be accepted by any invariant for $n$ but is not accepted by $C$, or (3) False paired with a heap that must not be accepted by any invariant for $n$ but is accepted by $C$.

```
1  switch ChkSub(S, C) do
2  |   case h ∈ Heaps: return (True, h) ;
3  endsw
4  NegCounters := ∅ ;
5  foreach {e | L(e) = L_E, n →* e}  do
6  |   switch ChkSub(AbsPost*[n', e](C), 0_D) do
7  |   |   case h ∈ Heaps: return (False, h) ;
8  |   endsw
9  end
10 return IsValid ;
```

**Algorithm 4:** $V_{T,n,S}$: for a fixed unwinding tree $T$, control node $n \in N_T$, and the strongest valid invariant $S$ for $n$, defines a validator for candidate shapes at $n$.

enumerates over the set of error nodes that are reachable from $n$ in $T$ (line 5). For each such error node $e$, if the abstract post-image of $C$ along the control path from $n$ to $e$ does not subsume $0_D$, then $V_{T,n,S}$ returns a heap $h$ accepted by the post-image (line 7). Otherwise, $V_{T,n,S}$ returns that $C$ is a valid invariant for $n$ (line 10).

#### 5.6.3 An invariant learner using a synthesizer oracle

Given access to a shape-*synthesizing* oracle $O$ for shape domain $D$, we can learn an optimal invariant for $n$ by running $O$ on the partial characteristic function for the current unwinding tree and $n$.

I.e., let $O \in \mathsf{SynOracles}_D$ be a synthesizer oracle for domain shape domain $D$. For each unwinding tree $T$, node $n \in N_T$, and strongest valid invariant $S \in \mathcal{S}$ for $n$:

$$\textsc{InvLearner}(T, n, S) = O(\chi_{T,n,S})$$

#### 5.6.4 An invariant learner using a characterization oracle

Given access to a shape-*characterizing* oracle $O$ for an automatically learnable shape domain $D$, we can learn an invariant for node $n$ in unwinding tree $T$ by running a learning procedure $P$ for $D$. The *total* characteristic function given to $P$ is implemented by running $O$ with access to the *partial* characteristic function defined by $T$ and $n$ (§5.6.1). The *validator* given to $P$ is implemented using the validator defined by $T$ and $n$ (§5.6.2).

I.e., let $\mathsf{ShapeLearner} \in \mathsf{ShapeLearners}_D$ be a learning algorithm for $D$ and let $O \in \mathsf{CharOracles}_D$ by a characterization oracle for $D$. For each unwinding tree $T$ and node $n \in N_T$ with strongest valid invariant $S$:

$$\textsc{InvLearner}(T, n, S) = \mathsf{ShapeLearner}(O(\chi_{T,n,S}), V_{T,n,S})$$

#### 5.6.5 Implementing invariant learners with end-users

***Validating the results of an oracles*** Both synthesizing oracles are characterizing oracles are defined as functions that always generate (the characteristic functions of) solutions of given optimal refinement inference problems. In practice, an end user serving as a synthesizing oracle may generate a shape that is not a valid refinement of a given partial characteristic function, or may not be optimal. We can extend the given implementations of invariant learners to validate the shapes provided by a user by invoking $V_{T,n,S}$, and rejecting shapes that are not valid refinements. Extending the invariant learner to validate that a shape provided by a user is optimal over all valid refinements depends on the optimality ordering of the particular shape domain, and may not always be feasible.

We can extend the invariant learner that uses a characterizing oracle to check if a given heap $h$ is defined by $\chi$ before passing $h$ to the user, precluding the possibility that the user will provide an unsound answer, and saving effort for the user.

***Advantages of using synthesizing oracles*** There are several potential advantages to developing an analysis that uses a synthesizing oracle:

1. The shape domain used by the analysis does not necessarily need to be automatically learnable.
2. It may be the case for some analysis problems that it is easier for a user to write a single shape that describes their solution, rather than serving as a characteristic function for their solution. In particular, this seems feasible if the analysis queries the user on the membership of large heaps.

***Advantages of using characterizing oracles*** Conversely, there are several potential advantages to developing an analysis that uses a characterizing oracle:

1. A user serving as a characterizing oracle does not need to formally understand a shape domain: they only need to accept or reject example concrete shapes.
2. Responses to Boolean queries can be more easily aggregated and weighted than across multiple users: an analysis that uses multiple users to serve as an oracle can simply ask all users to decide membership of a heap, and take its final result to be the result of a vote among all users. Combining multiple shapes provided by users as synthesizers may be feasible for some domains, but would likely be more complex.

### 5.7 PROVEIT Properties

In this section, we give the correctness properties of PROVEIT, namely its soundness (§5.7.1) and relative completeness (§5.7.2). Proofs of all theorems will be given in an extended version of this paper.

#### 5.7.1 Soundness

BH: soundness: verifier is sound, even when using unsound oracles

#### 5.7.2 Relative Completeness

BH: give result relating optimality ordering over shapes and invariants

#### 5.7.3 Relativization of Proofs

BH: Somesh: complete

## 6. Evaluation

DD: Need some text to introduce the whole 3-question setup and our goals with the evaluation Evaluation questions

1. What is the implementation effort required to use our technique?
2. What is practical impact of Oracle-Guided Heap Invariant Synthesis?
3. What is the overhead of our system, in terms of runtime and memory usage?

DD: Numbers, as they come in

## 7. Related Work

***Counter-example Guided Abstraction Refinement*** A broad class of verifiers of programs and transition systems have been proposed that implement *counterexample-example guided abstraction refinement (CEGAR)* [9]. The common structure of all of these analyses

is that they maintain an approximate model of the possible runs of a system, and refine the model until it represents a proof of correctness by iteratively (1) choosing a path of execution $p$ allowed by the model that, if feasible, constitutes a property violation, (2) refuting the feasibility of $p$, and (3) using the refutation to refine the paths of execution allowed by the model. Perhaps the CEGAR-based analysis that is most closely related to the one proposed in this work is actually a *theoretical* analysis that chooses program facts from which to construct a refutation by querying a *widening oracle* [5]. The key property of the oracle-guided analysis is that if there is sequence of widenings that the oracle can possible choose to cause the analysis to verify a program, then the analysis will eventually verify the program successfully. Because the oracle does not solve a distinct problem, but instead provides values to the analysis, it can be viewed as an agent of *angelic non-determinism* [7]. While PROVEIT also queries an oracle, the oracle solves a problem distinct from providing values to the analysis, namely an active learning problem over both positive *and negative* example graphs.

-predicate abstraction: predicates in logic can't describe shapes [4, 13, 14]

-interpolation: also builds an unwinding tree of invariants, but still can't describe shapes[2, 12, 15, 17]

-active learning, Mahdu's paper: only works on bounded tuples of lists [11]

***Shape Analysis*** -memory graphs: TVLA [18], SMG's[10]

-separation logic [8, 16]

## 8. Conclusion

## References


[1] SV-COMP 2015 - 4th international competition on software verification, 2015. http://sv-comp.sosy-lab.org/2015/.

[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, 2012.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2), 1987.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[5] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, 2002.

[6] J. Besombes and J. Marion. Learning tree languages from positive examples and membership queries. *Theor. Comput. Sci.*, 382(3), 2007.

[7] R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.

[8] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.

[10] K. Dudka, P. Peringer, and T. Vojnar. Byte-precise verification of low-level list manipulation. In *SAS*, 2013.

[11] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV*, 2013.

[12] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

[15] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.

[16] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[17] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.

[18] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 2002.