Learning Shape Analysis

Marc Brockschmidt[1], Yuxin Chen[2], Pushmeet Kohli[3], Siddharth Krishna[4], and Daniel Tarlow[5]

[1]Microsoft Research, [2]ETH Zürich, [3]Microsoft Research (now at Google DeepMind), [4]New York University, [5]Microsoft Research (now at Google Brain)

Abstract. We present a data-driven verification framework to automatically prove memory safety of heap-manipulating programs. Our core contribution is a novel statistical machine learning technique that maps observed program states to (possibly disjunctive) separation logic formulas describing the invariant shape of (possibly nested) data structures at relevant program locations. We then attempt to verify these predictions using a program verifier, where counterexamples to a predicted invariant are used as additional input to the shape predictor in a refinement loop. We have implemented our techniques in Locust, an extension of the GRASShopper verification tool. Locust is able to automatically prove memory safety of implementations of classical heap-manipulating programs such as insertionsort, quicksort and traversals of nested data structures.

1 Introduction

A number of recent projects have shown that it is possible to verify implementations of systems with complex functional specifications (e.g. CompCert [27], miTLS [6], seL4 [24], and IronFleet [19]). However, this requires highly skilled practitioners to manually annotate large programs with appropriate invariants. While there is little hope of automating the overall process, we believe that this annotation work could be largely automated.

A key problem in verification of heap-manipulating programs is the inference of formal data structure descriptions. Separation logic [33, 36] has often been used in automatic reasoning about such programs, as its frame rule favors compositional reasoning and thus promises scalable verification tools. However, the resulting techniques have often traded precision and soundness for automation [12], required extensively annotated inputs [20, 31, 35], or focused on the restricted case of singly-linked lists (often without data) [3, 5, 7, 9, 13, 17, 18, 29, 34].

We follow earlier work and infer likely invariants from observed program runs [14–16, 39–43]. At its core, finding a program invariant is searching for a general "concept" (in the form of a formula) that overapproximates all occurring program states. This is similar to many of the problems considered in statistical machine learning, and recent results have shown that program analysis questions can be treated as such problems [15,16,22,32,38–41]. With the exception of [32,38], these efforts have focused on numerical program invariants.

We show how to treat the prediction of formulas similarly to predicting natural language or program source code in Sect. 3. Concretely, we define a simple grammar for our abstract domain of separation logic formulas with (possibly

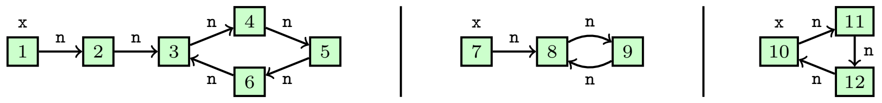

Fig. 1: Three heap graphs.

nested) inductive predicates. Based on a set of observed states, a formula can then be predicted starting from the grammar's start symbol by sequentially choosing the most likely production step. As our grammar is fixed, each such step is a simple classification problem from machine learning: "Considering the program states and the formula produced so far, which is the most likely production step?" Our technique can handle arbitrary (pre-defined) inductive predicates and nesting of such predicates, and can also produce disjunctive formulas.

We show how to use this technique in a refinement loop with an off-the-shelf program verifier (GRASShopper [35]) to automatically prove memory safety of programs in Sect. 4. We experimentally evaluate our approach in Sect. 5. There, we show that our shape analysis performs well on automatically generated synthetic data sets similar to our training data. Furthermore, we show that Locust is able to fully automatically verify programs from a standard test suite that are beyond the capabilities of other tools. Finally, we evaluate our method on a selection of programs handling nested data structures, which are at the core of much low-level code such as device drivers [5].

2 Example

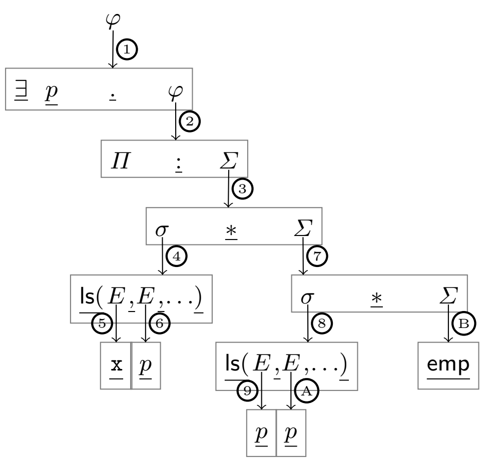

Fig. 2: Syntax tree of $\exists p.\Pi : \mathsf{ls}(\mathtt{x}, p, \ldots) * \mathsf{ls}(p, p, \ldots) * \mathsf{emp}$. Expansion of Π skipped, terminal symbols underlined, boxes indicate result of a single grammar production, circled indices indicate the order of productions.

Our central goal is to predict a separation logic formula describing the data structures used at a given program location from a set of observed program states. A core requirement is that the predicted formula should generalize well, i.e., also describe different, but structurally similar program states. For this, we first convert program states into *heap graphs*, in which memory locations are nodes, pointers are edges and program variables are node labels (we drop all non-heap information). As examples, consider the three graphs in Fig. 1, representing program states with a program variable x. These three heap graphs can be described by the separation logic formula $\exists p.\Pi : \mathsf{ls}(\mathtt{x}, p, \ldots) * \mathsf{ls}(p, p, \ldots) * \mathsf{emp}$. While we will discuss Π below, the remainder of the formula means that there is a heap location p such that there is a singly linked list from x to p and a disjoint list from p to itself. In this section, we discuss in detail how our method proceeds on the example graphs; the general method and technical details are discussed in the following sections.

Our method predicts this formula by constructing it iteratively, following its syntactic structure. We predict fromulas from a fragment of separation logic described by a grammar (cf. Fig. 4). The syntax tree for the predicted formula in this grammar is shown in Fig. 2. We generate formulas by starting with a singleton tree containing the grammar's start symbol and repeatedly expanding the leftmost leaf nonterminal in the syntax tree. At each step, the grammar allows only a few expansion rules, and we use a machine learning component to predict the next expansion step based on the partial syntax tree generated so far and the heap graphs provided as input. These predictions are made on features that represent general structural information about graph properties such as cyclicity, connections between labeled nodes, etc. This component is trained beforehand on a large amount of automatically generated, program-independent data (cf. Sect. 3.3). Thus, all of our predictions are based on learned patterns that were observed in the training data, and do not depend on hardcoded rules.

Initially, the syntax tree contains only φ. In production step ①, the root nonterminal φ can be expanded to either $\exists \mathcal{V}.\varphi$ or $\Pi : \Sigma$. Intuitively, choosing the former allows us to introduce a label for a node that we believe we will need to reference later in the procedure. To decide which production to choose, we extract a feature vector for each heap node that contains information about the number of incoming and outgoing edges and distance to other nodes with labels. Based on these features, our method predicts that we should introduce an existential quantifier for a fresh variable name (in this case p), and computes that it is most likely to refer to node 3 in the leftmost graph (resp. 8 in the second and 10 in the third) in Fig. 1. We attach the label p to these nodes for the remainder of the procedure, and extend the syntax tree according to the production $\exists p.\varphi$.

Next, in step ②, we expand the newly obtained φ nonterminal using the same procedure, but with a feature vector modified by the newly introduced label. This time, the production $\Pi : \Sigma$ is chosen. Π is a "pure" formula (i.e., not concerning heap shapes, but equalities between program variables and similar information), which we deterministically compute (cf. Sect. 3.3). We thus focus on Σ, the "spatial" formula describing heap shapes.

In step ③, the choice is between emp (implying that we believe that we are done describing the heap) and $\sigma * \Sigma$, which means that we predict that there are more heap regions to describe. We extract a feature vector summarizing structural knowledge about the heap graphs, (e.g., "are there nodes with in-degree i and out-degree j") and syntactic knowledge about the formula (e.g., "how many program variables have not been used yet in the formula"). Based on this, we predict that Σ should be expanded to $\sigma * \Sigma$, where σ is a "heaplet" that describes a single shape on the heap.

Now, in step ④, we choose whether the next heap region we describe is a list or a tree. We use similar features as for Σ to predict that σ should be expanded to $\mathsf{ls}(E, E, \ldots)$,[6] i.e., we predict that there is at least one list in the heap.

[6] We will discuss the role of $\ldots$ in Sect. 3.2.

The E (expression) nonterminals declare where this list begins and ends, and can be expanded to either a variable or the special 0 value. To make choices in steps ⑤ and ⑥, we extract a separate feature vector for each program variable and 0, again combining knowledge from the heap graphs (e.g., "are there nodes with in-degree i and out-degree j reachable from v") and from the partially generated formula (e.g., "has v already been used in the formula", ...). From these features, we predict the most likely identifier to expand E with. Our predictor chooses x here, but could equally well return p. Next, we need to expand the second E nonterminal. Here, we additionally consider a "reachable from syntactic sibling" feature, which allows our system to correctly rule out x and instead choose p.

The process continues for the remaining nonterminals in the same manner, using a frame inference to compute the *footprint* of already generated predicates $\mathsf{p}(v_1, \ldots, v_n)$ (i.e., heap nodes described by p). For instance, for the leftmost graph of Fig. 1, after predicting $\mathsf{ls}(\mathtt{x}, p, \ldots)$ we compute its footprint as $\{1, 2\}$. We use this information by restricting heap graph feature extraction to nodes outside of the footprint of already generated predicates; this provides enough information for the system to make progress and not get "stuck" predicting the same things repeatedly. Eventually (step Ⓑ), we predict that Σ should be expanded into emp, indicating the empty heap.

3 Predicting Shape Invariants from Heaps

In Sect. 3.1, we first present a general technique to predict derivations in a grammar G from a set of objects H, given functions that compute features from H. We then show how to apply this to our setting in Sect. 3.2, using a grammar for separation logic as G and heap graphs as input objects, and discuss the features used. Practical aspects of extending this core technique to a useful shape analysis tool (e.g., how to generate training data) are discussed in Sect. 3.3.

3.1 General Syntax Tree Prediction

Let G be a context-free grammar, $\mathcal{S}$ the set of all (terminal and nonterminal) symbols of G, and $\mathcal{N}$ just the nonterminal symbols. We assume that every sentence generated by G has a unique syntax tree, which we represent as a tuple $\mathcal{T} = (\mathcal{A}, g(\cdot), ch(\cdot))$ where $\mathcal{A} = \{1, \ldots, A\}$ is the set of nodes for some $A \in \mathbb{N}$, $g : \mathcal{A} \to \mathcal{S}$ maps a node to a terminal or nonterminal symbol from the grammar, and $ch : \mathcal{A} \to \mathcal{A}^*$ maps a node to its direct children in the syntax tree. A partial syntax tree $\mathcal{T}_{<a}$ is a syntax tree $\mathcal{T}$ restricted to nodes $\{1, \ldots, a-1\}$, where the ordering on nodes comes from the order in which they are predicted.

We assume there is an underlying unknown distribution $p(\mathcal{T} \mid H)$. This matches the observation that in our setting, there is no unique "correct" formula describing a set of heap graphs. Instead, many formulas (from the trivial "true" to formulas without inductive predicates, concretely describing the full observed heap) are valid candidates. Our problem is to learn this distribution, so that we can predict a syntax tree $\mathcal{T}$ given a set of objects H. As the set of valid syntax trees is extremely large, simply learning a mapping from inputs to a previously enumerated set of syntax trees is impractical. Instead, we learn this distribution following a technique that predicts source code from natural language [2]. The key idea is that instead of considering the probability of the full tree, we decompose the problem into learning the probability distributions for productions in our grammar, conditional on the inputs and the partial syntax tree generated so far.

$$p(\mathcal{T} \mid H) \simeq \prod_{\{a \in \mathcal{A} \mid g(a) \in \mathcal{N}\}} p(ch(a) \mid H, \mathcal{T}_{<a})$$

This decomposition allows us to treat the problem as a sequential prediction task in which we predict the syntax tree in a depth-first left-to-right node order. A further simplification step to aid learning is to not operate directly on input objects and syntax trees, but instead to compute a feature vector encoding existing domain knowledge $\boldsymbol{f} = \boldsymbol{\phi}_N(H, \mathcal{T}_{<a}) \in \mathbb{R}^{D_N}$ (where D_N is the number of features for N) that depends on the considered nonterminal N, the input objects H and the partial syntax tree $\mathcal{T}_{<a}$ generated so far.[7] The learned probability distribution is thus $p(\mathcal{T} \mid H) \simeq \prod_{\{a \in \mathcal{A} \mid g(a) \in \mathcal{N}\}} p(ch(a) \mid \boldsymbol{\phi}_{g(a)}(H, \mathcal{T}_{<a}))$.

We use two different models for these per-nonterminal probability distributions, depending on the production rules for N in G. If N has a fixed number of production rules in G (for example, $\varphi \to \exists \mathcal{V}.\varphi \mid \Pi : \Sigma$) then we view this as a standard multiclass classification task, i.e, where a probability is assigned to each allowed production ("class") based on a feature vector. If N can be expanded to any terminal from a dynamic set (for example E, which stands for any variable in scope at this point), then we instead learn a function that assigns a score to each production. We then obtain a probability distribution over the productions from these scores by a normalization procedure (see below). In both cases, we have a $\mathsf{Predictor}_N(\boldsymbol{f}; \theta)$ function for each nonterminal N that assigns probabilities to each production allowed by G based on the input feature vector. θ denotes learnable parameters of this function. In practice, we use a neural network with one fully connected layer for the classification tasks and a two layer network for the ranking tasks, such that θ consists of the weights used in each layer.

The pseudocode for this procedure PlatypusCore is given in Alg. 1, which is initially called with a syntax tree containing only the grammar's start symbol. Note that Alg. 1 is entirely independent of the semantics of the generated syntax tree. All domain knowledge about the meaning of the generated syntax and how it is related to the input objects needs to be encapsulated in the construction of $\boldsymbol{\phi}_N$, which extracts features to be used by the generic machine learning components. We discuss our choices for $\boldsymbol{\phi}_N$ below.

To train the overall system, we assume we are given a training set of $(\mathcal{T}, H)$ pairs drawn from the desired distribution (we discuss the details of this procedure for our setting in Sect. 3.3). To obtain training data for the individual $\mathsf{Predictor}_N$

[7] While we have experimented with avoiding this simplification to side-step the need for feature engineering by operating directly on input graphs [28], the resulting system was substantially harder to train and slightly less precise, as it had to learn domain knowledge from the training data.

Algorithm 1 Pseudocode for PlatypusCore (extension of [2])

```
Input: Grammar $G$, input objects $H$, (partial) syntax tree $\mathcal{T} = (\mathcal{A}, g, ch)$, nonterminal
    node $a$ to expand
1: $N \leftarrow g(a)$                                          {nonterminal symbol of $a$ in $\mathcal{T}$}
2: $\boldsymbol{f} \leftarrow \boldsymbol{\phi}_N(H, \mathcal{T}_{<a})$                          {compute features (see Sect. 3.2)}
3: $P \leftarrow$ most likely production $N \rightarrow \mathcal{S}^*$ from $G$ considering $\mathsf{Predictor}_N(\boldsymbol{f})$
4: $\mathcal{T} \leftarrow$ insert new nodes into $\mathcal{T}$ according to $P$
5: for all children $a' \in ch(a)$ labeled by nonterminal do
6:     $\mathcal{T} \leftarrow \mathsf{PlatypusCore}(G, H, \mathcal{T}, a')$
7: return $\mathcal{T}$
```

functions, we follow our PlatypusCore procedure. For each syntax tree node a labeled with a nonterminal N, we extract the feature vector $\boldsymbol{f} = \boldsymbol{\phi}_N(H, \mathcal{T}_{<a})$, but retrieve the chosen production rule P from the ground truth syntax tree $\mathcal{T}$ to generate a pair $(\boldsymbol{f}, P)$ which can be used to train the classifier or ranker for nonterminal N.

3.2 Predicting Separation Logic Formulas

To use the PlatypusCore algorithm for shape analysis, we need to specify the input objects H, the output grammar G, and the feature extraction function $\boldsymbol{\phi}_N$ and predictor $\mathsf{Predictor}_N$ for each nonterminal.

Inputs. Our inputs are directed—possibly cyclic—graphs representing the heap of a program and the values of program variables. Intuitively, each graph node v corresponds to an address in memory at which a sequence of pointers $v_0, \dots, v_t$ is stored.[8] Edges reflect these pointer values, i.e., v has edges to $v_0, \dots, v_t$ labeled with $0, \dots, t$. The node 0 is special (corresponding to the `null` pointer in programs) and may not have outgoing edges. Furthermore, we use unique node labels to denote the values of program variables $\mathcal{PV}$ and auxiliary variables $\mathcal{V}$, which can be introduced by existential quantification.

Definition 1 (Heap Graphs). *Let $\mathcal{PV}$ be a set of program variables and $\mathcal{V}$ be a set of (disjunct) auxiliary variables. The set of* Heap Graphs *$\mathcal{H}$ is then defined as* $2^{\mathbb{N}} \times 2^{(\mathbb{N}\setminus\{0\})\times\mathbb{N}\times\mathbb{N}} \times (\mathcal{PV} \cup \mathcal{V} \rightarrow \mathbb{N})$.

Outputs. We consider a fragment of separation logic [33,36]. Our method allows the *separating conjunction* $*$, list-valued *points-to* expressions $v \mapsto [e_1, \dots, e_n]$, existential quantification and higher-order inductive predicates [5], but no $-\!\!*$. As pure formulas, we only allow conjunctions of (dis)equalities, and use the constant 0 as the special null pointer. We will only discuss the singly-linked list predicate ls and the binary tree predicate tree in the following, though our method is applicable to generic inductive predicates. The following grammar describes our formulas, where nonterminals $\mathcal{V}$ and $\mathcal{PV}$ can be expanded to any terminal from the corresponding sets.

[8] Here, we discard non-pointer values.

$$\begin{array}{lll} \varphi ::= \exists\mathcal{V}.\varphi \mid \mathit{\Pi} : \mathit{\Sigma} & \mathit{\Sigma} ::= \mathsf{emp} \mid \sigma * \mathit{\Sigma} & \sigma ::= \mathsf{ls}(E, E, \lambda\mathcal{V}, \mathcal{V}, \mathcal{V}, \mathcal{V} \rightarrow \varphi) \\ \mathit{\Pi} ::= \mathit{true} \mid \pi \wedge \mathit{\Pi} & \pi ::= E = E \mid E \neq E & \quad \mid \mathsf{tree}(E, \lambda\mathcal{V}, \mathcal{V}, \mathcal{V}, \mathcal{V} \rightarrow \varphi) \\ E ::= 0 \mid \mathcal{V} \mid \mathcal{PV} & & \quad \mid \mathcal{V} \mapsto [E \dots E] \mid \mathcal{PV} \mapsto [E \dots E] \end{array}$$

Semantics are defined as usual for separation logic, i.e., $h \models \sigma_1 * \sigma_2$ for some $h = (V, E, \mathcal{L}) \in \mathcal{H}$ if h can be partitioned into two subgraphs h_1, h_2 such that h_1 (resp. h_2) is a model of σ_1 (resp. σ_2) after substituting variables in σ_1 and σ_2 according to $\mathcal{L}$. The empty heap emp is true only on empty subgraphs, and $v \mapsto [e_1, \dots, e_n]$ holds iff $V = \{v\}$ and for all $1 \leq i \leq n$, there is some edge (v, i, e_i). For detailed semantics, see [33,36]. The semantics of inductive predicates are the least fixpoint of their definitions, where nested formulas describe the shape of a nested data structure. For example, we define ls and tree as follows.

$$\begin{aligned} \mathsf{ls}(x, y, \varphi) \equiv& (x = y) \vee (\exists v, n.x \mapsto [v, n] * \mathsf{ls}(n, y, \varphi) * \varphi(x, y, v, n)) \\ \mathsf{tree}(x, \varphi) \equiv& (\exists v, l, r.l{\neq}0 \wedge r{\neq}0 : x \mapsto [v, l, r] * \mathsf{tree}(l, \varphi) * \mathsf{tree}(r, \varphi) * \varphi(x, v, l, r)) \\ &\vee (\exists v, r.r{\neq}0 : x \mapsto [v, 0, r] * \mathsf{tree}(r, \varphi) * \varphi(x, v, 0, r)) \\ &\vee (\exists v, l.l{\neq}0 : x \mapsto [v, l, 0] * \mathsf{tree}(l, \varphi) * \varphi(x, v, l, 0)) \\ &\vee (\exists v.x \mapsto [v, 0, 0] * \varphi(x, v, 0, 0)) \end{aligned}$$

Note that our definition of ls implies that $\mathsf{ls}(x, x)$ holds both for empty list segments as well as cyclic lists, and $\mathsf{tree}(x)$ implies that $x \neq 0$. We use $\top \equiv \lambda v_1, v_2, v_3, v_4 \rightarrow \mathit{true} : \mathsf{emp}$ to denote "no further nested data structure". Thus, $\mathsf{ls}(x, y, \lambda f_1, f_2, e_1, e_2 \rightarrow \mathsf{tree}(e_1, \top))$ describes a list of binary trees from x to y.

Example 2. A "pan-handle list" starting in i_2 is described by $\varphi(i_1, i_2, i_3, i_4) \equiv \exists p.\mathsf{ls}(i_2, p, \top) * \mathsf{ls}(p, p, \top)$, where an acyclic list segment leads to a cyclic list. Here, p is the existentially quantified node at which "handle" and "pan" are joined.

The formula $\psi(x) \equiv \mathsf{tree}(x, \varphi)$ describes a binary tree whose nodes in turn contain panhandle lists. An example of a heap satisfying the formula ψ is shown in Fig. 3. Blue nodes are elements of the tree data structure, having three outgoing edges labeled $0, 1, 2$. Each of the green boxes in Fig. 3 corresponds to a *subheap* that is described by the subformula φ. In each of these subheaps, one node is labeled with p, which is not a program variable, but introduced through the existential quantifier in φ.

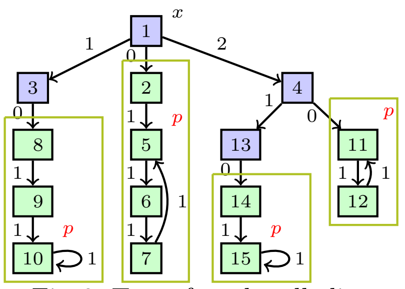

Fig. 3: Tree of panhandle lists

We found that our procedure PlatypusCore was often unnecessarily imprecise when generating the pure subformula $\mathit{\Pi}$ and $\mapsto$ atoms, which can simply be computed deterministically. Thus, we restrict our machine learning-based component to handle inductive predicates, and generate $\mathit{\Pi}$ deterministically using a nullness and aliasing analysis (see Sect. 3.3). While this can lead to predicting $\mathsf{ls}(x, y)$ even if x`.next` $= y$ in all observed states, our deterministic extension procedure then yields $x \neq y \wedge x$`.next` $= y$. Our grammar thus simplifies to Fig. 4, where $\mathit{\Pi}$ is now a terminal symbol.

$\varphi := \exists\mathcal{V}.\varphi \mid \Pi : \Sigma \qquad \Sigma := \mathsf{emp} \mid \sigma * \Sigma$

$E := 0 \mid \mathcal{V} \mid \mathcal{PV} \qquad \sigma := \mathsf{ls}(E, E, \lambda\mathcal{V}, \mathcal{V}, \mathcal{V}, \mathcal{V} \to \varphi) \mid \mathsf{tree}(E, \lambda\mathcal{V}, \mathcal{V}, \mathcal{V}, \mathcal{V} \to \varphi)$

Fig. 4: Grammar used by our Platypus procedure.

Predicting Flat Formulas We will first discuss the definitions of ϕ_N for the case where the input is a single graph h with nodes V, and predict formulas from a restricted separation logic grammar without nesting.

For any syntax node a, we define $\mathcal{I}(\mathcal{T}_{<a})$ as the set of identifiers that are in scope at point a in the partial syntax tree. Similarly, $\mathcal{D}(\mathcal{T}_{<a}) \subseteq \mathcal{I}(\mathcal{T}_{<a})$ is the set of "defined" identifiers that occur as first argument of any predicate, following the intuition that $\mathsf{ls}(x, y)$ and $SLtree(x)$ define the data structure starting at x.

An important class of features is based on the notion of *n-grams* of heap graphs. A 1-gram simply describes the in-degree and out-degree of some node v, i.e., is a pair $(indeg(v), outdeg(v)) \in \mathbb{N}^2$. n-grams extend this idea to a sequence of n connected nodes in a heap graph, e.g., a 2-gram is a pair of the 1-grams for two nodes connected by an edge in h. Based on this, we also define a refined measure of depth. For a path $v_1 \ldots v_t$ in the heap graph, we define its *1-gram depth* as the number of times the 1-gram changes, i.e., $|\{i \in \{1 \ldots t-1\} \mid indeg(v_i) \neq indeg(v_{i+1}) \vee outdeg(v_i) \neq outdeg(v_{i+1})\}|$. Then, $depth(v)$ is the minimal depth of paths leading from a node labeled by a variable in $\mathcal{PV} \cup \mathcal{V}$ to v. In our method, we extend 1-grams by this depth notion, i.e., represent each node by a $(indeg(v), outdeg(v), depth(v))$ triple. Intuitively, this information helps to discover the level of data structure nesting. As an example, consider Fig. 3 again. There, nodes 1, 10, 5, 15 and 11 have 1-gram depth 0, nodes 3, 4, 13, 6, 7 and 12 have 1-gram depth 1 (note that we haven't drawn the edges to 0 for some "tree" nodes), and nodes 8, 9, 2, and 14 have 1-gram depth 2.

As features of a heap graph, we use presence of n-grams in that graph, only considering the n-grams observed at training time. Thus for the graph in Fig. 3, we obtain $1gram_{(0,3,0)} = 1$ (cf. node 1), $1gram_{(1,3,1)} = 1$ (cf. node 3) and so on.

Σ, σ Nonterminals. Intuitively, the production choices for these nonterminals depend on the structure of the heap graph that has not been described by the partial syntax tree generated so far. As discussed above, we compute the footprint of (i.e., those heap nodes described by) already predicted predicates. We denote the set of nodes covered by predicates predicted up to syntax node a as $V_{<a}$.

Using this, we compute features for Σ, σ as the 1-grams and 2-grams from above restricted to the nodes $V \setminus V_{<a}$, i.e., those nodes that are not covered by the data structures described by the partial formula predicted so far. Their node degrees, contained in the 1-gram features, are indicators of the data structures present in the remaining heap. Additionally, we also include a feature reporting the number of identifiers not defined yet, i.e., $|\mathcal{I}(\mathcal{T}_{<a}) \setminus \mathcal{D}(\mathcal{T}_{<a})|$.

E Nonterminals. Here, we pick an expression as argument to a predicate. This decision depends on how well the part of the heap graph reachable from the expression matches the semantics of the surrounding predicate and possibly

already predicted other arguments. The set of legal outputs differs at each syntax node a: When making a prediction for E at node a, the set of legal outputs is $\mathcal{I}(\mathcal{T}_{<a}) \cup \{0\}$; i.e., the set of all identifiers that are in scope at this point and 0, which varies for each prediction. Thus, we treat this as a ranking task and, unlike the earlier case where we had a single feature vector, we compute one feature vector $\boldsymbol{f}_{E,z}$ for each $z \in \mathcal{I}(\mathcal{T}_{<a}) \cup \{0\}$.

To this end, we compute 1-gram and 2-gram features as above for each expression z separately, restricted to those heap graph nodes reachable from the node labeled by z. We also extract boolean features signifying if z is part of a non-trivial strongly connected component of the heap graph, or has a path to a strongly connected component. Additionally, to relate z to already predicted arguments of the same predicate, we define the sequence of "enclosing defined identifiers" $e_1, \ldots, e_t \in \mathcal{I}(\mathcal{T}_{<a})$, i.e., identifiers appearing in predicates enclosing the currently considered node a. As an example, consider the partially predicted formula $\mathsf{ls}(x, E, \ldots)$, where we are interested in predicting the expression at E. Here, we have $e_1 = x$ (for nested data structures, $e_2, \ldots$ correspond to the identifiers chosen in the outer data structures). We use a boolean feature to denote reachability of (resp. from) each e_1, …, e_t from (resp. of) z.

Furthermore, we refine our notion of reachability. We say that v "reaches" v' if there is path $v = v_0, \ldots, v_t = v'$ in the heap graph. If furthermore no $v_1 \ldots v_{t-1}$ is labeled by another identifier, we say that v "directly reaches" v'. If all edges used on the path have the same label, then we say that v "simply reaches" v'. Finally, we also say that x "syntactically reaches" y if our partial syntax tree $\mathcal{T}_{<a}$ contains a predicate $\mathsf{p}(x, \ldots, y, \ldots)$.

Thus, for an identifier z labeling heap graph node v, we use these features:

- The frequency of 1- and 2-grams reachable from v.
- v is part (resp. reaches a node that is part) of a strongly connected component of the graph.
- v reaches (resp. reaches directly, simply, or syntactically) the enclosing identifier e_i for $i = 1 \ldots t$.
- v is reached (resp. reached directly, simply, or syntactically) by the enclosing identifier e_i for $i = 1 \ldots t$.

To implement $\mathsf{Predictor}_E$, we use a neural network NN (with learnable parameters θ^E) to compute scores $s_z = \mathrm{NN}(\boldsymbol{f}_{E,z}; \theta^E)$ for each identifier. We normalize these scores using the softmax function to get a probability distribution over identifiers (a common trick to reduce the influence of outliers). The probability of expanding E by z is thus $p(z) = \frac{\exp(s_z)}{\sum_{z' \in \mathcal{I}(\mathcal{T}_{<a}) \cup \{0\}} \exp(s_{z'})}$.

φ *Nonterminals.* Here, we need to decide whether to declare new identifiers via existential quantification, so that we can refer to nodes not labelled by program variables (e.g., for panhandle lists). Thus, we not only predict *that* we need a quantifier, but also by which graph node it should be instantiated. To use this information later on, we allow modifying the input H after a production step (between line 4 and 5 of $\mathsf{PlatypusCore}$). In this case, we add the newly introduced identifier as a label to the corresponding node.

We thus predict, independently for each node $v \in V$, the probability that it is referred to by a new existential variable. We proceed similar to the E case and compute a feature vector $\boldsymbol{f}_{\varphi,v}$ for each node $v \in V$. As features we again use standard graph properties, such as membership in a strongly connected component, existence of labels for a node, its in-degree and out-degree. Additionally, we also use features comparing these values to each nodes' direct neighbors, i.e., "has higher in-degree than average in-degree of neighbors".

To make a prediction, we use a neural network NN (with learnable parameters θ^{φ}) to compute a score $s_v = \mathrm{NN}(\boldsymbol{f}_{\varphi,v}; \theta^{\varphi})$. Unlike the E case where we have to choose one option from many, here each v is an independent decision, and so we use the sigmoid function[9] $p(v) = \frac{\exp(s_v)}{1+\exp(s_v)})$ to get the probability that v is labelled by a new identifier. When choosing a production for φ, we thus compute probabilities for each v independently and return $\arg\max_v p(v)$ as the probability of declaring a fresh identifier. If more identifiers are required, they can be added in subsequent grammar expansion steps.

Predicting Nested Formulas We now discuss the general case, in which we have several input heap graphs H, and data structures may in turn contain other data structures. This requires us to make predictions that are based on the information in all graphs, and sometimes on several subgraphs of each of the graphs. As an example, consider again the heap in Fig. 3, and imagine that we have successfully predicted the outer part of the corresponding formula, i.e., $\mathsf{tree}(x, \lambda i_1, i_2, i_3, i_4.\varphi)$, and are now trying to expand φ. This subformula needs to describe all the subheaps corresponding to the contents of the green boxes in Fig. 3. Again, we modify the input H to reflect the newly introduced identifiers. So for our example, we would replace H with one heap graph with labels $\{x \mapsto 1, i_1 \mapsto 3, i_2 \mapsto 8, i_3 \mapsto 0, i_4 \mapsto 0\}$ for the leftmost box, one with labels $\{x \mapsto 1, i_1 \mapsto 1, i_2 \mapsto 2, i_3 \mapsto 3, i_4 \mapsto 4\}$ for the second box, and so on.

Everything but φ Nonterminals. We use the same features from Sect. 3.2, but lift them to handle a set of heap graphs H. We compute feature vectors for each heap graph independently as before, and then *merge* them into a new single feature vector by computing features based on the maximum f_{max}, minimum f_{min}, and average value f_{avg} across all H for each feature f. We also use the same $\mathsf{Predictor}_N$ functions.

φ Nonterminals. This covers the case in which we predict that we need to insert an existential quantifier. In Fig. 3, this is the prediction of $\exists p$, where p corresponds to one node in each of the green boxes. We again lift the feature extraction as mentioned above, but as the number of nodes may differ between the different heap graphs, we cannot simply lift the $\mathsf{Predictor}_{\varphi}$ from above.

This problem is a basic form of the structured prediction problem [4]. Suppose there are R heap graphs. For each of the graphs, there is a set of nodes V_r which

[9] Note that the softmax function used in the E case is the generalization of the sigmoid function to many values.

may require an existential quantifier to be described in our setting (in Fig. 3, these are the contents of the green boxes). Let y_v be a boolean denoting the event that a new identifier is introduced for node v. We train a neural network like in the single-heap case so that the probability of introducing an existentially quantified variable for node v is $p(y_v = 1) = \frac{\exp(s_v)}{1+\exp(s_v)}$, where s_v is the score output by the neural network.

We now need to compute the probability of introducing a new identifier (for all graphs) in terms of the scores s_v (which only take one graph into account). We first set the probabilities of illegal events (i.e., predicting that one graph requires an existential quantifier, but another one does not) to 0. Then, the probability of not declaring a variable is $\prod_{1\leq r\leq R}\prod_{v\in V_r}(1 - p(y_v = 1)) = \prod_{1\leq r\leq R}\frac{1}{Z_r}$, where $Z_r = \prod_{v\in V_r}(1 + \exp(s_v))$. The probability of the event y_v^r of selecting exactly node v from graph r is

$$p(y_v^r = 1) = \frac{\exp(s_v)}{1 + \exp(s_v)} \prod_{v'\in V_r, v'\neq v} \frac{1}{1 + \exp(s_{v'})} = \frac{\exp(s_v)}{Z_r}.$$

As the choice of node from each graph is independent given that we are declaring a new identifier, the probability of choosing the set of nodes $\{v_r\}_{1\leq r\leq R}$ is the product $\prod_{1\leq r\leq R}\frac{\exp(s_{v_r})}{Z_r}$. Noting that all legal joint configurations have the same denominator $\prod_r Z_r$, we can drop the denominator and compute the normalizing constant for the constrained space later. The total unnormalized probability of declaring a variable is the sum of the unnormalized probabilities of all ways to choose exactly one node from each graph r, which can be rewritten as $\prod_{1\leq r\leq R}\sum_{v\in V_r}\exp(s_v)$. Normalising this, the probability of not introducing an existential quantifier is $\frac{1}{1+\prod_{1\leq r\leq R}\sum_{v\in V_r}\exp(s_v)}$ while the probability of introducing an existential is $\frac{\prod_{1\leq r\leq R}\sum_{v\in V_r}\exp(s_v)}{1+\prod_{1\leq r\leq R}\sum_{v\in V_r}\exp(s_v)}$.

To make predictions for all graphs at the same time, we use the above to decide whether to introduce an existential quantifier. If not, we choose the $\Pi : \Sigma$ production. If we decide to use the $\exists\mathcal{V}.\varphi$ production, then we draw one node from each graph according to a softmax over the scores; i.e., the probability of choosing node v in graph r is $\frac{\exp(s_v)}{\sum_{v'\in V_r}\exp(s_{v'})}$.

3.3 Shape Analysis with Platypus

To obtain a shape analyzer, we have extended the procedure PlatypusCore to also produce disjunctive invariants and deterministically compute pure subformulas. Finally, whereas Alg. 1 selects a production "greedily" in line 3 (i.e., it will always pick the most likely one), we have generalized this behavior to instead also sample productions using the probabilities obtained from the Predictor$_N$ function. This allows to iteratively obtain more and more formulas from Platypus, recovering from cases where the system is uncertain about the correct formula.

Training the Analyzer Training the logistic regressors and neural networks from above requires large amounts of training data, i.e., sets of heap graphs labeled with corresponding formulas. To obtain this data, we generate synthetic data by fixing a small set of program variables $\mathcal{PV}$ (typically of size 2 or 3) and enumerate semantically valid derivations of formulas in our grammar, similar in spirit to [23]. Then, we enumerate models for each formula by expanding inductive predicates until only $\mapsto$ atoms remain. From this we read off heap graphs by resolving the remaining ambiguous possible equalities between variables. The result is a set of pairs (φ, H) such that $h \models \varphi$ for every $h \in H$. We compute the unique syntax tree $\mathcal{T}_\varphi$ of each φ to get the desired training data pairs $(\mathcal{T}_\varphi, H)$.

Pure Subformulas We use a deterministic procedure to expand the nonterminal Π describing the pure part of our formulas, using simple aliasing and nullness analyses. Namely, for all pairs of identifiers $x, y \in \mathcal{PV} \cup \{0\}$, we check if $x = y$ or $x \neq y$ holds in all input heap graphs. Similarly, for all fields f and $x \neq 0$, we consider the possible equalities $x\,.\,f = y$. Π is then set to the conjunction of all (dis)equalities that hold in all input graphs.

Handling Disjunctions We found *disjunctive* separation logic formulas to be needed even for surprisingly simple examples, as in many cases, the initial or final iteration of a loop requires a different shape description from all other steps. In our setting, the problem of deciding how many and what disjuncts are needed can be treated as a clustering problem of heap graphs. In machine learning, the clustering problem is the task of grouping a set of samples in such a way as to group "similar" samples together. The notion of simlarity depends on the target application and is normally defined through a distance measure. A widely used and effective clustering method is k-means clustering, where the aim is to find k cluster centers such that the sum of the distance of every point to the closest cluster center is minimized.

For our setting, we convert the input heap graphs into feature vectors capturing reachability between program variables and use the Euclidean distance between these feature vectors as a distance measure between graphs. Following our notion of different kinds of reachability from above, we define a function $r_h(u, v)$ ranging from 0 if there is no connection between the nodes labeled by u and v and 1 if u and v are labels on the same node, with steps for different kinds of reachability. Using this function, we define $\boldsymbol{f}_h$ as the vector $\langle r_h(u, v)\rangle_{u,v\in\mathcal{PV}}$ for some fixed order on $\mathcal{PV}$. In our implementation, we run the clustering algorithm for $k \in 1..5$ and predict formulas for all generated clusterings.

4 Refining and Verifying Shape Invariants

We construct our fully automatic memory safety verifier Locust (pseudocode in Alg. 2) by connecting our shape predictor from Sect. 3 with the program verifier GRASShopper. For this, we keep a list of *positive* $S^+(\ell)$ and *negative*

Algorithm 2 Pseudocode for Locust

```
Input: Program $P$ and entry procedure $p$ with precondition $\varphi_p$, locations $L$ requiring
    program annotations
 1: $I \leftarrow$ sample initial states satisfying $\varphi_p$                                   {see Sect. 4.1}
 2: $S^+ \leftarrow$ execute $P$ on $I$ to map location $\ell \in L$ to set of observed states
 3: while true do
 4:     for all $\ell \in L$ do
 5:         while true do
 6:             $\varphi'_\ell \leftarrow$ obtain fresh formula sample from Platypus($S^+(\ell)$)
 7:             if exists $\varphi'_\ell$ consistent with all $S^+(\ell), S^-(\ell)$ then        {see Sect. 4.3}
 8:                 $\varphi_\ell \leftarrow \varphi'_\ell$
 9:                 break (continue on line 4)
10:     $P' \leftarrow$ annotate $P$ with inferred $\varphi_\ell$
11:     if GRASShopper($P'$) returns counterexample $s$ then
12:         if $s$ is new counterexample then
13:             update $S^+, S^-$ to contain $s$ for correct location            {see Sect. 4.2}
14:         else return FAIL
15:     else return SUCCESS
```

state samples $S^-(\ell)$ for every program location ℓ at which program annotations for GRASShopper are required (i.e., loop invariants and pre/post-conditions for subprocedures). We first sample initial states (cf. Sect. 4.1) and use these to collect a first set of positive samples corresponding to valid program runs by simply executing the program. Then we obtain a set of candidate formulas from Platypus for each location and enter a refinement loop. If verification using these candidates fails, we get a counterexample state at some location ℓ, which we use to extend the sets $S^+(\ell)$ and $S^-(\ell)$. As it is possible that no correct set of program annotations can be found (due to an incorrect program or imprecisions in our procedure), we report failure when the same counterexample is reported for the second time (i.e., we have stopped making progress).

To simplify the procedure, we assume that Platypus always returns the most precise formula from our abstract domain holding for the given set of input heap graphs (†). While this assumption is not formally guaranteed, Platypus was trained to produce this behavior (by choosing training data according to this principle), and we have observed that it behaves like this in practice.

4.1 Initial State Sampling

We assume the existence of some set of preconditions describing the input to the main procedure of the program in separation logic.[10] To sample from these preconditions, we can add `assert false` to the beginning of the program. Then, every counterexample returned by GRASShopper is a model of the precondition.

[10] Conceivably, these could be provided by users in a pre-formal language and translated to separation logic using an interactive elaboration procedure. Alternatively, given a test suite, Platypus could predict the initial precondition as well.

To get more samples, and to ensure different sizes of input lists, we add cardinality constraints to the precondition. For example, to force a list starting at `lst` to have length ≥ 3, we add `requires lst.next.next != null`. States at other locations are then obtained by executing the program from the initial sample.

While this strategy is complete relative to the fragment of separation logic supported by GRASShopper, it is slow even for simple preconditions. Thus, we have implemented a simple heuristic sampling algorithm for preconditions using only simple predicates. If we detect that a precondition is simple enough for our heuristic, we use it instead to generate sample states of varying sizes.

4.2 Handling Counterexamples

If the program is incorrect, or the current annotations are incorrect or insufficient to prove the program correct, then GRASShopper returns a counterexample at a location ℓ. Depending on the context of such a counterexample and its exact form, we treat it as a positive or negative program state sample as follows.

- Case 1: A candidate invariant does not hold on loop entry. The counterexample state is reachable, but is not covered by the candidate loop invariant, and thus, the counterexample can be added as a positive sample to $S^{+}(\ell)$.
- Case 2: A candidate loop invariant is not inductive. This is an implication counterexample [15, 40], i.e., a state s that is a model of the candidate loop invariant and a state s' reached after evaluating the loop body on s. Based on our assumption (†), we conclude that s is likely to be a reachable state, and thus s' is. Hence, we treat s' as a positive sample and add it to $S^{+}(\ell)$.
- Case 3: A postcondition does not hold for a state s. Again, by (†), we conclude that s is a reachable state, and thus add the counterexample to $S^{+}(\ell)$.
- Case 4: Invalid heap access inside the loop. The counterexample state is consistent with the candidate loop invariant, but triggers an invalid heap access such as a `null` access. It is a negative sample and is added to $S^{-}(\ell)$.

4.3 Consistency Checking

For each prediction returned by the predictor, we check its consistency with the positive and negative samples obtained so far. This is needed because Platypus cannot provide correctness guarantees, and does not make use of negative samples. Thus we check each returned formula φ_ℓ for consistency with the observed samples, i.e., $\forall h \in S^{+}(\ell).h \models \varphi_\ell$ and $\forall h \in S^{-}(\ell).h \not\models \varphi_\ell$. As in our sampling strategy, we use the underlying program verifier for this. For this, we translate a state h into a formula φ_h that describes the sample h exactly, by introducing variables n_v for each node v and representing each edge (n, f, n') as $n.f \mapsto n'$. Then h is a model of φ_ℓ iff all models of φ_h also satisfy φ_ℓ. However, since by construction φ_h only has the model h, this is equivalent to checking if $\varphi_h \wedge \varphi_\ell$ has a model. This can be checked using a complete program verifier such as GRASShopper by using $\varphi_h \wedge \varphi_\ell$ as precondition of a procedure whose body is `assert false`.

5 Related Work & Experiments

We implemented the procedure PlatypusCore from Alg. 1 as a stand-alone tool Platypus in F#, also containing the feature extraction routines and support for data generation. The core machine learning models ($\mathsf{Predictor}_*$) are implemented in TensorFlow, using a small Python wrapper. Finally, we have extended GRASShopper [35] with the procedure from Sect. 4. The source code for Locust and Platypus is available at `https://github.com/mmjb/grasshopper`.

Limitations. As our method relies on a trained machine learning component, we cannot give any completeness guarantees. However, our integration with a program verifier checks that returned results are correct. This means that our performance depends on that of the underlying verifier, and in fact, time spent in GRASShopper dominates verification time. As our verification technique relies on observing a sample of occurring program states, it is sensitive to the choice of input samples (randomly sampled, taken from a test suite, or provided by a human) used in the sample collection phase. However, this is a limitation shared by other dynamic analysis systems, such as Daikon [14].

5.1 Related Work

Memory safety proofs have long been a focus of research, and we only discuss especially recent and close work here. (Bi)-abduction based shape analyses [10–12, 25, 26] have been used successfully in memory safety proofs, and can also be used to abduce needed preconditions or the required inductive predicates. In another recent line of work, forest automata have been used to verify heap-manipulating programs [1], but require hard-coded support for specific data structures.

In property-directed shape analysis [21], predicate abstraction over user-provided shape predicates ((sorted) list segments, ...) is combined with a variation of the IC3 property-directed reachability algorithm [8] to prove memory safety and data properties. This can be viewed as continuation of three-valued logic-based works (e.g. [37]), reducing the data type specification requirements. Similary, SplInter extends the Impact [30] safety prover with heap reasoning based on an interpolation technique for separation logic. Finally, we note that in prior work on shape analysis for nested data structures [5], abstractions of heap graphs using inductive predicates were found using manual heuristics and enumerative search routines. Core parts of our method could be adapted to replace these by directed, learned search.

A recent line of work is the use of machine learning techniques for inferring *numerical* invariants [15, 16, 22, 39–41]. In these methods, a machine learning model such as a decision tree is trained on observed program states using standard optimization techniques, and the trained model is interpreted as program invariant. However, while the translation from separating hyperplanes or decision trees to standard invariant formats is straightforward for numerical data, no such correspondence exists in the domain of heap data. In our approach, a predictor is trained offline, independent of the considered programs, and at test

Table 1: Precision of Platypus on synthetic data

Dataset	Greedy	Precision @1	@5	@10
Trained on 3 var., no nesting data:				
3 var., no nesting	92.68%	86.88%	91.30%	96.43%
Trained on 2 var., nested data:				
3 var., no nesting	70.50%	69.37%	73.36%	78.40%
2 var., with nesting	24.11%	24.53%	33.42%	34.19%

(i.e., verification) time produces invariant candidates. Thus, our method cannot make use of negative examples obtained at test time, but does not require a close correspondence between the structure of the learned model and target invariants. Closest to this work is [32] which infers likely heap invariants from program *traces* (i.e., it infers shapes from usage patterns) using machine learning techniques.

5.2 Platypus Experiments

To evaluate Platypus itself, we have generated two large data sets following our procedure from Sect. 3.3. The first set contains all formulas we enumerate for three variables without using nested predicates (327 formulas in total), and the second set contains a random sample of 4% of the 36822 formulas we enumerate for two variables with one data structure nesting level (i.e., we used 1472 formulas in total). For each formula, we have generated 500 models (for the nested dataset, we subsampled this again, picking 100 of the generated states at random). We split both datasets into training, validation and test sets using a 3:1:1 split along the formulas (i.e., no formula appeared in both training and test sets).

For our evaluation, we run Platypus on groups of 5 states generated as models for the same formula at a time, producing 10 formula predictions for each group of states. Note that due to our formula-based split into training and test sets, both the tested states as well as the corresponding ground truth formula have not been seen by the system before. As checking logical equivalence between the formulas produced and the ground truth formula is expensive, we instead approximate this by canonicalizing variable names and the order of commutative elements in the formulas before comparing for exact (string) equality. We report accuracy of our "greedy" mode (i.e., the result obtained by always picking the most likely production) as well as top K accuracy (i.e., how often the correct formula was in the K most probable formulas from the set of 10 sampled formulas) for $K \in \{1, 5, 10\}$, and display the results in Tab. 1.

Furthermore, we evaluate the accuracy of the per-nonterminal predictors, using a model trained on the two variable with nesting dataset and tested on the three variable no nesting dataset. Tab. 2 reports how often the production rule predicted with highest probability (i.e., the one chosen in our

Table 2: Precision per nonterminal.

Nonterminal	Accuracy
φ	73.03%
Σ	99.86%
σ	99.70%
E	87.66%

"greedy" syntax tree sampling strategy), using features extracted under the assumption that all prior nodes of the syntax tree were predicted correctly, is indeed correct.

Analysis. We observe that on data structures without nesting, Platypus performs very well, and that it generalizes reasonably well from one dataset to another. Most notably, we found that generalizing to a larger number of program variables posed no problem at all. Most wrong predictions are due to wrongly predicting the need for existential quantifiers, or wrongly identifying the heap graph nodes corresponding to these.

While performance on nested data structures is less encouraging, a detailed analysis yielded that most mistakes occurred on formulas that are unlikely to appear in practice (such as $\mathsf{tree}(x, \lambda t, l, d, r \to \mathsf{ls}(d, l, \top))$, where each tree element has a data field containing a list to its left child). In experiments involving more realistic data structures (cf. below), we observed no such problems.

5.3 Locust Experiments

To validate that we can infer formulas for algorithms used in practice, we have evaluated Locust on a number of standard example programs. For this, we consider all example programs processing singly-linked lists with integer data distributed with GRASShopper. These include standard algorithms such as list traversal, filtering and concatenation, as well as more complex algorithms such as quicksort, mergesort and insertionsort. Furthermore, we considered four simple traversal routines of nested list/tree data structures. We again use our model trained on the two variable with nesting dataset and compare Locust as a memory safety prover to S2/HIP [25, 26], Predator [13] and Forester [1]. The full set of results is displayed in Tab. 3, where a ✓ indicates that a tool was successful, and ✗ that it failed (either explicit failure or timeout after 300s). For Platypus, we also note the number of disjuncts in generated invariants, and for Locust the number of iterations in the counterexample-refinement loop.

Analysis. Our results indicate that generalization from synthetic data used to train Platypus to programs works well. For example, whereas our training data was restricted to two program variables, the most complex program example, `strand_sort`, required an invariant involving six variables. In most cases, our strategy of sampling initial program states is sufficient, but the `merge_sort` and `strand_sort` examples show that additional counterexamples are indeed useful to generalize predictions. Finally, as Locust is not optimized for time (e.g., each Platypus invocation starts a .Net VM and initializes a Python interpreter) we do not report detailed runtimes. However, the core shape analysis (factoring out these startup times) took around a second for these benchmark programs.

6 Conclusion & Future Work

We have presented a new technique for data-driven shape analysis using machine learning techniques, which can be combined with an off-the-shelf program verifier

Table 3: Results of memory safety provers on GRASShopper benchmarks.

Example	Platypus	Locust	S2/HIP	Forester	Predator
`concat`	✓ (1 disj.)	✓ (1 it.)	✓	✗	✓
`copy`	✓ (1 disj.)	✓ (1 it.)	✗	✓	✓
`dispose`	✓ (1 disj.)	✓ (1 it.)	✗	✓	✓
`double_all`	✓ (1 disj.)	✓ (1 it.)	✗	✓	✓
`filter`	✓ (2 disj.)	✓ (1 it.)	✗	✓	✓
`insert`	✓ (1 disj.)	✓ (1 it.)	✗	✓	✓
`insertion_sort`	✓ (2 disj.)	✓ (1 it.)	✗	✓	✓
`merge_sort`	✓ (3 disj.)	✓ (4 it.)	✗	✗	✗
`pairwise_sum`	✓ (1 disj.)	✓ (1 it.)	✗	✓	✓
`quicksort`	✓ (1 disj.)	✓ (1 it.)	✗	✗	✗
`remove`	✓ (2 disj.)	✓ (1 it.)	✗	✓	✓
`reverse`	✓ (2 disj.)	✓ (1 it.)	✗	✓	✓
`strand_sort`	✓ (3 disj.)	✓ (5 it.)	✗	✓	✓
`traverse`	✓ (1 disj.)	✓ (1 it.)	✓	✓	✓
`ls_ls_trav`	✓ (1 disj.)	✓ (1 it.)	✗	✗	✗
`ls_ls_trav_rec`	✓ (1 disj.)	✓ (1 it.)	✗	✗	✗
`tr_ls_trav`	✓ (1 disj.)	✓ (1 it.)	✗	✗	✗
`ls_tr_trav`	✓ (1 disj.)	✓ (1 it.)	✗	✗	✗

to automatically prove memory safety of heap-manipulating programs. All of our contributions have been implemented in our tool Locust, whose experimental evaluation shows that it is able to automatically prove memory safety of programs that other state-of-the-art tools fail on.

Future Work. We plan to extend this work in three aspects. Firstly, we aim to extend Locust to support the introduction of existential quantifiers that Platypus allows. Secondly, one aspect of Platypus that still requires manual and skilled work is feature extraction, which can make extending the tool to handle new inductive separation logic predicates precisely hard. We would like to automate the extraction of relevant features for each production rule, and have already made steps in this direction. We recently introduced Gated Graph Sequence Neural Networks [28] — a technique that leverages deep-learning techniques to make the predictions directly on graph-structured inputs instead of feature vectors. We plan to integrate this into our framework. Initial tests have shown promising results, but some of the features supported by Platypus (most significantly, disjunctive formula predictions) are not yet available in this new method. Finally, we are interested in integrating our method with interactive program verification assistants, to support verification engineers in their daily work.

Acknowledgements We thank Martin Hruška and Quang Loc Le for help with running their tools S2/HIP, Forester, and Predator for the experiments. We also thank Thomas Wies for valuable feedback on drafts of this paper.

References

1. P. A. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. *Acta Inf.*, 53(4):357–385, 2016.
2. M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *ICML '15*, pages 2123–2132, 2015.
3. R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL '11*, pages 599–610, 2011.
4. G. H. Bakir, T. Hofmann, B. Schölkopf, A. J. Smola, B. Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data*. MIT Press, 2007.
5. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV '07*, pages 178–192, 2007.
6. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *SP '13*, pages 445–459, 2013.
7. A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI '12*, pages 1–22, 2012.
8. A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI '11*, pages 70–87, 2011.
9. J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE '11*, pages 131–146, 2011.
10. J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *SAS '14*, pages 68–84, 2014.
11. J. Brotherston, N. Gorogiannis, and R. L. Petersen. A generic cyclic theorem prover. In *APLAS '12*, pages 350–367, 2012.
12. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26, 2011.
13. K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV '11*, pages 372–378, 2011.
14. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
15. P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV '14*, pages 69–87, 2014.
16. P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL '16*, pages 499–512, 2016.
17. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS '06*, pages 240–260, 2006.
18. C. Haase, S. Ishtiaq, J. Ouaknine, and M. J. Parkinson. SeLoger: A tool for graph-based reasoning in separation logic. In *CAV '13*, pages 790–795, 2013.
19. C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *SOSP '15*, pages 1–17, 2015.
20. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV '13*, pages 756–772, 2013.
21. S. Itzhaky, N. Bjørner, T. W. Reps, M. Sagiv, and A. V. Thakur. Property-directed shape analysis. In *CAV '14*, pages 35–51, 2014.

22. Y. Jung, S. Kong, C. David, B. Wang, and K. Yi. Automatically inferring loop invariants via algorithmic learning. *MSCS*, 25(4):892–915, 2015.
23. A. J. Kennedy and D. Vytiniotis. Every bit counts: The binary representation of typed data and programs. *JFP*, 22(4-5):529–573, 2012.
24. G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *TOCS*, 32(1):2, 2014.
25. Q. L. Le, C. Gherghina, S. Qin, and W. Chin. Shape analysis via second-order bi-abduction. In *CAV '14*, pages 52–68, 2014.
26. Q. L. Le, J. Sun, and W. Chin. Satisfiability modulo heap-based programs. In *CAV '16*, pages 382–404, 2016.
27. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
28. Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR '16*, 2016.
29. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL '10*, pages 211–222, 2010.
30. K. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136, 2006.
31. Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *J. Symb. Comput.*, 45(11):1184–1211, 2010.
32. J. T. Mühlberg, D. H. White, M. Dodds, G. Lüttgen, and F. Piessens. Learning assertions to verify linked-list programs. In *SEFM '15*, pages 37–52, 2015.
33. P. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01*, pages 1–19, 2001.
34. J. A. N. Pérez and A. Rybalchenko. Separation logic modulo theories. In *APLAS '13*, pages 90–106, 2013.
35. R. Piskac, T. Wies, and D. Zufferey. GRASShopper - complete heap verification with mixed specifications. In *TACAS '14*, pages 124–139, 2014.
36. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, 2002.
37. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
38. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV '14*, pages 88–105, 2014.
39. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP '13*, pages 574–592, 2013.
40. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS '13*, pages 388–411, 2013.
41. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV '12*, pages 71–87, 2012.
42. G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA '06*, pages 145–156, 2006.
43. H. Zhu, G. Petri, and S. Jagannathan. Automatically learning shape specifications. In *PLDI '16*, 2016.