Learning to Verify the Heap

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

[1] Microsoft Research
[2] ETH Zürich
[3] University College London
[4] New York University
[5] Purdue University

Abstract. We present a data-driven verification framework to automatically prove memory safety and functional correctness of heap programs. For this, we introduce a novel statistical machine learning technique that maps observed program states to (possibly disjunctive) separation logic formulas describing the invariant shape of data structures at relevant program locations. We then attempt to verify these predictions using a theorem prover, where counterexamples to a predicted invariant are used as additional input to the shape predictor in a refinement loop. After obtaining valid shape invariants, we use a second learning algorithm to strengthen them with data invariants, again employing a refinement loop using the underlying theorem prover.
We have implemented our techniques in Cricket, an extension of the GRASShopper verification tool. Cricket is able to automatically prove memory safety and correctness of implementations of a variety of classical list-manipulating algorithms such as insertionsort.

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 [20], miTLS [5], seL4 [19], and IronFleet [15]). However, this requires highly skilled practitioners to invest a significant amount of time in annotating programs with appropriate invariants. While there is little hope of automating the overall process, we believe that this annotation work could be largely automated. To achieve this, we follow earlier work and infer likely (partial) invariants from observed, concrete program runs [10–12, 31–35]. For the actual inference step, we use machine learning techniques to map sets of observations to likely invariants.

A key problem in verification of heap-manipulating programs is the inference of formal data structure descriptions in a suitable logic. Separation logic [26, 29] 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 [8], required extensively annotated inputs [24, 16, 28], or focused on the restricted case of singly-linked lists without data [13, 4, 22, 7, 9, 14, 27].

At its core, finding a program invariant (or widening the observed states "sufficiently") is a search problem. Given a set of program states that are known to occur, we search for a general "concept" (in the form of a formula from an abstract domain) that overapproximates *all* occurring program states. This is similar to many of the problems considered in machine learning, and recent results have shown that program analysis questions can be treated with machine learning techniques [30–33, 11, 25, 12, 35]. With the exception of [30, 25], these efforts have focused on numerical program invariants, and did not consider heap properties, which do not clearly correspond to classical machine learning problems.

In the following, we show how to treat the prediction of formulas similarly to problems such as predicting natural language or program source code (Sect. 3). Concretely, we define a simple grammar generating our abstract domain of separation logic formulas with (possibly nested) inductive predicates (which do not reason about data). 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.

In Sect. 4, we show how this technique can be used in a refinement loop with an off-the-shelf program verifier (in our case, GRASShopper [28]) to automatically prove memory safety of programs. We then show how to extend this with a second refinement loop integrating numerical invariants into the predicted shape invariants in Sect. 5, adapting a technique originally developed for immutable data structures in functional programming [35]. Finally, we combine these techniques in our tool Cricket. In Sect. 6 we experimentally evaluate it on standard list-manipulating algorithms and show that our approach can fully automatically verify programs that are beyond the capabilities of other state-of-the-art tools.

Limitations. As we rely entirely on the underlying program verifier to check correctness of our invariant predictions, we share all of its limitations. Consequently, our technique "inherits" the input language of the program verifier, and thus our implementation operates on the "simple programming language" (SPL) supported by GRASShopper. Furthermore, our performance depends on that of the underlying verifier, and in fact, time spent in GRASShopper dominates our implementation's runtime. As our technique relies on observing a wide sample of occurring program states, it is sensitive to the choice of initial samples (randomly sampled, taken from a test suite, or provided by a human) used in the sample collection phase. Finally, our two-step approach (first shape, then arithmetic invariants) is not able to reason about programs whose memory safety depends on arithmetic arguments, e.g., examples in which memory safety depends on two lists having the same length.

Related Work. Memory safety proofs have long been a focus of research (see above), but we are only aware of two other approaches that can reason about

```
procedure insert(lst: Node, elt: Node) returns (res: Node)
  requires slseg(lst, null) &*& elt.next |-> null
  ensures lseg(res, null) slseg(res, null) {
  if (lst == null || lst.data > elt.data) {
    elt.next := lst;
    return elt;
  } else {
    var cur := lst;
    while (cur.next != null && cur.next.data <= elt.data)
      invariant cur!=null &*& elt!=null &*& lseg(lst,cur) &*& lseg(cur,null) &*& lseg(elt, null)
      invariant cur!=null &*& elt!=null &*& lslseg(lst, cur, cur.data) &*& slseg(cur, null) &*&
                lseg(elt, null) &*& cur.data <= elt.data
    { cur := cur.next; }
    elt.next := cur.next;
    cur.next := elt;
    return lst; } }
```

Fig. 1: Procedure inserting element into a sorted list. Inferred shape (resp. shape-data) loop invariants and postconditions are shown in a box (resp. dashed box).

memory safety and heap data properties fully automatically. One is property-directed shape analysis [17], which uses predicate abstraction over a user-provided set of predicates describing shapes (such as (sorted) list segments, points-to relations, . . .) with a variation of the IC3 property-directed reachability algorithm [6] to prove memory safety and data properties. The other approach we are aware of is [1], which extends the Impact [23] reachability algorithm to reason about the heap using a technique for interpolation between separation logic formulas. Unlike our technique, both approaches can also handle examples where reasoning about data is required to prove memory safety. However, while we could not obtain (working) copies of the implementations, the reported experiments indicate that they are slower by up to an order of magnitude, and unlike our tool cannot prove correctness of more advanced examples such as sorting algorithms.

Closest to this work is probably [25] which infers likely heap invariants from program *traces* (i.e., it infers shapes from usage patterns) using machine learning techniques. However, it is not able to reason about data in these data structures.

2 Example

We demonstrate the key points of our method on a simple example. The program in Fig. 1 is taken from the GRASShopper test suite, and our goal is to *automatically* infer a loop invariant and postcondition. Note that our tool can also infer pre- and postconditions of subroutines.

The program operates on a singly-linked list type in which list elements are `Nodes` with a `next` and a `data` field, and inserts a given element into the correct position of a sorted list. So if `lst` is `[2, 4, 6, 9]` (a list containing the elements 2, 4, 6, and 9) and `elt` is the singleton list `[7]`, `insert` modifies `lst` such that it is `[2, 4, 6, 7, 9]`. In the precondition, `slseg(lst, null)` means that there is a (possibly empty, but acyclic) sorted list segment starting in `lst` and ending in `null`. The separating conjunction from separation logic is written as `&*&` and `elt.next |-> null` indicates that (i) `elt.next` is null and that (ii) `elt` is non-null.[6] Without an explicit loop invariant, `GRASShopper` cannot prove memory safety, nor can it prove that the program returns a sorted list segment.

Shape Invariants. First, we prove memory safety of the program using only *shape* properties. We sample program states satisfying the precondition `slseg(lst, null) &*& elt.next |-> null`, obtaining states such as the one at the top of Fig. 2, where a node is a memory location, data is displayed in it (which we ignore at this stage), and `next` pointers are shown as edges. We then execute the program on these states and record the state whenever execution reaches the head of the `while` loop to obtain a set of reachable program states S^+ (some displayed in Fig. 2 below the line).

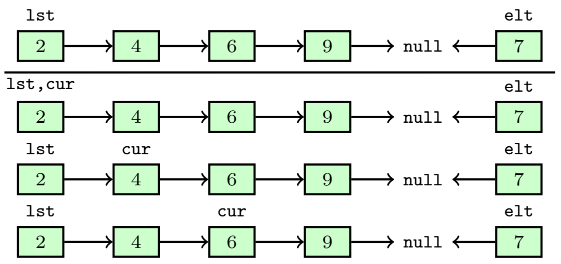

Fig. 2: Initial and execution states for Fig. 1

The obtained set S^+ is then passed to our shape invariant predictor Platypus. Platypus predicts a separation logic formula by following its syntactic structure, i.e., it predicts a syntax tree top-down (see Sect. 3 for details). For this, we use a simple grammar describing the fragment of separation logic that we support. Prediction then reduces to predicting a single production step in this grammar at a time. Each such step has a simple intuitive meaning, such as predicting what data structure is instantiated in a part of a heap graph, or at which variable such a data structure starts. These prediction steps are implemented using standard tools from statistical machine learning, using features extracted from the observed program states and the partial formula predicted so far. For our example, we predict that `lseg(lst, cur) &*& lseg(cur, null) &*& lseg(elt, null)` adequately describes the observed states. We then do a simple nullness analysis to additionally obtain `cur != null &*& elt != null` and thus generate the loop invariant shown in Fig. 1, which we insert into the program and ask GRASShopper to verify. Similarly, we infer the postcondition `lseg(res, null)`.

This proof succeeds, and thus we have proven memory safety of the program. If the proof had failed, a counterexample state would have been returned, which would be used as additional input for Platypus to produce a more precise invariant.

Shape/Data Invariants. In a second step, we aim to strengthen the obtained memory safety proof to prove functional correctness of our program. We use the same samples that we used to predict the shape invariants above. Using the structure of the obtained shape invariant, we can split the observed data into different groups corresponding to the observed (lseg) predicates, and infer data invariants on these. The data for different components (observed lists and data fields of objects referenced directly by stack variables) is shown in Fig. 3. Following [35], we use the observed list data to learn quantified invariants.

[6] (ii) follows implicitly from the statement about one of `elt`'s fields.

For this, we make the *footprint* of a separation logic predicate explicit using a function FP, e.g., FP(`lseg(lst, cur)`) contains the `Node` objects in the list between `lst` and `cur`, excluding `cur`. We are interested in the data properties satisfied by each node in the footprint and use a *containment* predicate $u \in$ FP(`lseg(lst, cur)`) to check whether a node u is in the footprint. We are also interested in the data properties between two nodes in the footprint. The *ordering* predicate FP(`lseg(lst, cur)`) $: u \rightarrow^{+} v$ is true iff v can be reached from u by repeated dereference in the footprint.[7]

State element	Iter. 1	Iter. 2	Iter. 3
`lseg(lst,cur)`	`[]`	`[2]`	`[2,4]`
`lseg(cur,nil)`	`[2,4,6,9]`	`[4,6,9]`	`[6,9]`
`lseg(elt,null)`	`[7]`	`[7]`	`[7]`
`lst.data`	2	2	2
`cur.data`	2	4	6
`elt.data`	7	7	7

Fig. 3: Data observed when running Fig. 1 on `[2,4,6,9]`.

We then infer universally quantified arithmetic invariants with quantifiers ranging over the footprints of the considered list segments using our observations. In our case, this yields $\forall u.\ u \in$ FP(`lseg(lst, cur)`) $\Rightarrow u.$`data` $\leq$ `cur.data`, reflecting that every element of the list from `lst` to `cur` only contains elements smaller than `cur.data`. Similarly, we also find $\forall u, v.$ FP(`lseg(lst, cur)`) $: u \rightarrow^{+} v \Rightarrow u.$`data` $\leq v.$`data`, $\forall u, v.$ FP(`lseg(cur, null)`) $: u \rightarrow^{+} v \Rightarrow u.$`data` $\leq v.$`data`, reflecting sortedness in these footprints, and `cur.data` $\leq$ `elt.data`. In Fig. 1 we denote this by using `slseg` for sorted list segments and `lslseg(x, y, v)` for a sorted list segment from `x` to `y` whose values are bounded by `v`. After strengthening the loop invariant obtained before with this information, GRASShopper can prove that the postcondition of `insert` shown in Fig. 1 always holds.

3 Predicting Shape Invariants from Heaps

In this section, we first discuss a general technique to predict derivations of an arbitrary grammar G from a set of objects $\hat{\mathcal{H}}$, given functions that compute features from these objects. We then show how to apply this to our setting, using separation logic as the grammar and heap graphs as input objects, and discuss the features used. Finally, we discuss some extensions of this general principle that were useful for our implementation Platypus.

3.1 General Parse Tree Prediction

Let G be a context-free grammar, $\mathcal{S}$ the set of all (both terminal and nonterminal) symbols of G, and $\mathcal{N}$ just the nonterminal symbols. We assume that every sentence generated by G has a unique parse tree, and try to learn how to predict such parse trees, following a similar technique that predicts source code from natural

[7] The predicates used here are a small generalization of those used in [17].

language [2]. The benefit of this view is that we can then define our machine learning model over these trees, which are fully observed at training time.

We represent parse trees as tuples $\mathcal{T} = (\mathcal{A}, g(\cdot), ch(\cdot))$ 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 node from the grammar, and $ch : \mathcal{A} \to \mathcal{A}^*$ maps a node to its direct children in the syntax tree. A partial parse tree $\mathcal{T}_{<a}$ is a parse tree $\mathcal{T}$ restricted to nodes $\{1, \ldots, a-1\}$, where the ordering on nodes comes from the order in which they are predicted (see below).

We formulate the algorithm as a sequential prediction task where we predict the parse tree node-by-node, in a depth-first left-to-right order. At each step, we predict the children of the current node conditional upon all of the predictions that have been made so far. In terms of the grammar G, this means we have to pick one from the set of valid production rules to expand the current nonterminal $N \in \mathcal{N}$. To make these predictions, we extract a feature vector $\boldsymbol{\phi}^N \in \mathbb{R}^{D_N}$ that depends on the considered nonterminal N, the input object and the partial tree generated so far.

Depending on the kind of production rules we have to expand N, we can then view this either as a multiclass classification task (if there is a fixed, small number of productions) or as a ranking task (if the production requires us to pick from a set of terminals that are not fixed at training time, for example, program variables). A standard machine learning algorithm can then be used on the computed $\boldsymbol{\phi}^N$ to make a prediction. In practice, we use logistic regression for all classification tasks and a simple two-layer neural network for the ranking tasks. The pseudocode for this procedure PlatypusCore is given in Alg. 1, which is initially called with a parse tree containing only the grammar's start symbol. The probability of a full parse tree $\mathcal{T}$ can be expressed as the product of probabilities of the individual production choices, i.e., $p(\mathcal{T} \mid \hat{\mathcal{H}}) = \prod_{\{a \in \mathcal{A} \mid g(a) \in \mathcal{N}\}} p(ch(a) \mid \hat{\mathcal{H}}, \mathcal{T}_{<a})$.

Algorithm 1 Pseudocode for PlatypusCore

```
Input: Grammar $G$, input objects $\hat{\mathcal{H}}$, (partial) parse 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: $\boldsymbol{\phi} \leftarrow \boldsymbol{\phi}^N(\hat{\mathcal{H}}, \mathcal{T})$                          {compute features}
3: $P \leftarrow$ most likely production $N \to \mathcal{S}^+$ from $G$ considering $\boldsymbol{\phi}$
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:     $\mathcal{T} \leftarrow$ PlatypusCore($G, \hat{\mathcal{H}}, \mathcal{T}, a'$)
7: return $\mathcal{T}$
```

To train the overall system, we process input sets $\hat{\mathcal{H}}$ with their corresponding ground truth parse tree $\mathcal{T}$ to obtain pairs $(\boldsymbol{\phi}^N(\hat{\mathcal{H}}, \mathcal{T}), P)$ of feature vectors and chosen production rules P for each nonterminal symbol N. For this, we follow the parse tree structure analogously to Alg. 1, at each step extracting the feature vector and the chosen ground truth production rule. We then use the extracted data as training data for the individual classifiers and rankers.

3.2 Predicting Separation Logic Formulas

Inputs As inputs we consider 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 $\mathcal{V}$.

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

Outputs We consider a fragment of separation logic [26, 29]. 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 [4], but no $-\!\!*$. As pure formulas, we only allow conjunctions of (dis)equalities between identifiers, 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. Given a set of fresh variables $\mathcal{V}$, the following grammar describes our formulas:

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

Semantics are defined as usual for separation logic, i.e., $\hat{H} \models \sigma_1 * \sigma_2$ for some $\hat{H} = (V, E, \mathcal{L}) \in \hat{\mathcal{HG}}$ if $\hat{H}$ can be partitioned into two subgraphs $\hat{H}_1, \hat{H}_2$ such that $\hat{H}_1$ (resp. $\hat{H}_2$) 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 for all $1 \leq i \leq n$, there is some edge (v, i, e_i). For detailed semantics, we refer to [26, 29]. 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, ls and tree are defined as follows:

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

We often use $\top \equiv \lambda v_1, v_2, v_3, v_4 \to \mathit{true} : \mathsf{emp}$ to denote "no further nesting". So a list of binary trees from x to y is described by $\mathsf{ls}(x, y, \lambda f_1, f_2, e_1, e_2 \to \mathsf{tree}(e_1, \top))$.

[8] In this part, we discard non-pointer values.

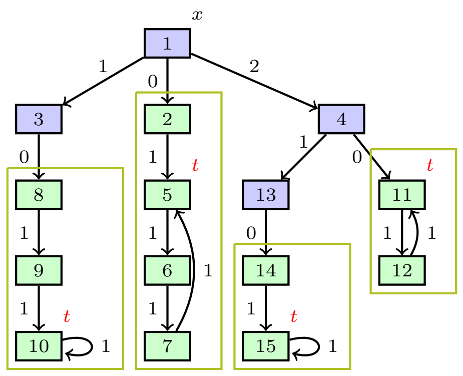

Fig. 4: Binary tree of panhandle lists described by the formula $\mathsf{tree}(x, \lambda i_1, i_2, i_3, i_4 \to \exists t.\mathsf{ls}(i_2, t, \top) * \mathsf{ls}(t, t, \top))$

Example 2. The formula $\varphi(i_1, i_2, i_3, i_4) \equiv \exists t.\mathsf{ls}(i_2, t, \top) * \mathsf{ls}(t, t, \top)$ describes a so-called "pan-handle list" (or "lasso") starting in i_2, where an acyclic list segment leads to a cyclic list. Here, t is the existentially quantified node at which "handle" and "pan" are joined. Hence, the formula $\psi(x) \equiv \mathsf{tree}(x, \varphi)$ describes a binary tree whose nodes in turn contain (separate) panhandle lists. An example of a heap satisfying the formula ψ is shown in Fig. 4. Blue nodes are elements of the tree data structure, having three outgoing edges labeled $0, 1, 2$. Each of the green boxes in Fig. 4 corresponds to a *subheap* that is described by the subformula φ. In each of these subheaps, one node (where the "handle" of a panhandle list meets the "pan") is labeled with t – note that t is not a program variable, but introduced through the existential quantifier in φ.

We found that our procedure $\mathsf{PlatypusCore}$ was often imprecise when generating the pure subformula Π and $\mapsto$ atoms. For this reason, we generate Π deterministically with a nullness analysis (see Sect. 3.3), and completely omit $\mapsto$. Our grammar thus simplifies as follows (where Π is now a terminal symbol).

$$\begin{aligned} \varphi &:= \exists \mathcal{V}.\varphi \mid \Pi : \Sigma & \Sigma &:= \mathsf{emp} \mid \sigma * \Sigma \\ E &:= 0 \mid \mathcal{V} \mid \mathcal{P}\mathcal{V} & \sigma &:= \mathsf{ls}(E, E, \lambda\mathcal{V}, \mathcal{V}, \mathcal{V}, \mathcal{V} \to \varphi) \\ & & &\mid \mathsf{tree}(E, \lambda\mathcal{V}, \mathcal{V}, \mathcal{V}, \mathcal{V} \to \varphi) \end{aligned}$$

Predicting Flat Formulas We first consider the case where the input is a single graph $\hat{H}$ with nodes V, and we predict formulas from a restricted separation logic grammar without nesting. These two restrictions are connected, as we treat nested data structures by considering each "subheap" as an additional input (see below). We will discuss the construction of $\boldsymbol{\phi}^N$ for this simple case first.

Basic Definitions For any a, we define $\mathcal{I}(\mathcal{T}_{<a})$ as the set of identifiers that are in scope at point a in the partial parse tree. Furthermore, let $\mathcal{D}(\mathcal{T}_{<a}) \subseteq \mathcal{I}(\mathcal{T}_{<a})$ be the set "defined" identifiers that occur in the first argument of ls and tree

predicates (i.e., whose corresponding data structure has already been predicted). This will help us to compute features useful for predicting the expansion of an E nonterminal into an identifier.

Some features were found to be useful for all nonterminals, e.g., the depth of the node in the parse tree and the number of occurrences of nonterminal nodes of the same type in the partial parse tree generated so far. Finally, we use boolean features of the form "the i-th parent node is a nonterminal of type n", and analogous features for the last few nodes generated in the tree.

An important class of features is based on the notion of of *n-grams* of paths in the heap graph. For this, each node v is identified with a 1-gram: A pair $(\mathit{indeg}(v), \mathit{outdeg}(v)) \in \mathbb{N}^2$ corresponding to its indegree and outdegree. A 2-gram is then a pair of the 1-grams for two nodes connected by an edge in $\hat{H}$. Based on this, we have developed a measure of *depth*. For a path $v_1 \dots v_t$ in the heap graph, we define its *1-gram weight* as the number of times the 1-gram information changes, i.e., $|\{i \in \{1 \dots t-1\} \mid \mathit{indeg}(v_i) \neq \mathit{indeg}(v_{i+1}) \vee \mathit{outdeg}(v_i) \neq \mathit{outdeg}(v_{i+1})\}|$. Then, $\mathit{depth}(v)$ is the minimal weight of paths leading from a node labeled by a variable to v. In our method, we extend 1-grams by this depth notion, i.e., represent each node by a $(\mathit{indeg}(v), \mathit{outdeg}(v), \mathit{depth}(v))$ triple. Intuitively, this information helps to discover the level of data structure nesting. To compute features for a heap graph, we count the number of occurrences of an n-gram in that graph, only considering the n-grams observed at training time.

As an example, consider Fig. 4 again. There, node 1 has 1-gram depth 0, nodes 3, 4, and 13 have 1-gram depth 1 (note that we haven't drawn the edges to 0 for some "tree" nodes), and nodes 8, 9, 2, 11 and 14 have 1-gram depth 2. Thus for the whole graph, we compute the 1-gram features $1gram_{(0,3,0)} = 1$ (for node 1), $1gram_{(1,3,1)} = 3$ (for nodes 3, 4, and 13) and so on.

Features for Σ, σ Nonterminals Intuitively speaking, the Σ production step is the question of whether the syntax tree generated so far "explains" the observed heap graphs, or if further heaplets are needed. In the σ production step, the right predicate (i.e., either ls or tree) needs to be picked. To this end, we have developed a technique that approximates the footprint of the syntax tree generated so far (i.e., graph nodes that are "explained" by the partial formula),[9] denoted as $V_{<a}$ for some syntax tree node a. We then compute 1- and 2-gram features 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.

Let $\mathcal{I}_a^- = \mathcal{I}(\mathcal{T}_{<a}) \setminus \mathcal{D}(\mathcal{T}_{<a})$ be the identifiers that are in scope, but have not yet appeared in the first position of a predicate. We also compute n-gram features restricted to the nodes $V|_{\mathcal{I}_a^-} \setminus V_{<a}$, where $V|_{\mathcal{I}_a^-}$ denotes those nodes in $\hat{H}$ reachable from $\mathcal{I}_a^-$. Note that we directly choose fresh names for all $\mathcal{V}$ nonterminals bound by the lambda in the σ production rules.

[9] This is a deterministic procedure exploiting our knowledge of the semantics of the supported predicates ls and tree. See Sect. 6.2 for a discussion of how to avoid this procedure.

Features for E Nonterminals Here, our aim is to pick an expression as argument to a predicate. 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 the prediction as a ranking problem over these legal choices.

To handle this, we compute one feature vector $\boldsymbol{\phi}^E_z$ for each $z \in \mathcal{I}(\mathcal{T}_{<a}) \cup \{0\}$. The used features are mostly based on connectivity to related graph nodes. For this, we define the sequence of "enclosing defined identifiers" $e_1, \ldots, e_t \in \mathcal{I}(\mathcal{T}_{<a})$. These are the identifiers appearing in predicates enclosing the currently considered node a, that define the data structures the predicates represent. As an example, consider the partially predicted formula $\mathsf{ls}(x, y, \lambda i_1, i_2, i_3, i_4 \to \mathsf{ls}(v, \mathbf{?}, \ldots))$, where we are interested in predicting the expression at $\mathbf{?}$. Here, we would have $e_1 = v$ and $e_t = e_2 = x$. We then compute boolean features reflecting the reachability of z from $e_1, \ldots, e_t$ (and for the reachability of $e_1, \ldots, e_t$ from z).

To make a prediction, we use a neural network NN (with learnable parameters θ^E) with a single output to compute a score $s_z = \mathrm{NN}(\boldsymbol{\phi}^E_z; \theta^E)$. We then normalize via a softmax over the scores for all possible expressions at this point, i.e., $p(z) = \frac{\exp s_z}{\sum_{z' \in \mathcal{I}(\mathcal{T}_{<a}) \cup \{0\}} \exp s_{z'}}$.

Features for φ Nonterminals The main challenge here is to decide whether to declare new identifiers via existential quantification, so that we can later refer back to them (e.g., for panhandle lists, or join points of separate lists). We found it advantageous to not only predict *that* we need a quantifier, but also by which graph node it should be instantiated.

Practically, we proceed similar to the E case. For each node $v \in \hat{H}$, we compute a feature vector $\boldsymbol{\phi}^\varphi_v$. The used features are based on standard graph properties, such as membership in a strongly connected component, existence of labels for a node, and the 1- and 2-gram features discussed above. We then use a neural network NN (with learnable parameters θ^φ) with a single linear output to compute a score $s_v = \mathrm{NN}(\boldsymbol{\phi}^\varphi_v; \theta^\varphi)$. The output is passed through a logistic sigmoid to give the probability that a new identifier, corresponding to node v in the heap graph, should be instantiated. When choosing a production for φ, we thus make independent predictions for each v and instantiate a fresh identifier i_v for each v that was predicted to be "on".

Predicting Nested Formulas We now consider the general case, in which we have several input heap graphs $\hat{\mathcal{H}}$, 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 several subgraphs of each of the graphs. As an example, consider again the heap graph in Fig. 4, and imagine that we have successfully predicted the outer part of the corresponding formula, i.e., $\mathsf{tree}(x, \lambda i_1, i_2, i_3, i_4.\mathbf{?})$, and are now trying to expand the "hole" marked by $\mathbf{?}$. This subformula needs to describe all the subheaps corresponding to the contents of the green boxes in Fig. 4. To handle this, we now allow modifying the input $\hat{\mathcal{H}}$ after a production step (between line 4 and 5 of $\mathsf{PlatypusCore}$).

So for our example, we would change $\hat{\mathcal{H}}$ to contain 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 directly lift the techniques from Sect. 3.2. The main difference is that we now have to handle a set of heap graphs $\hat{\mathcal{H}}$. 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} of each feature f.

φ nonterminals Here, we cannot easily follow the strategy from above, as the number of nodes to consider may differ between the different heap graphs. For example, in Fig. 4, when declaring t inside the nested structure, t will correspond to one node inside each green box.

In this case, we have a basic form of the structured prediction problem [3]. Suppose there are R heap graphs. For each of the graphs, there is a set of nodes V_r which may require an existential quantifier to be described in our setting (in Fig. 4, these would be the contents of the different green boxes).

Let y_v be a boolean denoting the event that a new identifier is declared corresponding to node v. We train a neural network like in the single-heap case so that the probability of declaring a variable corresponding to node v is predicted to be $p(y_v = 1) = \frac{\exp s_v}{1+\exp s_v}$, where s_v is the score output by the NN. To make predictions, we set the probability of all illegal events to 0 and then make predictions from the distribution over the remaining possibilities. Letting $Z_r = \prod_{v \in V_r} (1 + \exp s_v)$, we can write the probability of not declaring any variables as $\prod_r \prod_{v \in V_r} (1 - p(y_v = 1)) = \prod_r \frac{1}{Z_r}$. The probability of selecting exactly one node v from subheap r is $\frac{\exp s_v}{1+\exp s_v} \prod_{v' \in V_r, v' \neq v} \frac{1}{1+\exp s_v} = \frac{\exp s_v}{Z_r}$. As the choice of node from each subheap is independent given that we are declaring a new identifier, the probability of choosing the set of nodes $\{v_r\}_r$ is the product $\prod_r \frac{\exp s_{v_r}}{Z_r}$. 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. We can then ask what the total unnormalized probability of declaring a variable is. This is the sum of the unnormalized probability of all possible ways to choose one v_r from each subheap r, which can be written as $\prod_r \sum_{v \in V_r} \exp s_v$.

To make predictions, we compute the set of scores $\boldsymbol{s}_r = \{s_v \mid v \in V_r\}$. We first decide whether to declare a new identifier. Following from above, the probability of not declaring a new identifier is $\frac{1}{1+\prod_r \sum_{v \in V_r} \exp s_v}$ while the probability of declaring a new identifier is $\frac{\prod_r \sum_{v \in V_r} \exp s_v}{1+\prod_r \sum_{v \in V_r} \exp s_v}$. To make a prediction under these constraints, we can first compute the probability that a new identifier is declared in each subheap. If we decide not to declare a variable, we instead choose the $\Pi : \Sigma$ production. If we do decide to declare a variable, then we can draw one node from each subheap according to a softmax over the scores; i.e., the probability of choosing node v in subheap r is $\frac{\exp s_v}{\sum_{v' \in V_r} \exp s_{v'}}$. Similar reasoning can be applied to find the most likely configuration of y_v variables to make a prediction.

3.3 Predictions with Platypus

Our formula prediction algorithm Platypus instantiates PlatypusCore as discussed above, but extends it slightly to produce pure subformulas and to generate disjunctive formulas. Furthermore, we slightly adapt the strategy from above to not only greedily select the most likely production rule at each step, but to sample several invariant candidates, returned in order of their respective probability.

Pure Subformulas We use a deterministic procedure to expand the nonterminal Π describing the pure part of our formulas, using a simple aliasing and nullness analysis. 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. Π is then set to the conjunction of all equalities that hold in all inputs graphs.

Handling Disjunctions We also use standard machine learning techniques to predict *disjunctive* separation logic formulas. We found this to be needed even for surprisingly simple examples, as in many cases, the initial or final iteration of a loop requires a (slightly) different shape description from all other steps. In our setting, the problem of deciding how many and what disjuncts we need in the resulting formula can be treated as a clustering problem of heap graphs. Using features similar to those we use for formula predictions, we first partition the input states into clusters using a clustering algorithm, and then predict a disjunctive formula by predicting formulas for each cluster independently.

As features, we use reachability features among program variables. For any pair of variables $u, v \in \mathcal{PV}$ and each heap graph $\hat{H}$, let $r_{\hat{H}}(u, v)$ be 1 if the node labeled by u can reach the node labeled by v in $\hat{H}$ by following zero or more edges and 0 otherwise. Similarly, let $\overline{r}_{\hat{H}}(u, v)$ be 1 if there is a path from the node labeled by u to the one labeled by v that does not pass a node labeled by $w \notin \{u, v\}$. Using these features, we define $\boldsymbol{\phi}^{\hat{H}}$ as the concatenation of the vectors $\langle r_{\hat{H}}(u, v)\rangle_{u,v\in\mathcal{PV}}$ and $\langle \overline{r}_{\hat{H}}(u, v)\rangle_{u,v\in\mathcal{PV}}$ for some fixed order on $\mathcal{PV}$.

For the clustering algorithm, we use standard k-means clustering, using the Euclidean distance between the generated feature vectors as distance between heap graphs. In our implementation, we run the clustering algorithm for $k = 1, 2, 3, 4$ and use the clustering on which our formula predictor reports the highest probability of a prediction. As this was sufficient for our examples, we have not explored more complex clustering algorithms.

Generating Training Data Training the logistic regressors and neural nets from above requires large amounts of labeled training data, i.e., heap graphs labeled with formulas describing their shapes. To obtain this data, we fix a set $\mathcal{PV}$ of program variables and then enumerate derivations of formulas in our grammar, similar in spirit to [18]. For each of the generated formulas, we then enumerate models by systematically expanding inductive predicates until only $\mapsto$ atoms remain. From this form of the formula, we can then easily read off models in the form of heap graphs, by resolving the remaining ambiguous possible equalities between variables.

4 Refining and Verifying Shape Invariants

We describe how to connect our shape predictor from Sect. 3 with the program verifier GRASShopper to create our fully automatic memory safety verifier Locust. For this, we keep a list of *positive* $S^+(\ell)$ and *negative* 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 obtain positive samples corresponding to valid program runs by executing the input program under supervision. Then we use Platypus to obtain a set of initial candidate formulas for each location, and enter a refinement loop. If verification using these candidates fails, GRASShopper returns a counterexample as a program state at some location ℓ, which we then use to extend the sets $S^+(\ell)$ and $S^-(\ell)$ of positive resp. negative samples. If no counterexample is returned, soundness of GRASShopper implies memory safety. It is possible that no correct set of program annotations can be found (either in case of an incorrect program, or due to imprecisions in our prediction procedure). If the same counterexample is reported for the second time (i.e., we stopped making progress), our algorithm reports failure. The pseudocode of this procedure is displayed as Alg. 2.

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:         $\varphi_\ell^1, \dots, \varphi_\ell^k \leftarrow$ obtain $k$ candidates from Platypus($S^+(\ell)$)
 6:         $\varphi_\ell \leftarrow$ first $\varphi_\ell^i$ proven consistent with all $S^+(\ell), S^-(\ell)$      {see Sect. 4.3}
 7:     $P' \leftarrow$ annotate $P$ with inferred $\varphi_\ell$
 8:     if GRASShopper($P'$) returns counterexample $s$ then
 9:         if $s$ is new counterexample then
10:             update $S^+, S^-$ to contain $s$ for correct location           {see Sect. 4.2}
11:         else return FAIL
12:     else return SUCCESS
```

To simplify the remaining procedure, we assume that our prediction procedure Platypus always returns the most precise (i.e., weakest) formula from our abstract domain holding for the given set of input heap graphs. While this is not guaranteed, the machine learning system was trained to produce this behavior, and we observe that our implementation 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

[10] Conceivably, these could be provided by users in a pre-formal language and translated to separation logic using an interactive elaboration procedure.

preconditions, we can use GRASShopper as an oracle. By adding `assert false` to the beginning of the program, any returned counterexample is a model of the precondition. This provides us with a memory state at the beginning of the procedure. To get more samples, and to ensure different sizes of input lists, we add cardinality constraints to the precondition. For example, to force the list starting at `lst` to have length ≥ 3, we add `requires lst.next.next != null`. States at other program locations are then obtained by executing the program from the sample obtained at the initial location.

We found that while this strategy is complete relative to the fragment of separation logic supported by GRASShopper, it is slow even for simple preconditions. Thus, in our implementation, we use a simple heuristic sampling algorithm (similar to our model enumeration procedure to generate Platypus training data) that works on simple preconditions (without arithmetic constraints), and generates 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. In this case the counterexample will be a state that is reachable by the program at the beginning of the loop, but which is not covered by the candidate loop invariant. Thus, this 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 [32, 11], 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 that Platypus returns the most precise formula describing our set of samples, 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 our assumption about Platypus, we conclude that s is a reachable state that the postcondition does not cover, and thus add the counterexample to $S^+(\ell)$.
- Case 4: Invalid heap access inside the loop. Here, the counterexample is a state that is consistent with the candidate loop invariant, but which triggers an invalid heap access such as a `null` dereference. Thus, it is a negative sample and we add it 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. As Platypus cannot provide correctness guarantees, it may return a formula that does not fit the positive samples we provided. Furthermore, we do not use the negative samples at all in Platypus. Thus we check each returned formula φ_ℓ for consistency with the observed samples, i.e., $\forall \hat{H} \in S^+(\ell).\hat{H} \models \varphi_\ell$ and $\forall \hat{H} \in S^+(\ell).\hat{H} \not\models \varphi_\ell$. As in our sampling strategy, we use the underlying program verifier for this.

We first translate each state $\hat{H}$ into a formula $\varphi_{\hat{H}}$ that describes the sample $\hat{H}$ exactly, by introducing variables n_v for each node v and representing each edge (n, f, n') as $n.f \mapsto n'$. Then $\hat{H}$ is a model of φ_ℓ iff all models of $\varphi_{\hat{H}}$ also satisfy φ_ℓ. However, since by construction $\varphi_{\hat{H}}$ has only one model, namely $\hat{H}$ (modulo renaming of locations), this is equivalent to checking if $\varphi_{\hat{H}} \wedge \varphi_\ell$ has a model. This can be checked using a complete program verifier by using $\varphi_{\hat{H}} \wedge \varphi_\ell$ as precondition of a procedure containing only `assert false`.

5 Learning Shape/Data Invariants

Finally, we discuss how to use samples from observed program runs to strengthen predicates in memory-safety proofs with information about the data contained. These shape/data invariants are useful, for example, to prove that a linked list is sorted. For this, we have adapted the DOrder procedure [35] originally developed for immutable data structures in functional programming to our setting, calling our extension DOrderImp. Note that while we only discuss the case of linked lists and binary trees here, the procedure is applicable to all linked data structures. The overall analysis (which we call Beetle, see Alg. 3) proceeds similarly to our procedure Locust, but takes the obtained memory safety proof as additional input. We will give a short overview of DOrder and our extension in the following.

Algorithm 3 Pseudocode for Beetle

```
Input: Program $P$ and entry procedure $p$ with precondition $\varphi_p$, locations $L$ requiring
    program annotations, shape annotations $\varphi_\ell$ for $\ell \in L$
 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:         $\varphi_\ell^{sd} \leftarrow$ DOrderImp($\varphi_\ell, S^+(\ell), S^-(\ell)$)
 6:     $P' \leftarrow$ annotate $P$ with inferred $\varphi_\ell^{sd}$
 7:     if GRASShopper($P'$) returns counterexample $s$ then
 8:         if $s$ is new counterexample then
 9:             update $S^+, S^-$ to contain $s$ for correct location     {see Sect. 4.2}
10:         else return FAIL
11:     else return SUCCESS
```

DOrder uses a *hypothesis domain* of atomic predicates used to express shape properties (e.g., list segments) and data properties (e.g., arithmetic relations). We are primarily interested in the shape properties stating either the containment of a certain heap node in a data structure, or relations establishing ordering between two nodes found within the data structure.

The main difference of DOrderImp relative to DOrder is the set and meaning of considered shape predicates. While DOrder could directly derive these from algebraic type definitions, we extract them from the shape annotations generated by Locust. For a separation logic predicate d, we use $\mathsf{FP}(d)$ to return the footprint of d, i.e., all nodes in the heap that are part of the data structure described by that predicate. A containment predicate $u \in \mathsf{FP}(d)$ holds if and only if the heap node u is in the footprint of d. Furthermore, if d is a list segment predicate, $\mathsf{FP}(d) : u \rightarrow^{+} v$ relates a pair (u, v) to d if u occurs before v in d (transitively). Similar definitions are given if d refers to a tree. For example, the predicate $\mathsf{FP}(d) : u \searrow v$ is satisfied only if v occurs in a subtree of d rooted at u. The semantics of ordering predicates directly inherit the semantics of reachability predicates (cf. `Btwn`) in GRASShopper [28].

When inferring shape-data properties at program location ℓ, we first extract the atomic separation logic predicates (e.g. `lseg(x,y)`) from the given shape annotation φ_ℓ using the function $\mathit{Mem}(\varphi_\ell)$. We then consider the following set of (well-typed) predicates over the footprints of separation logic predicates in φ_ℓ:

$$\Omega_{\text{shape}}(\varphi_\ell) = \{u \in \mathsf{FP}(d), \mathsf{FP}(d) : u \rightarrow^{+} v, \mathsf{FP}(d) : u \searrow v \mid d \in \mathit{Mem}(\varphi_\ell)\}$$

Our data predicates are binary predicates, which are restricted to range over relational data ordering properties. For this, let $\mathit{Vars}(\ell)$ return all stack variables in scope at ℓ and u, v be two fresh variables not used in the program. Given φ_ℓ, the data domain, over some object field fld containing integer data (denoted by Ω_{fld}), is constructed from the following atomic predicates.

$$\begin{aligned}\Omega_{\mathit{fld}}(\ell) = &\{u.\mathit{fld} \leq x, x \leq u.\mathit{fld}, u.\mathit{fld} \leq x.\mathit{fld}, x.\mathit{fld} \leq u.\mathit{fld} \mid x \in \mathit{Vars}(\ell)\} \\ &\cup \{u.\mathit{fld} \leq v.\mathit{fld}, v.\mathit{fld} \leq u.\mathit{fld}\}\end{aligned}$$

where only well-typed predicates are considered. While Ω_{fld} only permits a small number of predicates, our experiments show that it is sufficient to learn useful properties.

For a given program location ℓ, we now look for shape/data invariants of the form $\forall u, v.\ \omega_{\text{shape}} \Rightarrow \psi_{\mathit{fld}}$ where $\omega_{\text{shape}} \in \Omega_{\text{shape}}(\varphi_\ell)$ and ψ_{fld} is an arbitrary boolean combination of predicates from $\Omega_{\mathit{fld}}(\ell)$. To solve this inference problem, we compute $\forall u, v.\ \mathsf{Learn}(S^{+}(\ell), \Omega_{\text{data}}(\ell), \Omega_{\text{shape}}(\varphi_\ell))$ where Learn implements the relational learning algorithm from DOrder [35]. In the algorithm, the free variables u and v range over node objects observed in the footprints of the samples in $S^{+}(\ell)$. The learned formulas are "separators" between positive and negative samples, such that they are true on all positive samples and false on at least some of the negative samples. This algorithm produces exactly the specifications described in Sect. 2 using our hypothesis domain for the program in Fig. 1. For verification, we translate discovered invariants into annotations and ask GRASShopper to verify them. The translation is straightforward because the ordering predicates follow the reachability predicates in GRASShopper as discussed above.

6 Experiments & Conclusion

By combining the procedures Locust and Beetle, we obtain our tool Cricket (see Fig. 5 for an overview of all components). Our implementation Cricket is an

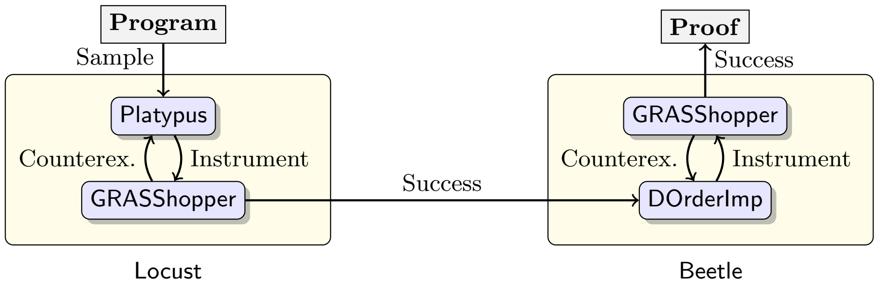

Fig. 5: Overview of our tool Cricket.

extension of the GRASShopper tool [28], with the exception of Platypus, which we have implemented as a stand-alone tool in F#.

6.1 Experimental Evaluation

We have evaluated our tool on the example programs distributed with GRASShopper that operate on lists with integer data. These examples include standard algorithms such as traversal, filtering, and concatenation of sorted and bounded lists, as well more complex algorithms such as quicksort, mergesort and insertionsort. We prove the "natural" program properties, i.e., that modification of a sorted structure again yields a sorted structure, and that sorting algorithms turn an unsorted list into a sorted list. The full set of results is displayed in Tab. 1, where a ✓ indicates that Locust (resp. Beetle) were able to prove memory safety (resp. the given postcondition, starting from the result of Locust), and ✗ that they failed (either due to timeout or by noticing that no further progress could be made). The provided times are cumulative, e.g., the value shown for Locust is the total time used by Platypus, GRASShopper and the few lines implementing the refinement loop. The number in brackets gives the number of iterations spent in the refinement loop. The experiments were performed on an Intel Xeon E5-1620 machine with 16GB of RAM running Windows 10 with a timeout of 300s. We cannot distribute our tool at this time due to dependencies on proprietary machine learning components, but have provided detailed log files of the experiments and the examples under `https://www.dropbox.com/s/c87frsp360of124/experiment_data.zip` and are working on open-sourcing our implementation.

Analysis We find that we can easily prove memory safety of all considered benchmarks. In most cases, Platypus directly predicts the correct shape annotations from the state samples gathered in the initial execution on random inputs. Overall, Cricket was able to prove functional correctness of almost all programs (including "hard" cases such as insertionsort with its nested loops and many interacting variables in scope) fully automatically.

Most time spent in Locust is spent on the consistency check of annotations and samples, which could be improved using a more specialized tool. The Cricket failures on `double_all` and `pairwise_sum` are due to the fact that the abstract

Example	Platypus	Locust	DOrderImp	Beetle	Cricket
`concat`	2s	✓ 68s (1 it.)	32s	✓ 34s (4 it.)	102s
`copy`	2s	✓ 30s (1 it.)	8s	✓ 18s (2 it.)	52s
`dispose`	1s	✓ 4s (1 it.)	0s	✓ 1s (2 it.)	4s
`double_all`	2s	✓ 29s (1 it.)	–	✗	–
`filter`	2s	✓ 6s (1 it.)	1s	✓ 5s (2 it.)	12s
`insert`	2s	✓ 8s (1 it.)	1s	✓ 3s (1 it.)	11s
`insertion_sort`	5s	✓ 23s (1 it.)	41s	✓ 207s (30 it.)	249s
`merge_sort`	15s	✓ 83s (4 it.)	21s	✓ 159s (41 it.)	273s
`pairwise_sum`	2s	✓ 254s (1 it.)	–	✗	–
`quicksort`	5s	✓ 21s (1 it.)	24s	✓ 41s (11 it.)	67s
`remove`	2s	✓ 7s (1 it.)	19s	✓ 24s (5 it.)	31s
`reverse`	2s	✓ 8s (1 it.)	1s	✓ 4s (1 it.)	12s
`strand_sort`	17s	✓ 85s (5 it.)	–	✗	–
`traverse`	2s	✓ 6s (1 it.)	1s	✓ 1s (1 it.)	7s

Table 1: Experimental results of Cricket on example set.

domain used by DOrderImp is insufficient to represent information such as $2 \cdot x = y$. The failure on `strand_sort` is due to GRASShopper timing out (i.e., needing more than 5 minutes) to check annotations provided by Beetle.

6.2 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 to automatically prove memory safety of heap-manipulating programs. Furthermore, we have combined this with a second technique to strengthen such shape invariants with information about the data contained in the described data structures. All of our contributions have been implemented in our tool Cricket, whose experimental evaluation shows that it is able to automatically prove functional correctness of programs that are beyond other state-of-the-art tools.

We plan to extend this work in three aspects. Firstly, we aim to make use of the features of Platypus that are currently not supported by Locust, namely nesting of data structures and the introduction of existential quantifiers. Secondly, one aspect of Platypus that still requires manual and skilled work is feature extraction, which makes it hard to extend the tool to handle new inductive separation logic predicates. We would like to automate the decision of choosing the features relevant for each production rule, and have already made steps in this direction. We recently introduced Gated Graph Sequence Neural Networks [21] - 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.

References

1. A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. In *ESOP '15*, pages 634–660, 2015.
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. G. H. Bakir, T. Hofmann, B. Schölkopf, A. J. Smola, B. Taskar, and S. V. N. Vishwanathan. *Predicting Structured Data*. MIT Press, 2007.
4. 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.
5. K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *SP '13*, pages 445–459, 2013.
6. A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI '11*, pages 70–87, 2011.
7. J. Brotherston, D. Distefano, and R. L. Petersen. Automated cyclic entailment proofs in separation logic. In *CADE '11*, pages 131–146, 2011.
8. 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.
9. 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.
10. 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.
11. P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV '14*, pages 69–87, 2014.
12. P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *POPL '16*, pages 499–512, 2016.
13. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS '06*, pages 240–260, 2006.
14. 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.
15. 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.
16. 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.
17. S. Itzhaky, N. Bjørner, T. W. Reps, M. Sagiv, and A. V. Thakur. Property-directed shape analysis. In *CAV '14*, pages 35–51, 2014.
18. A. J. Kennedy and D. Vytiniotis. Every bit counts: The binary representation of typed data and programs. *JFP*, 22(4-5):529–573, 2012.
19. 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.
20. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
21. Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. In *ICLR '16*, 2016. To appear.
22. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL '10*, pages 211–222, 2010.

23. K. McMillan. Lazy abstraction with interpolants. In *CAV '06*, pages 123–136, 2006.
24. Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *J. Symb. Comput.*, 45(11):1184–1211, 2010.
25. 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.
26. P. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01*, pages 1–19, 2001.
27. J. A. N. Pérez and A. Rybalchenko. Separation logic modulo theories. In *APLAS '13*, pages 90–106, 2013.
28. R. Piskac, T. Wies, and D. Zufferey. GRASShopper - complete heap verification with mixed specifications. In *TACAS '14*, pages 124–139, 2014.
29. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02*, pages 55–74, 2002.
30. R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV '14*, pages 88–105, 2014.
31. 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.
32. R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS '13*, pages 388–411, 2013.
33. R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV '12*, pages 71–87, 2012.
34. G. Yorsh, T. Ball, and M. Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA '06*, pages 145–156, 2006.
35. H. Zhu, G. Petri, and S. Jagannathan. Automatically learning shape specifications. In *PLDI '16*, 2016. To appear.