Learning to Decipher the Heap for Program Verification

Marc Brockschmidt[1], **Yuxin Chen**[2], **Byron Cook**[3], **Pushmeet Kohli**[1], **Daniel Tarlow**[1]
1 Microsoft Research 2 ETH Zurich 3 University College London

Abstract

An open problem in program verification is to verify properties of computer programs that manipulate memory on the heap. A key challenge is to find formal descriptions of the data structures that are instantiated, which are used as input to a proof procedure that verifies the program. Standard approaches to the problem are to severely restrict the space of data structures that can be recognized (e.g., just lists), or to use search guided by hand-specified heuristics. In this work, we explore a machine learning-based approach, where we execute the program and then learn to map the state of heap memory (represented as a labelled directed graph) to a logical description of the instantiated data structures. We formulate the learning problem as one of mapping from graphs to an element of a grammar over data structure descriptions. We report preliminary empirical results showing this to be a promising new direction.

1. Problem Description

Consider the following program fragment:

```
int maxLen(ToL* t) {
    if (t == NULL) return 0;
    int len = length(t->value);
    int rLen = maxLen(t->right);
    int lLen = maxLen(t->left);
    int cLen = rLen > lLen ? rLen : lLen;
    return (len > cLen ? len : cLen);
}
```

where `length` computes the length of a list. Suppose that we would like to prove that all pointer dereferences in this program are valid and will not cause the program to crash. How can we characterize the program's heap at the beginning of `maxLen` so as to guarantee this property? One answer is "a binary tree of lists", i.e., a nested data structure whose top level is a binary tree, and each tree node points to a (linked) list. In *separation logic*, a logic used to describe heap structures, this can be expressed as follows:

$$
\begin{aligned}
\mathit{listtree}(x) \equiv\ & x = \texttt{NULL} \\
& \vee \exists v, l, r. \mathit{list}(v) * \mathit{listtree}(l) * \mathit{listtree}(r) \\
& \quad * x \mapsto \{\texttt{val} : v, \texttt{left} : l, \texttt{right} : r\} \\
\mathit{list}(x) \equiv\ & x = \texttt{NULL} \\
& \vee \exists v, n. \mathit{list}(n) \\
& \quad * x \mapsto \{\texttt{val} : v, \texttt{next} : n\}
\end{aligned}
$$

Here, $x \mapsto \{\texttt{val} : v, \texttt{next} : n\}$ means that x points to a memory region that contains a structure with `val` and `next` fields whose values are in turn v resp. n. The $*$ connective is a conjunction as $\wedge$ in Boolean logic, but additionally requires that its operators refer to "separate" parts of the heap. Thus, $\mathit{listtree}(x)$ implies that x is either `NULL`, or that it points to three values v, l, r on the heap, where the "value" v has to satisfy the list predicate and l and r are in turn again described by $\mathit{listtree}$. We can then prove that under the assumption $\mathit{listtree}(x)$, no program run will fail due to dereferencing an unallocated memory address (this property is called *memory safety*) using a Hoare-style verification scheme (Hoare, 1969),

The hardest part of this process is coming up with the description of data structures, and this is where we propose to use machine learning. Given a candidate description, tools from static program verification (Piskac et al., 2014) can determine whether the description is accurate and whether the description allows one to prove that the program satisfies the desired properties (e.g., memory safety). Thus, from the machine learning perspective, we can focus on generating a small number of candidate descriptions, and if any one is correct, then we have succeeded on the example.

Given a new program at test time, we will run it a small number of times, extract the state of memory at relevant program locations (e.g., at the beginning of method calls), and then predict a separation logic formula. In reality we will map from several memory states from the same program location to a separation logic formula, but in this paper for simplicity we treat the problem as mapping from a single memory state to a separation logic formula. This paper describes our initial approach to this problem.

Proceedings of the Constructive Machine Learning workshop @ ICML 2015.

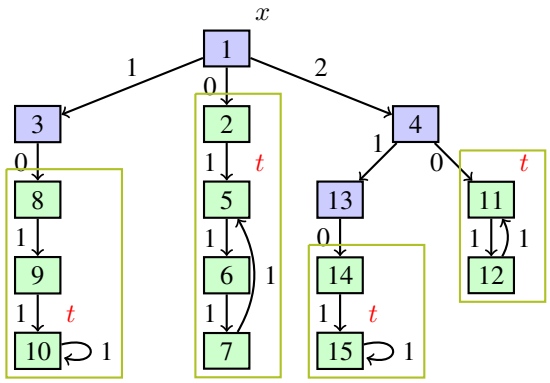

Figure 1. Binary tree of panhandle lists described by the formula $\psi = \mathsf{tree}(x, \lambda i_1, i_2, i_3, i_4 \to \exists t.\mathsf{ls}(i_2, t, \lambda i_5, i_6, i_7, i_8 \to \top) * \mathsf{ls}(t, t, \lambda i_9, i_{10}, i_{11}, i_{12} \to \top))$. $\top$ denotes the "true" formula which holds for any heap, often used to express that there is no further description of a nested data structure.

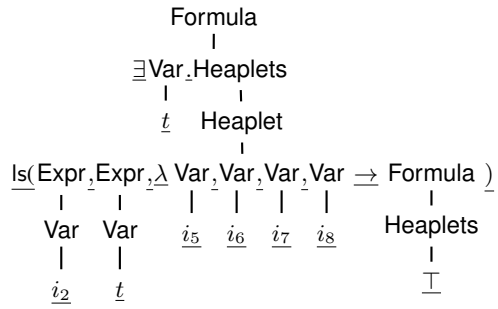

Figure 2. Parse tree of $\exists t.\mathsf{ls}(i_2, t, \lambda i_5, i_6, i_7, i_8 \to \top)$ subformula of ψ from Fig. 1. Terminal symbols underlined. $*\top$ fragments dropped for brevity.

2. Formalization

Input Representation. As inputs we consider directed, possibly cyclic graphs representing the heap of a program. These graphs can be automatically constructed from a program's memory state; an example graph appears in Fig. 1. Intuitively, each graph node v corresponds to an address in memory at which a sequence of pointers $v_0, \ldots, v_k$ is stored. For the purpose of this work, we discard non-pointer values. Edges reflect these pointer values, i.e., v has edges labeled with $0, \ldots, k$ that point to nodes $v_0, \ldots, v_k$, respectively. A subset of nodes are labelled as corresponding to program variables (x but not t; see below).

Output Representation. To describe the shape of the heap, we use separation logic, which is a common tool to reason about heap-manipulating programs in formal verification. Just as a first order formula $\phi(x_1, \ldots, x_n)$ denotes a set of allowed valuations of variables $x_1, \ldots, x_n$, a separation logic formula describes a set of allowed heaps. We refer to O'Hearn et al. (2001); Reynolds (2002) for a precise description of the semantics of separation logic.

As we have seen in §1, inductive data structures such as lists or trees are described using *inductive predicates* such as ls for list segments and tree for trees. We do not define a predicate $listtree$ for trees of lists, but generalize our predicates such that they allow for nested subformulas that restrict the shape of values included in the data structure, following Berdine et al. (2007). For example, $\mathsf{tree}(x, \varphi)$ means there is a binary tree that starts at variable x, and each value of the list is described by the formula φ. For this, the subformula φ takes 4 arguments, which are instantiated by the corresponding values of fields. So for example, in Fig. 1, the subformula of the *tree* predicate is instantiated by values 1, 2, 3 and 4 at the root x, where the first value is the only argument of the *tree* predicate, and the next three correspond to the values of fields `val`, `left`, `right`. Thus, the formula $\exists t.\mathsf{ls}(2, t, \lambda i_5, i_6, i_7, i_8 \to \top) * \mathsf{ls}(t, t, \lambda i_5, i_6, i_7, i_8 \to \top)$ is derived, describing the "panhandle" shape of the list.

The formula ψ in the caption of Fig. 1 describes a tree of so-called "panhandle lists." In the figure, nodes contain the address they are representing and are labeled with variable names to the side. Blue nodes are elements of the tree data structure, having three outgoing edges labeled $0, 1, 2$. Edges which are not displayed lead to the special `NULL` value. Green nodes are part of the nested list data structures. Each of the green boxes corresponds to a *subheap*, which is the inner part of a nested data structure. Note that t is not a program variable, but a name that is introduced through existential quantification. We have found it useful to explicitly label the heap graph node(s) that correspond to existentially quantified variables (the t's in Fig. 1), so we may assume that these correspondences are also included in the output representation (though they are not needed to specify the separation logic formula or by the downstream verifier).

3. Approach

Our aim is to automatically predict a separation logic formula from a given heap H, i.e., given the graph in Fig. 1, obtain ψ from the caption. We consider separation logic formulae described by the following grammar:

$$
\begin{aligned}
\mathsf{Formula} &\to \mathsf{Heaplets} \mid \exists \mathsf{Var}.\mathsf{Heaplets} \mid \exists \mathsf{Var}, \mathsf{Var}.\mathsf{Heaplets} \mid \ldots \\
\mathsf{Heaplets} &\to \top \mid \mathsf{Heaplet} * \mathsf{Heaplets} \\
\mathsf{Heaplet} &\to \mathsf{ls}(\mathsf{Expr}, \mathsf{Expr}, \lambda \mathsf{Var}, \mathsf{Var}, \mathsf{Var}, \mathsf{Var} \to \mathsf{Formula}) \\
&\quad \mid \mathsf{tree}(\mathsf{Expr}, \lambda \mathsf{Var}, \mathsf{Var}, \mathsf{Var}, \mathsf{Var} \to \mathsf{Formula}) \\
\mathsf{Expr} &\to \texttt{NULL} \mid \mathsf{Var}
\end{aligned}
$$

We denote the symbols occurring in the grammar with $\mathcal{S}$, and use $\mathcal{N} \subset \mathcal{S}$ to refer to the subset of nonterminal symbols that have appear on the left side of a production. The nonterminal Formula is the start symbol of the grammar.

Any formula in the fragment of separation logic we consider has a unique parse tree according to this grammar. An example parse tree is displayed in Fig. 2. Based on this, we can assume we observe full parse trees for each formula in the dataset. The benefit is that it allows a directed model over trees, where at training time all quantites are fully observed. This means that maximum likelihood training can be treated as independent classification problems of predicting the child of each parent nonterminal in the tree.

Notationally, 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, $g : \mathcal{A} \to \mathcal{S}$ maps a node to a terminal or nonterminal node in the separation logic grammar, and $ch : \mathcal{A} \to \mathcal{A}^*$ maps a node to a tuple of nodes that are its children in the tree. Nodes are labeled from smallest to largest according to a depth-first, left-to-right traversal of the parse tree.

A *partial tree* $\mathcal{T}_{<a}$ is a parse tree $\mathcal{T}$ restricted to nodes $\{1, \ldots, a-1\}$. We formulate the model over trees as a sequential prediction task where we traverse over nodes, at each step predicting the next node conditional upon everything that has been predicted so far. That is,

$$P(\mathcal{T}) = \prod_{a : g(a) \in \mathcal{N}} p(ch(a) \mid H, \mathcal{T}_{<a}). \tag{1}$$

For each nonterminal type, we extract nonterminal-specific features of both the input heap graph and the partial tree that has been generated so far. There are two key challenges here: first, there are several non-standard prediction problems; for example, when deciding to declare an existential variable within a nested data structure, this amounts to simultaneously choosing a corresponding heap node within each subheap, which we view as a structured prediction problem. These correspondences are used in later feature computations. Another example is when predicting the child of a Var nonterminal, there are a variable number of possible variables to choose from, so we define a model that keeps track of which variables are in scope and only ever choose from amongst the legal options. Second, it is important to craft features that depend jointly on the partial tree that has been generated so far and the input heap graph; we experimented initially with a feature set that was simply a concatenation of features $\phi(H)$ and $\phi(\mathcal{T}_{<a})$, and performance was poor.

As an example, consider observing a heap graph corresponding to a list that originates at variable x and ends at variable y, and suppose we have generated a partial tree that corresponds to the formula $\mathsf{ls}(x, ?, ?)$; that is, we have decided that there is a list that originates at x, but we have not specified where it terminates or what values the list has. Our task is now to predict where the list terminates. If we have a set of candidate variables and want to determine which could be chosen for the first '?', a useful feature is whether the variable is reachable if one starts at the node in the heap graph labelled x and traverses edges in the forward direction (since all nodes in a list should be reachable from the start point). We call this a joint feature because it is jointly a function of decisions we have made so far about the formula (that the list in question starts at x) and the heap graph (the reachability determination). Further details of the features and nonterminal-specific prediction problems are omitted due to space constraints.

4. Data & Experiments

We can generate synthetic (labeled) datasets of arbitrary size and complexity using a simple enumeration strategy. First, fix a set of predicates, which in our case are ls and tree (extensions could consider doubly-linked list segments, multi-trees, ...) together with their inductive definitions. Second, decide on a set of shapes constructed from these basic predicates. In the current version of the dataset, these are simple lists, cyclic lists, panhandle lists and trees. Then, given a set of program variables and a maximal nesting depth of data structures, we obtain separation logic formulas by enumerating all possible combinations of basic shapes instantiated by the given program variables. This yields 127 formulas for one variable and one level of nesting, 33254 for the case of two variables and one nesting level, and 3515 for four variables and no nesting. The dataset size grows exponentially in the size of all parameters. Given a separation logic formula, we can then enumerate corresponding heap graphs of a growing size to obtain a dataset of (heap graph, separation logic) pairs.

For this work, we have produced one dataset of 1757 formulas sampled from the four variable, no nesting set, with 500 heap graphs per formula, which yields 878,500 formula/heap graph combinations. To evaluate, we split the data into training, validation and test sets using a 6:2:2 split on the formulas (i.e., the formulas encountered in the test set were not encountered in the training set). We measure correctness by whether the full heap graph produced at test time is logically equivalent to the ground truth; equivalence is approximated by canonicalizing names and orders of the formulas then comparing for exact equality. We evaluate top K accuracy (how often was the ground truth was in a set of K predictions) for $K = 1$ and 10. We compare our model with a variant that uses a concatenation of syntax tree and heap features (note this subsumes a probabilistic context free grammar conditional upon the heap graph).

Method	Top 1 Acc.	Top 10 Acc.
Concatenation features	0.05%	0.07%
Joint features	91.5%	91.6%

Table 1. Test Results

The difference in performance between the two models is dramatic, which shows that a machine learning approach is viable, but significant care must be taken in the definition of features in order to make it work. An example instance, which was labelled correctly, appears in the Appendix.

5. Related Work & Discussion

In program verification over the past decade, several data-driven program analysis methods have been proposed (e.g., (Ernst et al., 2007; Csallner et al., 2008; Khyzha et al., 2012) and many others). In the data-driven setting, a set of (hopefully representative) program runs is observed, and a program invariant is deterministically derived by generalizing from these observations. This prediction is then verified using a theorem prover, and a proof failure can be viewed as another program run that needs to be generalized.

Recently, implementing this generalization step for the restricted class of "numeric programs" with machine learning techniques has been shown to be viable (Sharma et al., 2012; 2013; Garg et al., 2014; Krishna et al., 2015). In these approaches, a binary classifier is trained to separate states occurring in program executions from those that do not. The learned classifier is then interpreted as the program invariant; e.g., a separating hyperplane is viewed as a linear expression (Sharma et al., 2012), or a decision tree as a disjunctive invariant (Krishna et al., 2015). Our approach is unique in the richness of the invariants that can be output, and in the approach of directly outputting a symbolic description of the occuring states.

In addition to doing larger-scale experimentation, we are now working on integrating our model into a full verification framework, and we expect the resulting tool to succeed on program verification tasks far beyond the reach of existing methods.

References

Berdine, Josh, Calcagno, Cristiano, Cook, Byron, Distefano, Dino, O'Hearn, Peter, Wies, Thomas, and Yang, Hongseok. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pp. 178–192. Springer, 2007.

Csallner, Christoph, Tillmann, Nikolai, and Smaragdakis, Yannis. DySy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 281–290. ACM Press, 2008.

Ernst, Michael D., Perkins, Jeff H., Guo, Philip J., McCamant, Stephen, Pacheco, Carlos, Tschantz, Matthew S., and Xiao, Chen. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3): 35–45, 2007.

Garg, Pranav, Löding, Christof, Madhusudan, P., and Neider, Daniel. ICE: A robust framework for learning invariants. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV '14)*, volume 8559 of *Lecture Notes in Computer Science*, pp. 69–87. Springer, 2014.

Hoare, Charles Antony Richard. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Khyzha, Artem, Parizek, Pavel, and Pasareanu, Corina S. Abstract pathfinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

Krishna, Siddharth, Puhrsch, Christian, and Wies, Thomas. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

O'Hearn, Peter, Reynolds, John C., and Yang, Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*, volume 2142 of *Lecture Notes in Computer Science*, pp. 1–19. Springer, 2001.

Piskac, Ruzica, Wies, Thomas, and Zufferey, Damien. GRASShopper - complete heap verification with mixed specifications. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '14)*, volume 8413 of *Lecture Notes in Computer Science*, pp. 124–139. Springer, 2014.

Reynolds, John C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science (LICS '02)*, pp. 55–74. IEEE Computer Society, 2002.

Sharma, Rahul, Nori, Aditya V., and Aiken, Alex. Interpolants as classifiers. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*, volume 7358 of *Lecture Notes in Computer Science*, pp. 71–97. Springer, 2012.

Sharma, Rahul, Gupta, Saurabh, Hariharan, Bharath, Aiken, Alex, Liang, Percy, and Nori, Aditya V. A data driven approach for algebraic loop invariants. In *Proceedings of the 22nd European Symposium on Programming (ESOP '13)*, volume 7792 of *Lecture Notes in Computer Science*, pp. 574–592. Springer, 2013.

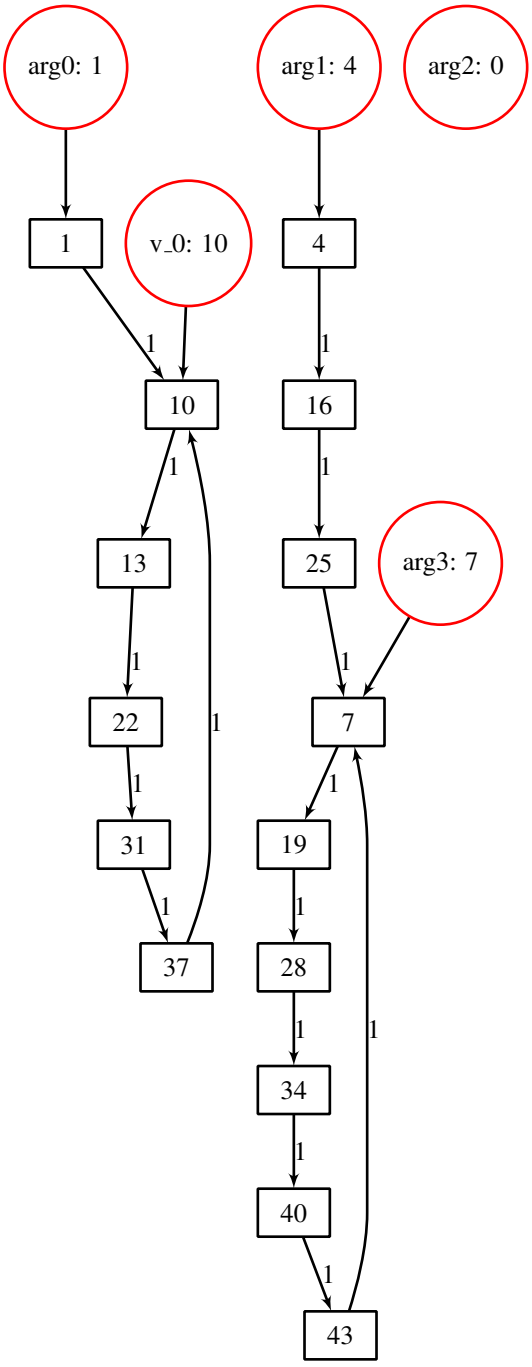

Figure 3. Example input graph from dataset and correctly predicted formula, $\exists v_0.\mathsf{ls}(arg0, v_0, \top)*\mathsf{ls}(arg1, arg3, \top)*\mathsf{ls}(arg3, arg3, \top)*\mathsf{ls}(v_0, v_0, \top)$.