Inferring Universally Quantified Invariants for Dynamic Languages

Will Klieber, Soonho Kong, Edmund Clarke
Carnegie Mellon University

Overview

We are working on a method for automatically inferring loop invariants that are universally quantified over elements of a collection — e.g., properties such as "for any two distinct keys k_1 and k_2 in dictionary d, $d[k_1] \neq d[k_2]$". Our approach is especially suitable for dynamic languages such as Python that support dictionaries (associative arrays), dynamic dispatch, and dynamic typing. We can handle recursive data structures such as AVL binary search trees, for which we can infer invariants about sortedness and non-aliasing. Our technique works by computing an overapproximation of the set of reachable states via a fixed-point procedure. The main contribution of our work is a technique for summarizing heap memory for dynamic languages in a way that automatically preserves invariants.

Representation of Program States

Given a program, we use a finite set of atomic propositions (shown in a box below) to describe states of the program. As usual in symbolic model checking, a state of a program is identified with the set of atomic propositions that are true in the state. Likewise, a propositional formula is identified with the set of states in which it is true.

In order to have a bounded-size representation of programs that use heap-allocated data structure (whose size doesn't have a fixed bound), we must perform some sort of *summarization* of the program state. In our approach, we restrict the names of objects to have no more than two dictionary `lookup`s; an object that would require three or more `lookup`s is summarized.

For summarization of heap memory, we use two *free ghost* variables, denoted "u_1" and "u_2". These represent arbitrary objects, so it is valid to perform Universal Generalization on them. In other words, if we know that a formula ϕ containing u_1 is true in a program state, then we can obtain another true formula by replacing all occurrences of u_1 in ϕ with any object. For example, to represent the fact that a dictionary d is empty, we use the atomic proposition $\texttt{lookup}(d, u_1) = \texttt{undef}$. This implies that no object, not even a summarized object, is a key in d. As another example, to represent that the keys of a dictionary d_1 are mutually exclusive with the keys of another dictionary d_2, we can use:

$$(d_1\,\texttt{[}u_1\texttt{]} = \texttt{undef}) \vee (d_2\,\texttt{[}u_1\texttt{]} = \texttt{undef})$$

The restriction to only two free ghost variables means that we can only infer invariants with at most two quantifiers. Although the restriction is arbitrary, we believe it represents a good trade-off between precision and speed of analysis.

Language

We ultimately aim to analyze and verify real-world programs written in a language such as Python. However, in order to present our ideas more clearly and concisely, we consider a stripped-down language. The `nondet()` function returns an arbitrary object.

```
expr  ::=  var  |  integer-literal  |  string-literal
        |  (var == var)  |  (var < var)  |  !var
stmt  ::=  var = expr
        |  var = nondet()
        |  var = new dict()   /* dict alloc */
        |  var = var[var]     /* dict read */
        |  var[var] = var     /* dict write */
        |  var = (var in var) /* dict test */
        |  stmt ; stmt
        |  assume(var)  |  assume(∀u₁. ϕ)
        |  verify(var)  |  verify(∀u₁. ϕ)
        |  if (var) {stmt} else {stmt}
        |  while (nondet()) {stmt}
        |  func var(var) {stmt}
        |  var(var)
```

Atomic Propositions

```
ap         ::= obj_ref = obj_ref
             | obj_ref < obj_ref
             | AllocOf(obj_ref) = alloc_site
obj_ref    ::= var(var_name, ctrl_pt)
             | lookup(obj_ref_dict, obj_ref_key, ctrl_pt)
             | str_const(string_literal)
             | int_const(integer_literal)
             | undef
             | u₁  |  u₂
             | outer_stack_frame(func, var)
alloc_site ::= ast_node  |  str  |  int
ctrl_pt    ::= old, now, entry
ast_node   ::= Node in the abstract-syntax tree
```

To bound the number of possible atomic propositions, we allow at most two `lookup`s in an *obj_ref*.

We may abbreviate $\texttt{lookup}(d, k, \mathit{ctrl\_pt})$ by $d\,\texttt{[}k\texttt{]}$ when *ctrl_pt* is irrelevent or understood from context.

Representation of State Formulas

The performance of our approach depends greatly on the representation of the state formula. If the representation is inefficient, we can easily run out of time and memory. We would like a representation that (1) is generally small, (2) can be efficiently manipulated by the predicate transformers associated with program statements, (3) allows for us to efficiently check whether a verification condition (e.g., a user assertion or absense of a run-time exception) is valid, and (4) provides a reasonably closed-form final answer for loop invariants.

One clear optimization is to simplify the state formula by leaving out facts implied by the theory of equality, uninterpreted functions, and partial orders. For example, $(a = b) \wedge (b = c) \wedge \Rightarrow (a = c)$ is true in all possible states, but this tautology doesn't necessarily need to explicitly be part of the state formula. On the other hand, if this implied fact is contradicted (e.g., by the guard of a conditional statement or by a user assertion), we would like to detect it and take appropriate action.

When we overwrite a variable or heap cell, or when a heap cell becomes summarized, or when we join two memory states, we may in danger of losing information implied by the theory of equality if we don't explicitly include it in the state formula. For example, consider the below program:

```
a = nondet(); b = nondet(); c = nondet();
if (a == b && b == c) {
    a = (a == c);
    verify(a);
}
```

If we're too lazy in propagating equality information, then we might lose it and be unable to verify that the above program is correct. Our current plan is to propagate such information only when necessary. We had tried eager propagation, thinking that would be acceptable for small programs, but even on small toy programs it quickly exploded into an unmanageably huge size.

Summarization By Allocation Site

Consider the following program that builds a singly-linked list and then traverses it:

```
1.  head = new dict();
2.  head["next"] = 0;
3.  while (nondet()) {
4.     node = new dict();
5.     node["next"] = head;
6.     head = tmp;
7.  }
8.  while (head["next"]) {
9.     head = head["next"];
10. }
```

If presented with this program, our analyzer should be able to verify that the key "`next`" is present in the dictionary whenever it is read. How? We let the free ghost variables u_1 and u_2 range over dictionary references as well as ints and strings. At the start of the program, our initial state formula includes $\texttt{AllocOf}(u_1) \neq \texttt{Line1}$. After Line 1 is executed, this becomes $(\texttt{AllocOf}(u_1) = \texttt{Line1}) \Leftrightarrow (u_1 = \texttt{head})$. Eventually, the original head node may no longer be reachable within two hops of the program variables. Before this happens, we must propagate the information about this node to the free ghost variables, which now may be the node's only nameable aliases. Specifically, for each atomic proposition *ap* mentioning the node, we add $(\texttt{head} = u_1) \Leftrightarrow ap[\texttt{head}/u_1]$ to the state formula, where $ap[\texttt{head}/u_1]$ denotes the result of substituting `head` with u_1. After we quantify away all atomic props that mention `head`, the state formula will imply $(u_1\,\texttt{["next"]} = \texttt{undef}) \Rightarrow (\texttt{AllocOf}(u_1) \neq \texttt{Line1})$.

Function Summaries

Small functions may be profitably inlined. However, this cannot be done with recursive functions; instead, we use function summarization. To create a function summary, when we enter a function, we maintain a copy of the memory state at entry (using *ctrl_pt*=`entry`). We can then relate the output memory state to the input memory state; this relation comprises the function summary. The values of the local variables of the calling function are explicitly included in the memory state. If there are multiple stack frames for a given caller, the stack frames are summarized together.