Not seeing the objects for the references

Diego S. Zaccai
Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210
Email: zaccai.1@osu.edu

Abstract—Current tools that attempt to verify “real-world” object oriented languages, such as KeY for Java, focus solely on the value of references neglecting the value of the objects involved in a program. This leads to the verification of components that are technically correct, since the source code matches the specifications. Yet these components define the stored values in terms of references to objects rather than object values, greatly diminishing their applicability.

I. INTRODUCTION

Most of the tools currently used to verify real world languages use either separation logic [6] or dynamic frames [4] to account for the changes done to variables as a result of a method call. The problems with both these logics is that they assume that a variable is only a reference, neglecting the value of the objects they refer to.

This is problematic in a number of ways. First, a client reasoning about a component such as a List does not think about the reference values of the elements stored inside it and, ideally, we would like the documentation of a component to match the reasoning used to understand it. Second, and more importantly, using references as the value stored in an object in frameworks that restrict access to parts of the heap, prevents implementations of data structures that depend on such object values (such as Sets and Maps).

The difficulty for separation logic of separating the mechanism for passing objects in a modern language from the values of objects in a program has been explained in great detail in [5]. This work led to J-Star [3], and while it is currently abandoned, it made some progress into solving the problems introduced before. However, most tools currently using separation logic to verify real world languages, such as VCC [2], still suffer from many of the shortcomings that Parkinson attempted to solve. One only needs to look at the proposed implementation of a Set in [1] to see these shortcomings.

II. THE KEY TOOL AND JML*

A tool that is under heavy development for verifying a real-world languages is KeY with Java. The KeY verifier uses JML* as its specification language, a version of JML updated to work with dynamic frames as the logic behind updates and assigns clauses. Unfortunately, while separation logic has had some of its most ardent proponents express that there are constructs in real world languages that make verification difficult—if not impossible—such a concern has not been raised about dynamic frames.

Yet, one look at specifications such as List in Listing 1, obtained from [7] clearly demonstrates the limitations of this way of reasoning. In it, we can see that the value of the List is expressed only in terms of its size and the references stored in it, as exemplified by the use of the == operator in the ensures clauses of most methods.

```java
public interface List {
    //@ public model instance \locset footprint;
    //@ public accessible \inv footprint;
    //@ public accessible footprint: footprint;
    //@ public instance invariant 0 <= size();
    //@ public_normal Behaviour
    @ accessible footprint;
    @ ensures \result == size();
    @*/
    public /*@ pure */ int size();
    //@
    public /*@ requires 0 <= index && index < size();
    @ accessible footprint;
    @ ensures \result == get(index);;
    @*/
    public Object get(int index);
    //@
    public /*@ assignable footprint;
    @ assignable footprint;
    @ ensures size() == \old(size()) + 1
    @ && get(size() - 1) == o;
    @*/
    public boolean contains(Object o);
    //@
    public /*@ assignable footprint;
    @ assignable footprint;
    @ ensures \forall int i; 0 <= i && i < size
    @ get(i) == \old(get(i));
    @*/
    public void add(Object o);
}
```

Listing 1. A List interface
Using references as the values stored in a `List` is not a problem if we are only interested in keeping objects stored in certain order. However, as soon as we attempt to do something that depends on the actual values stored in them, such as implementing a `contains` method, we find that the current specifications are unable to support this type of reasoning.

### III. The `contains` Method

The idea behind a `contains` method is that it will return `true` if and only if there is an object inside of the `List` that has the same value as its parameter. Such a method in Java will invariably use the `equals` method to check whether two variables have the same abstract object value. Yet, the implementation of the `contains` method for an `ArrayList` in Listing 2 is verified as correct despite making no such call. A closer look at the specification for the method reveals the root of the problem. The proposed contract for `contains`, while being wishfully named in the same manner as the one found in `java.util`, only checks for reference equality.

That is, instead returning `true` if and only if there is an object inside of the `List` that has the same value as its parameter, the `contains` method presented in KeY returns `true` iff its parameter is an alias to something stored inside of the `List`.

In order to fix the contract so that its meaning matches the intended behavior one would only need to change the expression `get(i) == o` inside the `ensures` clause of the method to `o.equals(get(i))`. However, doing so would run afoul of the method’s `accessible` clause.

Since the method is only able to access the variables defined as part of the `footprint` in our `ArrayList`, it can only use the references in the array, but not any references reachable by them. This, in turn, makes the use of the `equals` method impossible (since the method would need to access the representation of its arguments).

Attempting to fix this by expanding the `footprint` of the `ArrayList` would be impossible, since there is no specification construct that allows you to include the closure of the variables, those that are reachable from each of the array indexes. Further, even if this was possible, using such a construct would be inadvisable: dynamic frames is capable of simplifying reasoning when location sets are disjoint, and an operation that indiscriminately increases the size of a location set is going to greatly reduce the likelihood that two location sets are disjoint.

### IV. Not an Isolated Mistake

This problem of reasoning about reference values instead of object values is not isolated. The code for `MySet` in Listing 3 presents an implementation of a set layered on top of the previously introduced `List` that heavily relies in the incorrectly documented `contains` method.

If the error was not clear enough in the previous example, any attempt at using the `MySet` class should make the

```
public class ArrayList implements List {
    private /*@ nullable */ Object[] array = new Object[5];
    private int size = 0;
    //@ footprint = array, array[...], size;
    //@ array[0] != null;
    @ private invariant 0 <= size <= array.length;
    @ private invariant (forall int i; 0 <= i && i < size; @ array[i] != null);
    @ private invariant typeof(array) == typeof(Object[]);
    @*/
    public /*@ pure */ ArrayList() { }
    public int size() { ... } public Object get(int index) { ... } public void add(Object o) { ... }
    public boolean contains(Object o) { /*@ loop_invariant 0 <= i && i < size && (forall int x; 0 <= x && x < i; array[x] != o); @ assignable \nothing; @ decreases size - i; @*/ for (int i = 0; i < size; i++) { if (array[i] == o) { return true; }
    } return false; }
}
```

Listing 2. The `ArrayList` implementation of the `List` interface in Listing 1

shortcomings of this way of reasoning painfully obvious. A further check into the set\(^2\) implemented in [1] shows that its implementation got around the problem of using object values by forcing the "type" of the stored values in the set to be `ints` and not references to objects.

### V. Conclusion

This short paper introduced a serious limitation of the tools currently used to verify languages with reference semantics. The problem problem is two-fold: having abstraction as an optional part of a specification language means that a generic collection cannot reference the abstract values of its components. Similarly, using references as the value of a variable makes the objects that they point to irrelevant in

---

1. Primitives and references.
2. Not added to this paper due to space constraints.
public class MySet {
    //@ public model locset footprint;
    //@ public accessible @inv: footprint;
    //@ public accessible footprint: footprint;
    private List list;
    //@ private represents footprint = this.*, list.
    footprint;
    //@ private invariant list."inv;
    @ private invariant "disjoint(list.footprint,
    this."
    *=);
    @ private invariant \forall int x, y; 0 <= x
    this."
    *=);
    @ \forall x < list.size() \&\& 0 <= y this."
    *=);
    @ \forall y < list.size() \&\& x != y this."
    *=);
    @ list.get(x) != list.get(y));
    */
    ...
    //@ normal Behaviour
    @ accessible footprint;
    @ ensures \result == contains(o);
    @*/
    public /*@ pure @*/ boolean contains(Object o) {
        return list.contains(o);
    }
    ...
    //@ public normal Behaviour
    @ assignable footprint;
    @ ensures \forall Object x;
    @ contains(x) == \old{contains(x)} ||
    o == x);)
    @ ensures \new elems fresh{footprint};
    @*/
    public void add(Object o) {
        if (!list.contains(o)) {
            list.add(o);
        }
    }
    ...
}

Listing 3. The MySet implementation

the documentation, despite the fact that they are the most important part of a program.

REFERENCES


