The Modularity of Object Propositions

Ligia Nistor
School of Computer Science, Carnegie Mellon University, USA
lnistor@cs.cmu.edu

Abstract

A significant concern in verification research is the ability to reason modularly about programs with state. Recent work has used substructural logics including separation logic, permissions, and Hoare Type Theory to specify each function in terms of its effect on its footprint. The motivation of our work is the need for formal specifications that allow one to hide shared data between two abstractions. In 2014, we proposed object propositions [1] as an automatable extension to abstract predicates. We allow state to be shared between two objects, by providing fractional permissions to access the common data hidden in a predicate, without revealing this sharing in the objects' specifications. Unlike conventional object invariant and ownership-based work, our system allows *ownership transfer* by passing unique permissions (permissions with a fraction of 1) from one reference to another. Unlike separation logic and permission systems, we can modify objects without owning them. This has information-hiding and system-structuring benefits.

Categories and Subject Descriptors D.2.4 [*Software/Program verification*]

Keywords Object propositions, modularity, observer pattern

1. The Theory of Object Propositions

Object propositions combine predicates on objects with aliasing information about the objects, represented by fractional permissions. They are declared by programmers as part of method pre- and post-conditions. An object proposition looks like `obj@k Pred(x)`, where `obj` is a reference to an object, `k` is a fraction representing how much of a permission `obj` has to the abstract predicate `Pred(x)` that `obj` satisfies. A critical part of our work is allowing clients to depend on a property of a shared object. To gain read or write access to the fields of an object, we have to *unpack* it. After a method finishes working with the fields of a shared object , we *pack* back the shared object to that predicate (or to another predicate if we had a fraction of 1 to it).

2. Using Separation Logic for Verification

We explore the modularity of our system by analyzing an example, where our specifications do not expose the shared information. We consider the example of a spreadsheet in which each cell contains a formula that adds two integer inputs. Each cell may point to two other cells. Whenever the user changes a cell, each of the two cells which transitively depend upon it must be updated (an instance of the observer pattern).

Let us assume we have two cells `a1` and `a2` at the top of the spreadsheet, `a1` points to `a4` and `a3`, while `a2` points to `a3` and `a5`. Finally `a3` points to (and uses its additive output as one of the inputs to) `a6`. Let us say we have a method `setInput1(x)` that sets the first input of a cell (one of the two inputs of a cell).

In conventional first-order separation logic, the specification of any method has to describe the entire footprint of the method (all heap locations that we read/write to inside the method). The verification using separation logic is as follows. We define the predicate `SepOK(cell)` to state that the sum of the two inputs of `cell` is equal to the output, and that the predicate `SepOK` is verified by all the cells that directly depend on the output of the current cell. In separation logic, the natural pre- and post-conditions of the method `setInput1()` are both `SepOK(this)`, i.e., the method takes in a cell that is in a consistent state in the spreadsheet and returns a cell with the input changed, but that is still in a consistent state. Before calling `setInput1(x)` on a2, we have to combine `SepOK(a3) * SepOK(a5)` into `SepOK(a2)` (where * is the separating conjunction of separation logic). We observe the following problem: in order to call `setInput1()` on `a2`, we have to take out `SepOK(a3)` and combine it with `SepOK(a5)`, to obtain `SepOK(a2)`. But the specification of the method is not expressive enough to allow it, hence the problem in conventional separation logic. Because the shared cells `a3` and `a6` have to be mentioned in the specification of all methods that modify the cells `a1` and `a2`, the specification in separation logic exposes some of the shared data.

3. Using Object Propositions for Verification

We verified the same example using object propositions and the predicate `OK()`, which uses fractions to ensure that all the cells in the spreadsheet are in a consistent state, where the sum of their inputs is equal to their output. Since we only use a fractional permission for the dependency cells (such as `a3`), it is possible for multiple predicates `OK()` to talk about the same cell without exposing the sharing in the spreadsheet. More specifically, using object propositions we only need to know `a1@k OK()` before calling `a1.setInput1(10)` (same for `a2`) and we can share the cell `a3` by using the fractional permissions of object propositions. Here `k` is an arbitrary fraction in the (0,1] interval meaning that `a1` holds a `k` permission to the predicate `OK()`.

I thank Prof. Jonathan Aldrich for his guidance.

References

[1] Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. Object propositions. In *FM*, 2014.