Using Machine Learning in the Automatic Translation of Object Propositions

Ligia Nistor and Jonathan Aldrich

School of Computer Science, Carnegie Mellon University, USA
{lnistor,aldrich}@cs.cmu.edu

1 Introduction

Object propositions [5] are used for the modular verification of object-oriented code in the presence of aliasing, i.e., the existence of multiple references to the same object. They are associated with object references and declared by programmers as part of method pre- and post-conditions in the process of formally proving the correctness of object-oriented code. Object propositions use abstract predicates [6] to characterize the state of an object. Those predicates are embedded in a logical framework and aliasing information is specified using *fractions* [3].

If in the system there is only one reference to an object, that reference has a fraction of 1 to the object, and thus full modifying control over its fields. If there are multiple references to an object, each reference has a fraction less than 1 to the object and each can modify the object as long as that modification does not break a predefined invariant (expressed as a predicate). In case that modification is not an atomic action (and instead is composed of several steps), the invariant might be broken in the course of the modification, but it must be restored at the end of the modification.

To verify a method using object propositions, the *abstract* predicate in the object proposition for the receiver object is interpreted as a *concrete* formula over the current values of the receiver object's fields.

A critical mechanism in the object propositions methodology is *packing/unpacking* [4]. When the code modifies a field, the specification has to follow suit and unpack the predicate that contains that field (unpacking a predicate gives read/write access to the fields of that predicate). At the end of a method, the fields have been modified and after checking that a predicate holds, we are allowed to pack back that predicate.

2 Example

The code in Figure 1 represents a class *DoubleCount* that has two integer fields *val* and *dbl*. The predicate *OK* states that the value of the field *dbl* is double the value of the field *val*. This predicate is the invariant of the class. If there are two aliases $r1$ and $r2$ to an object o of type *DoubleCount*, $r1$ can assume that the invariant holds even if $r2$ has modified object o by calling the method

increment on it. This is ensured by the pre- and postcondition of the method *increment*. The precondition is $this@k\ OK()$, which states that the caller *this* of the method satisfies the invariant OK. After the execution of the method the same invariant holds, according to the postcondition.

```
class DoubleCount {
      int val;
      int dbl;
  predicate OK() ≡ ∃v : int, d : int . val → v ⊗ dbl → d ⊗ d == v ∗ 2
  void increment()
   ∃k : int . this@k OK() −∘ this@k OK()
  {val = val+1;
   dbl = dbl+2;}
 }
```

Fig. 1. DoubleCount class and OK predicate

3 Translation to Boogie

Below we present the manual translation of the code and specifications of Figure 1 into Boogie [2]. In our Boogie encoding, we created a type *Ref* to represent references of type DoubleCount. We represented the heap by creating maps from objects to their fields: for example we represented the field *val* by *var val: [Ref]int* which maps an object of type DoubleCount to its *val* field of type *int*. We created a new map type to keep count of fractions *type FractionType = [Ref, PredicateTypes] int* and another map type *PackedType* to keep track of which predicates are packed, for a specific object. The first axiom represents the necessary conditions that have to be met for *this* to be packed to the predicate OK, while the second axiom is used when *this* is unpacked from the predicate OK.

```
type Ref;
type PredicateTypes;
type FractionType = [Ref, PredicateTypes] int;
type PackedType = [Ref, PredicateTypes] bool;
const null: Ref;
const unique okP: PredicateTypes;
var val: [Ref]int;
var dbl: [Ref]int;
var packed: PackedType;
var frac: FractionType;

axiom ( forall this: Ref, val: [Ref]int,
                dbl: [Ref]int, packed: PackedType::
      (dbl[this]==val[this]*2) ==> packed[this, okP]);

axiom ( forall this: Ref, val: [Ref]int,
                dbl: [Ref]int, packed: PackedType::
      packed[this, okP] ==> (dbl[this]==val[this]*2) );
```

```
procedure increment(this: Ref)
modifies val, dbl, packed;
requires packed[this, okP] && (frac[this, okP] >0);
ensures  packed[this, okP] && (frac[this, okP] >0);
{ val[this]:= val[this]+1;
  dbl[this]:=dbl[this]+2; }
```

4 Using Machine Learning

Our goal is to automatically translate the code and specifications in Figure 1 to the Boogie code in Section 3. We are currently defining the automatic translation rules. They are going to be the most interesting feature of our compiler which will do the translation. While most of the translation rules are clear, there are some places where machine learning (ML) could be useful. The compiler could generate most of the translation but use ML, more specifically structured prediction[1] [1], to help the compilation in the creation of the axioms.

If our ML algorithm has access to 20 [2] programs that have been manually translated into Boogie and proved correct, the ML algorithm could learn and predict how the axioms will look like. The output generated by the ML algorithm does not need to be precise; it would be very useful if it gives multiple possible axioms from which the programmer can choose the one that looks correct. The programmer could also run all the variants suggested by the structural prediction algorithm and see how each performs. Let's assume we create a new class *TripleCount* that has two fields *val* and *trpl*, and the invariant states that *trpl* should always be triple the value of *val*. The ML algorithm, given enough training translations, among which the translation presented in Section 3, could suggest that the first axiom be:

```
axiom ( forall this: Ref, val: [Ref]int,
                trpl: [Ref]int, packed: PackedType::
      (trpl[this]==val[this]*3) ==> packed[this, okP]);
```

References

1. http://en.wikipedia.org/wiki/Structured_prediction.
2. http://rise4fun.com/Boogie/.
3. John Boyland. Checking interference with fractional permissions. In *Static Analysis Symposium*, pages 55–72, 2003.
4. Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.
5. Ligia Nistor, Jonathan Aldrich, Stephanie Balzer, and Hannes Mehnert. Object propositions. In *19th International Symposium on Formal Methods*, 2014.
6. Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[1] Thank you to Jayant Krishnamurthy for suggesting this technique.

[2] or more, if we involve more people in the manual translation of object propositions specifications into Boogie