Abstract
Verification projects on industrial code have required reasoning about functional programming constructs in Java 8. In our experience so far, verification of functional programming in practice has needed only simple techniques such as inlining. However, in general functional programming will require reasoning about how the specifications of function objects that are inputs to a method combine to produce output function objects. This paper describes our in-progress experience in adapting prior work (Kassios & Müller) to Java 8, JML, and OpenJML.

Keywords JML, OpenJML, ACSL, ACSL++, formal verification, specification, functional programming

ACM Reference Format:
were subject to some simple applications or extensions of JML and OpenJML (cf. [2] for details):
- inlining lambda expressions where they were used as actual arguments to methods;
- extending JML’s model method syntax to be able to represent, and inline, the FP functionality of library methods;
- writing model interfaces that give specifications for function objects that are more specific than general Function<?>

However, none of these techniques are sufficient to reason about the general case in which output function objects are a function of input function objects, a case that did not arise in our verification project. For that situation, OpenJML is adapting and implementing the technique from [5], as described in §4.

3 Related work

FP has not been a focus of other Behavioral Interface Specification Languages. Neither ACtUS for C nor SPARK for Ada nor Spec# for C# has particular syntax for specifying or reasoning about function pointers or function objects. VeriFast [4], building on [6], uses an idiom in which uninterpreted predicates are defined for each general function object or functional interface. These are then made more specific for derived classes. Dafny only permits pure functions to be used as function objects and these are then directly inlined for verification.

4 Reasoning about specifications

This short paper only has space for an outline of the design in the context of an example. Consider a function compose(f,g) { return i -> f.apply(g.apply(i)); } whose arguments and result are all functions of an Integer argument with Integer argument and result, and the result is the composition of f and g, written using Java’s lambda expression syntax. The specifications of the result depend on the specifications of the input arguments. Following [5] this design uses the following syntax, adapted to JML and ACSL:

- \pre(h,i) is a predicate that is true iff h is an Integer->Integer function whose precondition is true when its argument has value i;
- \post(h,r,i) is a predicate that is true iff h is an Integer->Integer function whose postcondition is true for an argument of value i and a result of value r;
- \assigns(h,i) represents the set of possibly modified memory locations by applying function object h to Integer argument i.

Note that \pre, \post and \assigns are keywords, not functions. Their signatures depend on the type of the function object. There also may be an implicit this argument. An alternate syntax might be \pre(h)(i), where \pre(h) denotes a function object; also the this argument might alternatively be made explicit.

As examples, using JML specification syntax, a function dec that decrements its argument by 1 satisfies the following equalities,
- \pre(dec,i) == (i != Integer.MIN_VALUE)
- \post(dec,r,i) == (r == i - 1)

For a function bump that increases its argument
- \pre(bump,i) == (i != Integer.MAX_VALUE)
- \post(bump,r,i) == (r > i)

The composition function itself has the following specification:
- requires (f != null && g != null);
- ensures \pre(<result,i) == (\pre(g,i) && (forall int k;; \post(g,k,i) == \pre(f,k));
- ensures \post(<result,r,i) == (exists int k;; \post(g,k,i) && \post(f,r,k));

The SMT translation of the above design has the following features:
- Lambda expressions, and in general functional literals in the programming language, are represented as distinct constant literals of a user-defined Function sort in SMT-LIB.
- The \pre, \post and \assigns JML keywords become functions in SMT-LIB; the SMT-LIB names are mangled using the function type signature.
- Since \assigns represents a set of memory locations, the logic must necessarily use dynamic frame conditions and not just the static lists of locations that are the familiar part of JML frame conditions.

One proof obligation is that uses of the method, in this case compose, can be verified. With the set of specifications above, a program fragment such as

```
int h = compose(dec,bump).apply(j);
//@ assert h >= j;
```

is readily proved. In cases where an assertion is invalid, we found that some solvers required non-default settings to reliably terminate in reasonable time.

The other proof condition is to verify that the implementation of compose satisfies the specification. That also is readily provable by the SMT-LIB encoding described above.

5 Current status and future work

The syntax described above and the encoding in SMT-LIB have been implemented in OpenJML as an extension to JML. Its expressivity and usefulness for verification are being evaluated with additional use cases and example programs, which will be presented in the workshop presentation. In addition the Java library specifications are being fleshed out for release along with the enhanced OpenJML.
References


[7] VESSEDIA. 2018. The work on C++ specification is part of the VESSEDIA project https://vessedia.eu. The VESSEDIA project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 731453.