Reasoning about Functional Programming in Java and C++

David R. Cok
CEA, LIST, Software Safety and Security Laboratory
Gif-sur-Yvette, France
david.r.cok@gmail.com

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:
David R. Cok. 2018. Reasoning about Functional Programming in Java and C++. In *ACM Digital Library*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.475/123_4

1 Introduction

As of version 8, Java now contains explicit functional programming (FP) features. Although programming with the equivalent of function objects was possible in Java before Java 8, it was not common and not a focus of verification tools and technology. However, the target software of a recent verification project of industrial code by the author and colleagues [2] used Java 8's FP features heavily, requiring that JML and OpenJML be extended to reason about these constructs. As described later, in this software the FP features could be handled using fairly simple techniques, such as inlining. However, more generally, methods can be written in which input function objects are combined to produce output function objects, such that the specifications of the output function depend on the specifications of the (formal) input parameters. To remain within the typed-first-order paradigm of current BISL-based verification tools and SMT backends, we want the logical embedding of FP features to remain first order.

Similarly, the ACSL (ANSI-C Specification Language [1]) and Frama-C [3] team at CEA, in the EC-sponsored VESSEDIA project [7], is designing ACSL++: a specification language for C++. The design of ACSL++ needs to provide syntax, semantics, and logical encodings to enable reasoning about such FP features in C++. Indeed, even C has function pointers, which need similar capability, not yet present in ACSL.

For both these purposes, we propose adapting the solution proposed by Kassios and Müller in 2011 [5] to both Java and C++. This paper describes experimentation with this solution in the context of JML and ACSL++ and our progress in implementing the design in OpenJML. As this design is a work in progress and is a change in JML, discussion and input are welcome.

2 Prior work with FP features in Java

The author and colleagues are completing the verification of a proprietary[1] software library at Amazon [2]. Both Amazon and key customers consider the security and correctness of this cryptographic network communication library to be important and worth the investment in static deductive verification. We are applying (and extending) JML and OpenJML to do so.

The target software uses functional programming heavily. The essential problem is that a method, say `m`, that takes a functional argument, say one of type `Function<?>`, knows very little about the effect of the function represented by that formal argument. The specifications of `Function<?>` are necessarily very general; for example, it may have arbitrary side effects. A related problem is the use of functional combinators that have implicit iteration. An example is the `forEach` method of a Java 8 Collection: it applies a function object to each element of a collection without an explicit loop. The loop is implicit, but the syntactic problem is that there is no place in user code to write the loop specifications that are customarily needed to reason about loops.

Unexpectedly, the target software in this verification project, though a heavy user of FP, used FP in just a few idioms that

FTfJP 2018, July 2018, Amsterdam NL

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

[1] The software is proposed to be open-sourced, but that legal process is not yet completed.

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<?>` and the like types;
- extending JML and OpenJML to combine loop specifications from the caller and callee in case of implicit iteration.

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 ACSL 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` 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:

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

Note that $\backslash pre$, $\backslash post$ and $\backslash assigns$ 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 $\backslash pre(h)(i)$, where $\backslash pre(h)$ 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 $\backslash pre$, $\backslash post$ and $\backslash assigns$ JML keywords become functions in SMT-LIB; the SMT-LIB names are mangled using the function type signature.
- Since $\backslash assigns$ 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

[1] P. Baudin, P.Cuoq, J-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. 2008ff. ACSL: ANSI C Specification Language. http://frama-c.com/download/acsl_1.4.pdf.

[2] David R. Cok and Serdar Tasiran. 2018. Practical Methods for Reasoning about Java 8's Functional Programming Features. Submitted to VSTTE 2018.

[3] Frama-C 2011ff. https://frama-c.com.

[4] Bart Jacobs, Jan Smans, and Frank Piessen. 2017. The VeriFast Program Verifier: A Tutorial. https://zenodo.org/record/1068185#.WvbWhy97HyU.

[5] I. T. Kassios and P. Müller. 2011. *Modular Specification and Verification of Delegation with SMT Solvers.* Technical Report. ETH Zurich.

[6] Matthew Parkinson and Gavin Bierman. 2005. Separation Logic and Abstraction. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05).* ACM, New York, NY, USA, 247–258. https://doi.org/10.1145/1040305.1040326

[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.