Interactive Formal Verification of Object-Oriented Code

Hannes Mehnert

IT University of Copenhagen, 2300 København S, Denmark
hame@itu.dk

Abstract. We present a formal verification of a snapshotable search tree data structure in Java with a fully formalized correctness in Coq and tools which we developed to conduct this case study.

Within the research project Tools and Methods for Scalable Software Verification we are developing practical tools to interactively verify Java programs using a higher-order separation logic. Full functional verification of real-world object-oriented code is challenging due to mutable heap and aliasing. Another challenge for object-oriented code with dynamic dispatch is modular verification: a client of an interface should be able to reason abstractly without knowledge of the implementation. Several existing tools target these problems, but these use different approaches: either fully automated, or with a simpler logic, or targeting a simpler language. We target a real-world programming language and want to enable developers to prove already existing code.

We implemented a higher-order separation logic and operational semantics for a subset of Java using the well-established theorem prover Coq. We are also developing Kopitiam, an Eclipse plugin providing seamless integration of Java coding and proof development of the Java code.

Using these tools we conducted a case study on the fully formalized specification and verification of a Java program that implements snapshotable search tree data structure. A snapshot is a read-only view of a tree. The challenge of this data structure was to provide a specification that enables the client to reason locally to a tree or a snapshots, and allow implementations to share data between a tree, its snapshots and its iterators. We achieve local and modular reasoning for a client of the tree interface: the client of the interface can treat the tree and snapshots each as a disjoint data structure, whereas the implementor uses extensively sharing of the heap to meet performance (time and space) expectations. Our approach uses an idea from; we introduce an abstract predicate global to each tree data structure, consisting of a finite set of abstract structures: a Tree, a Snap or an Iter. The client can reason modularly about an abstract structure, framing out the remaining structures.

1 http://www.itu.dk/research/tomeso
2 https://github.com/hannesm/Kopitiam
Kopitiam is an Eclipse plugin that provides an environment for editing Coq proofs (with syntax highlighting and an outline), interaction with the Coq top level, a proof goal viewer and automatic translation of Java code to our subset of Java. A user of Kopitiam integrates the proof development into the standard software development workflow: refactoring of code can be done intermingled with proof development. A screencast of an early development version of Kopitiam is available at http://www.itu.dk/people/hame/kopitiam.mov. We developed the Kopitiam plugin in the Scala language to benefit from its advanced functional features as well as its ability to interface with Java code (Eclipse).

The ongoing development of Kopitiam includes tighter integration of Java code, formal specifications and Coq proofs. We plan in the future to provide more automation. The ratio between Java coding and proof development should increase, we needed to write roughly 5000 lines of proof script for our snapshotable tree implementation, roughly 100 lines of Java code.

Another part of my research will be how advanced type system concepts like typestate [1] can be used to discharge proof obligations automatically, to reduce the burden of the developer who wants to verify her object-oriented software.

References