Kopitiam: Modular Incremental Interactive Full Functional Static Verification of Java Code

Hannes Mehnert

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

Abstract. We are developing Kopitiam, a tool to interactively prove full functional correctness of Java programs using separation logic by interacting with the interactive theorem prover Coq. Kopitiam is an Eclipse plugin, enabling seamless integration into the workflow of a developer. Kopitiam enables a user to develop proofs side-by-side with Java programs in Eclipse.

1 Introduction

It is challenging to reason about object-oriented programs, because these contain implicit side effects, shared mutable data and aliasing. Reasoning with Hoare logic always has to consider the complete heap, which does not preserve the abstractions of the programming language. Separation logic [18] extends Hoare logic to allow modular local reasoning about programs with shared mutable state.
Coq [4] is an interactive theorem prover based on the calculus of constructions with inductive definitions. Kopitiam generates proof obligations from specifications written in Java, which the user needs to discharge by providing Coq proof scripts. A proof script is a sequence of tactics.
The contribution is Kopitiam, a tool combining the following verification properties:

- **Modular** Extensions of a verified Java library can rely on the specification of the library, without reverifying the library.
- **Incremental** While parts of the code can be verified and proven, other parts might remain unverified, and development of proofs and code can be interleaved, as in Code Contracts [10].
- **Interactive** Automated proof systems like jStar [8] are limited in what they can prove. We use an interactive approach where the user discharges the proof obligations using provided tactics, thus Kopitiam does not limit what a user can prove.
- **Full functional** Given a complete, precise formal specification the proof shows that the implementation adheres to its specification.
- **Static** The complete verification is done at compile time, without execution of the program. Other code verification approaches, like design by contract [15], may depend on run time checks. Especially in mission critical systems, compile time verification is indispensable, since a failing run time check would be disastrous.

The structure of the paper is: we give an overview of Kopitiam in Section 2, demonstrate a detailed example in Section 3, relate Kopitiam to similar tools in Section 4, and in Section 5 conclude and present future work.

2 Overview of Kopitiam

Kopitiam provides an environment that is familiar to both Java programmers and Coq users. Coq developers use Proof General (based on Emacs) or CoqIDE (a self-hosted user interface). Many Java programmers use an IDE for development, the major Java IDEs are Eclipse and IntelliJ. To integrate seamlessly into the normal development workflow we develop Kopitiam as a plugin for Eclipse, so a developer does not have to switch tools to prove her code correct. We base Kopitiam on Eclipse because it is open source, popular and easily extendible via plugins. While an Eclipse integration for Coq [6] already exists, Kopitiam provides a stronger integration of Java code and Coq proofs. This is achieved by a single intermediate representation for both code and proofs. A change to either code or proof directly changes this intermediate representation.

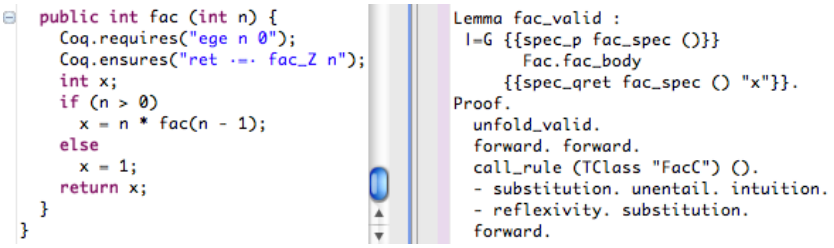

Fig. 1. Java and Coq editor side-by-side; closeup of Coq editor in Fig 2

In Figure 1 Kopitiam is shown. It consists of a standard Eclipse Java editor on the left and a specially developed Coq proof editor on the right. The content of the Java editor is the method `fac`, a recursive implementation of the factorial function. The Java code contains a call to `Coq.requires` and a call to `Coq.ensures`, whose arguments are the pre- and postcondition of the method. The right side shows the Coq lemma `fac_valid`, stating that factorial fulfills its specification, together with parts of the proof script (full code in Section 3). Due to the single intermediate language, Kopitiam reflects every change to the content of one editor to the other editor, e.g. a change to the specification on the Java side changes the Coq proof obligation.

Kopitiam consists of a Java parser, with semantic analysis, a transformer to SimpleJava (presented in Section 2.2), a Coq parser, and communication to Coq via standard input and output. All these parts are expressible in a functional way, so we chose Scala [16] as the implementation language of Kopitiam. Scala is a type-safe functional object-oriented language supporting pattern matching. It compiles to Java bytecode, allowing for seamless integration with Eclipse (every Scala object is a Java object and vice versa). Kopitiam is open source under the Simplified BSD License and available at https://github.com/hannesm/Kopitiam.

2.1 Coq Editor and Goal Viewer

To develop proofs, Kopitiam provides a Coq editor and a goal viewer, shown in Figure 2. The Coq code on the left side states the lemma `fac_step`: for all `n`, `n` greater than 0 implies that `n * fac(n - 1)` equals

`fac(n)` (lines 1-3). All except the last 2 lines of the Coq code that have been processed by Coq (highlighted in blue in Kopitiam, the unprocessed ones are black). The goal viewer on the right side shows the current state of proof assumptions, proof obligations and subgoals. The current state is after doing induction over `n` and discharging the base case using the `intuition` tactic. The remaining proof obligation is the induction step. As in other Coq user interfaces, there are buttons (not shown) to step forward and backward through the proof.

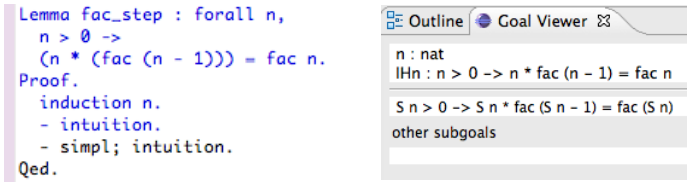

Fig. 2. Coq editor and goal viewer of Kopitiam, closeup of Figure 1

If Coq signals an error while processing, this error is highlighted in Kopitiam. Figure 3 shows on the left side the erroneous Coq proof script next to Eclipse's corresponding problems tab. Errors are indicated by red wiggly lines, similar to the way programming errors are displayed in Eclipse.

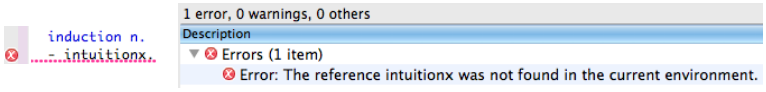

Fig. 3. Coq proof script containing an error and Eclipse's problems tab

2.2 The SimpleJava Programming Language

We formalized SimpleJava, a subset of Java, and implemented it using a shallow embedding in Coq (details in an upcoming paper by Bengtson, Birkedal, Jensen and Sieczkowski). SimpleJava syntax is a prefix (S-expression) notation of Java's abstract syntax tree. Dynamic method dispatch is the core ingredient of object oriented programming, and supported by SimpleJava. A SimpleJava program consists of classes and interfaces. An interface contains a set of method signatures and a set of interfaces, that it inherits from; a class consists of a set of implemented interfaces, a set of fields, and a set of method implementations. A method body consists of a sequence of statements (allocation, conditional, loop, call, field read, field write and assignment) followed by a single return statement. Automatic transformation of unstructured returns to a single return would impose method-global control flow changes; and the SimpleJava code would distract the Java programmer while proving.

3 Example Verification of Factorial

An example program is the factorial, shown in Figure 4. Figure 5 shows the SimpleJava code, automatically translated by Kopitiam. A call (lines

```
class FacC {
 int fac (int n) {
  Coq.requires("ege n 0");
  Coq.ensures
     ("ret ·=· facZ n");
  int x;
  if (n > 0)
   x = n * fac(n - 1);
  else x = 1;
  return x;
 }
}
```

Fig. 4. Java code

```
(cif (egt (var_expr "n") 0)
 (cseq
  (ccall "x" "this" "fac"
   ((eminus
     (var_expr "n") 1))
   (TClass "FacC"))
  (cassign "x"
   (etimes
    (var_expr "n")
    (var_expr "x"))))
 (cassign "x" 1))
```

Fig. 5. SimpleJava code

```
Fixpoint fac n :=
 match n with
 | S n => (S n) * fac n
 | 0 => 1
 end.
Definition facZ :=
fun (n:Z) =>
 match ((n ?= 0)%Z) with
 | Lt => 0
 | _ =>
  Z_of_nat(fac(Zabs_nat n))
 end.
```

Fig. 6. Coq definitions

3-6) consists of the return value binding (`x`), the receiver (`this`), the method (`fac`), the argument list and the receiver class (`TClass "FacC"`). In Figure 6 the fixpoint `fac` is defined, which is the common factorial function on natural numbers. Our Java code uses integers, so we additionally need `facZ`, which extends the domain of `fac` to integers.

The specification of a program consists of specifications for all classes and interfaces. An example specification of method `fac` is shown in Figure 7, whose code is automatically generated by Kopitiam from the Java code (Figure 4). The precondition (line 3 of both Figures) requires that the parameter `n` must be equal or greater (`ege`) than 0. The postcondition (lines 4-5 of both Figures) ensures that the returned value (`ret`) is equal to `facZ n`. The bottom block of Figure 7 defines `Spec`, which connects the specification `fac_s` to the actual program, class `FacC`, method `fac`.

```
Definition fac_s :=
 Build_spec unit (fun _ =>
  (ege "n" 0,
  ((("ret":expr) ·=·
    (facZ ("n":expr))):asn))).

Definition Spec := TM.add
 (TClass "FacC")
 (SM.add "fac" ("n" :: nil, fac_s))
  (SM.empty _)
(TM.empty _).
```

Fig. 7. Specification

```
Lemma fac_valid : |=G {{spec_p Fac_spec.fac_spec ()}}
 Fac.fac_body {{spec_qret Fac_spec.fac_spec () "x"}}.
Proof.
  unfold_valid. forward. forward.
  call_rule (TClass "FacC") ().
  - substitution. unentail. intuition.
  - reflexivity. substitution.
  forward. unentail. intuition. subst. simpl.
    rewrite Fac_spec.facZ_step; [reflexivity | omega].
  forward. unentail. intuition. subst.
    destruct (Z_dec (val_to_int k) 0).
    assert False; [|intuition]. destruct s; intuition.
    rewrite e. intuition.
  Existential 1:=().
Qed.
```

Fig. 8. Coq proof script for factorial

Figure 8 shows the hand-written proof that the Java implementation of factorial satisfies its specification. The proof uses the `forward` tactic [2]. This extracts the first Hoare triple; the resulting proof obligation (Hoare triple) is the original precondition combined with the extracted postcondition, the remaining statement sequence, the original postcondition. If the extracted precondition cannot be discharged trivially, the user has to do it. After applying `forward` twice (line 4, for `cif` and `cseq`), the proof obligation for the call is discharged by the `call_rule` tactic (line 5).

4 Related Work

Several currently available proof tools are compared in Table 1. Only Krakatoa [11], jStar [8] and Kopitiam target Java. Krakatoa uses Why, which uses a simple While language where mutable variables cannot be aliased. The automated proof system jStar targets Jimple [19], a Java intermediate language built from Java bytecode. Kopitiam directly translates from a subset of Java source code to SimpleJava.

Different code contracts [15] implementations focus on C# (Code Contracts [10]) and Java (JML [5]). Code contract implementations translate some non-trivial specifications to run time checks, while we focus on

Name	T	Language	Specification logic	Automation
Krakatoa	sta	Java; While	multi-sorted FOL	several provers
Ynot	sta	higher-order imp	separation logic	Coq tactics
jStar	sta	Java; Jimple	separation logic	user proof rules, SMT
Spec#	dyn	C#	C#/Java	run time assertions
Dafny	inc	imp + generics	Boogie	Z3 (SMT-solver)
Kopitiam	inc	Java; SimpleJava	separation logic	Coq tactics

Table 1. Comparison of verification tools

static verification. The integration of code contracts in an IDE is beneficial, as the developer can incrementally develop code and proofs in the same environment. An example for an industrial grade IDE with code contracts is the KeY tool [1], based on UML and OCL. Code Contracts [10] do not focus on full functional correctness, while some JML tools such as Mobius [3] do. In contrast to those tools, we use separation logic, thus a user does not need to specify frame conditions.

Dafny [14] is a proof tool for an imperative programming language supporting generics and algebraic data types, but not subtyping. Dafny is well integrated into Microsoft Visual Studio and also allows incremental proofs. It provides a multi-sorted first-order logic as specification logic.

Ynot [7] uses a shallow embedding in Coq for a higher-order imperative programming language without inheritance. Thus to verify code with Ynot the program has to be reimplemented in the Ynot tool.

The jStar [8] tool is fully automated and does a proof search on available proof rules, which are extensible by the user. A user can introduce unsound proof rules, since these are treated as axioms and are not verified. Moreover it is difficult to guide the proof search in jStar, since the order of rules matters. Both Ynot and Kopitiam use the proof assistant Coq, in which proof rules have to be proven before usage.

5 Conclusion and Future Work

We are developing Kopitiam, an Eclipse plugin for interactive full functional static verification of Java code using separation logic. Our implementation is complete enough to prove correctness of factorial and in-place reversal of linked lists. We currently do not handle the complete Java language, e.g. unstructured returns and `switch` statements. Class to class inheritance is also not supported. Kopitiam does not support more advanced Java features like generics and exceptions.

We plan to integrate more automation: We will provide context aware suggestions, a technique widely used in Eclipse for code completion, for specifications, whose syntax we also plan to improve. We will provide separation logic lemmas and tactics for Coq, allowing the user to focus on the non-trivial proof obligations. We also want the user to discharge separation logic proof obligations instead of exposing the Coq layer.

We are also working on more and larger case studies ranging from simple object-oriented code (`Cell` and `ReCell` from [17]), to the composite pattern and other verification challenges [20], to real-world data structures like Linked Lists with Views [12] and Snapshottable Trees [9], to the C5 collection library [13], the extensive case study of our research project.

Acknowledgement We want to thank Peter Sestoft, Jesper Bengtson, Joe Kiniry and the anonymous reviewers for their valuable feedback.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Software and System Modeling 4 (2005)
2. Appel, A.W.: Tactics for separation logic. INRIA Rocquencourt and Princeton University, Early Draft (2006)
3. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS proof carrying code infrastructure. In: Boer, F.S., Bonsangue, M.M., Graf, S., Roever, W.P. (eds.) Formal Methods for Components and Objects. Springer Verlag (2008)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: the Calculus of Inductive Constructions. Springer Verlag (2004)
5. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transf. 7 (Jun 2005)
6. Charles, J., Kiniry, J.R.: A lightweight theorem prover interface for Eclipse. UITP at TPHOL'08 (2008)
7. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. ACM Proc. of ICFP '09 (Aug 2009)
8. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. ACM Proc. of OOPSLA '08 (Oct 2008)
9. Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R.: Making data structures persistent. ACM proc. of STOC '86 (Nov 1986)
10. Fähndrich, M., Barnett, M., Logozzo, F.: Embedded contract languages. ACM Proc. of SAC '10 (Mar 2010)
11. Filliâtre, J., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. Proc. of CAV'07 (Jul 2007)
12. Jensen, J.B., Birkedal, L., Sestoft, P.: Modular verification of linked lists with views via separation logic. Proc. of FTfJP 2010 (May 2010)
13. Kokholm, N., Sestoft, P.: The C5 generic collection library for C# and CLI. Tech. Rep. ITU-TR-2006-76, IT University of Copenhagen (2006)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. Proc. of LPAR-16 (Mar 2010)
15. Meyer, B.: Design by contract. Advances in Object-Oriented Software Engineering (1991)
16. Odersky, M., al: An overview of the Scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004)
17. Parkinson, M.J., Bierman, G.: Separation logic, abstraction and inheritance. ACM Proc. of POPL '08 (Jan 2008)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. IEEE Proc. of 17th Symp. on Logic in CS (Nov 2002)
19. Vallée-Rai, R., Hendren, L.J.: Jimple: Simplifying Java bytecode for analyses and transformations. Tech. Rep. 4, McGill University (1998)
20. Weide, B., Sitaraman, M., Harton, H., Adcock, B., Bucci, P., Bronish, D., Heym, W., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. Proc. of VSTTE '08 (Oct 2008)