Specifying Generic Java Programs: two Case Studies*

A. Giorgetti[1,2], C. Marché[3,4], E. Tushkanova[1,2], and O. Kouchnarenko[1,2]

[1]LIFC, Univ. of Franche-Comté, Besançon F-25030
[2]INRIA Nancy - Grand Est, Villers-lès-Nancy F-54600
[3]INRIA Saclay - Île-de-France, Orsay F-91893
[4]LRI, Univ. Paris-Sud, Orsay F-91405

Abstract

This work investigates the question of modular specification of *generic* Java classes and methods. We propose extensions to the *Krakatoa Modeling Language*, a part of the Why platform for proving that a Java or C program is a correct implementation of some specification. The new constructs we propose for the specification of generic Java programs are presented through two significant examples: the specification of the generic method for sorting arrays which comes from the `java.util.Arrays` class in the Java API, and the specification of the `java.util.HashMap` class defining a generic hash map and its use for memoization. The key features are the introduction of parametricity both for types and for *theories* and an instantiation relation between theories. We discuss soundness conditions and their verification.

1 Introduction

The problem of deductive verification of programs is to check that there exists a derivation in the Hoare logic for the Hoare triple, $\{P\}\ S\ \{Q\}$, where P and Q are assertions and S is a program statement. P is called the precondition and Q the postcondition: if the precondition is met and the statement terminates, then it should establish the postcondition.

Why [4] is a platform for deductive verification of source code. From a source program annotated by specifications, it extracts verification conditions and transmits them to provers like *SMT provers* (Simplify, Z3, CVC3, Yices, Alt-Ergo, etc.) or *proof assistants* (Coq, Isabelle/HOL, PVS, etc.) For Java, these

*This work is supported by the INRIA *ARC CeProMi*, `http://www.lri.fr/cepromi/`.

specifications are given in the Krakatoa Modeling Language (KML) [?], a variant of the Java Modeling Language [2, 5].

A new feature introduced in Java 5 is genericity, but it is supported neither by upstream JML nor KML. Supporting genericity naturally requires to add type parameters to specifications, but not only: more complex issues arise when one tries to formally specify generic programs. The goal of this paper is to present those issues on two typical examples, and to propose new specification constructs to solve them.

In Section 2 we first present an excerpt of KML, focusing on the part of that language which allows to specify algebraic-style data types [8] and theories. Section 3 presents a specification for a generic method for sorting arrays (the one from the `java.util.Arrays` class in the Java API): our first contribution is adding type parameters to theories and a notion of *theory parameter* to classes and interfaces. Section 3.4 shows how we deal with client code for this sorting method, emphasizing issues which do not arise in the non-generic case, which we solve by proposing the notion of *theory instantiation*. Section 4 presents a specification of generic hash maps (`java.util.HashMap` class of the API) also with a client code: here we add theory parameters also to theories themselves, in order to specify the expected properties of the map keys. We compare with related work and conclude in Section 5.

2 Overview of the Specification Language

A specification language is a formal language used during requirement analysis and system design. There are algebraic-based specification languages like CASL, Z, B ; and program-oriented ones like JML for Java, Spec# for C#. KML is a specification language for Java, inspired by JML, while sharing many features with ACSL, the ANSI/ISO C Specification Language [?]. The main difference between JML and KML is that the main application of JML is *runtime assertion checking*, whereas KML is designed for automated or assisted *deductive verification*, by statically producing verification conditions. For this reason, KML was designed to allow algebraic-style specifications because they are suitable for theorem proving.

2.1 Basic standard features

Specifications are given as annotations in the source code, in a special style of comments after `//@ ...` or between `/*@` and `*/`.

Method *contracts* have the general form

`//@` ***requires*** R***;*** ***assigns*** L***;*** ***ensures*** E***;***

where R and E are logical assertions, and L is the set of memory locations that may be modified by the method. The *precondition* R is supposed to hold in the method pre-state, for any value of its arguments. It must be checked valid by

the caller. The *postcondition* E must be established by the method's code at the end of its execution. In that formula, `\result` denotes the returned value.

A *class invariant* is declared at the level of class members. It has the form

//@ **invariant** id: P;

where P is a property that must be established by each constructor, and preserved by each method of the class. A *model field* is introduced in the specification with the keyword `model`, and is related to concrete fields with an invariant. Its type must be a logic type. Model fields are used to provide abstract specifications to functions whose concrete implementation must remain private.

An *assertion* in the code has the form

//@ **assert** P;

and specifies that the property P holds at the corresponding program point. The construct `\at`(e, L) refers to the value of the expression e in the program state at label L. There exist predefined labels, e.g. `Old` and `Here`. `\old`(e) is in fact syntactic sugar for `\at`$(e,$ `Old`$)$. The label `Here` is visible in all statement annotations, where it refers to the state where the annotation appears. It refers to the pre-state in a method precondition (`requires` clause), and to the post-state in a method postcondition (`ensures` clause). The label `Old` is visible in `ensures` clauses and refers to the pre-state of the method's contract. More details can be found in [**?**].

2.2 Logical specifications

KML does not allow pure methods to be used in annotations but it permits to declare new logic functions and predicates. They must be placed at the global level, i.e. outside any class declaration, and respectively have the form

//@ **logic** $t\ id(t_1\ x_1, \ldots, t_n\ x_n) = e$;
//@ **predicate** $id(t_1\ x_1, \ldots, t_n\ x_n) = p$;

where e has type t, and p is a proposition. The types t, $t_1, \ldots, t_n$ can be either Java types or purely logic types. The logic types of mathematical integers and reals are built-in and respectively denoted `integer` and `real`.

Logic functions and predicates can also be *hybrid*. It means that they depend on some memory state. More generally, they can depend on several memory states, by attaching them several labels. The general form of a hybrid function and predicate definition respectively is

//@ **logic** $t\ id\{L_1, \ldots, L_k\}(t_1\ x_1, \ldots, t_n\ x_n) = e$;
//@ **predicate** $id\{L_1, \ldots, L_k\}(t_1\ x_1, \ldots, t_n\ x_n) = p$;

where $L_1, \ldots, L_k$ are memory state labels on which the function or predicate depends, and t, $t_1, \ldots, t_n$, e and p are as before.

A predicate may also be defined by an *inductive* definition of the form

```
/*@ inductive P{L_1,...,L_k}(t_1 x_1,...,t_n x_n) {
      case c_1 : p_1;
      ...
      case c_m : p_m;
    } */
```

where $c_1, \ldots, c_m$ are identifiers and $p_1, \ldots, p_m$ are propositions. The semantics of this definition is that P is the least fixpoint of the cases, i.e. the smallest predicate (in the sense that it is false the most often) satisfying the propositions $p_1, \ldots, p_m$. To ensure existence of a least fixpoint, it is required that each of these propositions is of the form `\forall` $y_1, \ldots, y_m,\ h_1$ `==>` $\ldots$ `==>` h_l `==>` $P(t_1, \ldots, t_n)$ where P occurs only positively in hypotheses $h_1, \ldots, h_l$.

Finally, a set of types, functions and predicates can be declared *axiomatically* by an algebraic-style axiomatization. Such a set forms a block of declarations called a *theory*. It has the following general form

```
theory Th {
    type id;
    logic t id{L_1,...,L_k}(t_1 x_1,...,t_n x_n);
    predicate id{L_1,...,L_k}(t_1 x_1,...,t_n x_n);
    axiom id_1 : p_1;
    axiom id_2 : p_2;
    ⋮
}
```

Arbitrarily many types, functions, predicates and axioms can be given. Notice that functions and predicates are only given by their profiles. Unlike inductive definitions, there are no syntactic conditions which would guarantee theories to be consistent. It is up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

3 A Generic Sorting Function

An array sorting function is a routine that modifies the order of elements in a given array. The resulting array must satisfy two properties: (1) the array elements are a permutation of the elements at the beginning; (2) the elements are in increasing order w.r.t. some ordering relation.

Filliâtre and Magaud [3] study several algorithms for sorting, and both specify and prove them correct with the Why tool, but only on the particular instance of an array of integers and the usual "less-than" order. Predicates specifying the meaning of an array to be in increasing order, and of two arrays being permutation of each other, are defined with the Coq proof assistant [?]; and proof scripts of generated verification conditions are fulfilled manually within Coq. A selection sorting algorithm is written in Java by Marché [?] with similar specifications of predicates done in KML. It improves over the former work because

proofs are done fully automatically with SMT provers (Simplify and Alt-Ergo). However, it is still specific to integers and the usual less-than order.

We go further by specifying a generic method for sorting arrays, where the array elements are of any type T. A significant challenge is to specify the ordering relation which is given as a parameter, under the form of a comparison function on T. As we will see, it is also important to study how this generic specification can be used by *client* code, because it has to be instantiated.

3.1 Generic sorting in Java

The class `java.util.Arrays` defines a generic sorting method with the profile:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

In this method `<T>` is a type parameter and the syntax `<? super T>` denotes an unknown type that is a supertype of `T` (or `T` itself). The `java.util.Comparator<T>` interface imposes a total ordering on some collection of objects.

```
interface Comparator<T> {
  public int compare(T x, T y);
}
```

`T` is the type of objects that may be compared by this comparator. The method `compare` is expected to return a negative integer, zero, or a positive integer when the first argument is respectively less than, equal to, or greater than the second one; for the desired ordering relation.

The sample code given in Figure 1 illustrates an instance of use of this `sort` method. It ends with a simple assertion which we expect to be able to prove, as a consequence of the generic specification we will provide. The validity of this assertion indeed depends on the comparator we choose. Here it is an instance of the class `IntLtComparator` given in Figure 2, which implements the usual "less-than" ordering on integers. Changing this comparator say to the "greater-than" ordering would of course sort the array in decreasing order instead, violating the assertion.

```
class Main {
  public static void main(String[] args) {
    IntLtComparator intc = new IntLtComparator();
    Integer[] b = {new Integer(2),new Integer(1),new Integer(3)};
    java.util.Arrays.sort(b,intc);
    //@ assert b[0].value <= b[1].value;
  }
}
```

Figure 1: A sample client code calling the generic sorting method

```
class IntLtComparator implements Comparator<Integer> {
   public int compare(Integer x, Integer y) {
       if (x.intValue() < y.intValue()) return −1;
       if (x.intValue() == y.intValue()) return 0;
      return 1;
   }
}
```

Figure 2: The usual "less-than" comparator on integers

3.2 Type parameters: the permutation property

The first extension we propose to KML is to allow type parameters in algebraic specifications, as follows

//@ ***predicate*** $id\langle T_1, \ldots, T_l\rangle\{L_1, \ldots, L_k\}(t_1\ x_1, \ldots, t_n\ x_n) = p$;

and similarly for functions, inductive predicates, and such.

For the sorting example, and following [?], we define a predicate $\texttt{Permut}\langle T\rangle\{L_1, L_2\}(a, l, h)$ which means that the part of array a between indexes l and h, in some program state L_1 is a permutation of the same array part in state L_2. It is defined inductively in Figure 3. The first postcondition of the `sort` method is then specified below.

```
/*@ ensures Permut<V>{Old,Here}(a,0,a.length−1); */
public static <V> void sort(V[] a, Comparator<? super V> cmp);
```

3.3 Theory parameters: the sorting property

The challenge is to specify the behavior of the comparator given as argument. What we propose is to allow to *pass theories as parameters*. On the sorting

---

```
predicate Swap<T>{L1,L2}(T a[], integer i, integer j) =
 \at(a[i],L1) == \at(a[j],L2) && \at(a[j],L1) == \at(a[i],L2) &&
 \forall integer k; k != i && k != j ==> \at(a[k],L1) == \at(a[k],L2);

inductive Permut<T>{L1,L2}(T a[], integer l, integer h){
 case Permut_refl{L}: \forall T a[], integer l h;
   Permut<T>{L,L}(a, l, h);
 case Permut_sym{L1,L2}: \forall T a[], integer l h;
   Permut<T>{L1,L2}(a, l, h) ==> Permut<T>{L2,L1}(a, l, h);
 case Permut_trans{L1,L2,L3}: \forall T a[], integer l h;
   Permut<T>{L1,L2}(a, l, h) &&
     Permut<T>{L2,L3}(a, l, h) ==> Permut<T>{L1,L3}(a, l, h);
 case Permut_swap{L1,L2}: \forall T a[], integer l h i j;
   l <= i <= h && l <= j <= h &&
      Swap<T>{L1,L2}(a, i, j) ==> Permut<T>{L1,L2}(a, l, h);
}
```

Figure 3: The permutation predicate

```
theory ComparatorTheory<T> {
  predicate eq{L}(T x, T y);
  axiom eq_ref{L}: \forall T a; eq{L}(a,a);
  axiom eq_sym{L}: \forall T a b; eq{L}(a, b) ==> eq{L}(b,a);
  axiom eq_trans{L}: \forall T a1 a2 a3;
    eq{L}(a1, a2) && eq{L}(a2,a3) ==> eq{L}(a1,a3);

  predicate lt{L}(T x, T y);
  axiom lt_irref{L}: \forall T a; ! lt{L}(a,a);
  axiom lt_antisym{L}: \forall T a1 a2; !(lt{L}(a1,a2) && lt{L}(a2,a1))
  axiom lt_trans{L}: \forall T a1 a2 a3;
    lt{L}(a1,a2) && lt{L}(a2,a3) ==> lt{L}(a1,a3);
  axiom lt_totality{L}: \forall T a1 a2;
    eq{L}(a1,a2) || lt{L}(a1,a2) || lt{L}(a2,a1);

  predicate leq{L}(T x, T y) = eq{L}(x,y) || lt{L}(x,y);

  predicate sorted{L}(T[] a, integer l, integer h) =
    \forall integer i; l <= i <h ==> leq{L}(\at(a[i],L),\at(a[i+1],L));
}
```

Figure 4: General theory for `Comparators`

```
interface Comparator<U> /*@ <Th instantiating ComparatorTheory<U> > */ {

  /*@ ensures (Th.lt(x,y) <==> \result < 0) &&
              (Th.eq(x,y) <==> \result == 0) &&
              (Th.lt(y,x) <==> \result > 0);    */
  public int compare(U x, U y);
}
```

Figure 5: Specification of the `Comparator` interface

---

example, the first step is to define a general theory for types equipped with an ordering relation. Figure 4 shows a theory named `ComparatorTheory` which defines two predicates `eq` for equality and `lt` for an arbitrary strict total order. Equality is reflexive, symmetric and transitive. The strict total order satisfies four properties: irreflexivity, antisymmetry, totality and transitivity.

The `Comparator` interface should take some comparison theory as a parameter. This is shown in Figure 5. The `Comparator` interface thus has two parameters: a Java type `U` and a theory `Th`. The syntax `Th instantiating ComparatorTheory<U>` says that `Th` is an instance of the general theory defined in Figure 4. One may wonder why we require an *instance* of the comparison theory `ComparatorTheory`: this will be discussed in Section 3.4.

Figure 6 specifies with a second postcondition the sorting property of the method `sort`. The sorting method is not only parameterized by the type `V` but also by the type `W` which denotes the super type of `V` on which the comparator operates, and by a theory `th` which can be any instance of the general

```
/*@ ensures th.sorted(a,0,a.length-1); */
public static <V> /*@ <W> <th instantiating ComparatorTheory<W> > */
  void sort(V[] a, Comparator<? /*@ as W */ super V> /*@ <th> */ cmp) {
}
```

Figure 6: Specification of the generic sorting method

```
public final class Integer extends Number implements Comparable {
    private int value;

    /*@ assigns this.value; ensures this.value == v; */
    public Integer(int v) { this.value = v; }

    /*@ assigns \nothing; ensures \result == this.value; */
    public int intValue() { return this.value; }
}
```

Figure 7: Annotated `Integer` class

```
theory IntLtComparatorTheory instantiates ComparatorTheory<Integer> {
  predicate eq{L}(Integer x, Integer y) = \at(x.value == y.value, L);
  predicate lt{L}(Integer x, Integer y) = \at(x.value < y.value,L);
}
```

Figure 8: Theory for "less-than" comparison

`ComparatorTheory` on `W`. Notice the new `as` keyword added to relate the anonymous Java type denoted by `?` and the explicit name `W` we need to introduce for it in the specification.

The comparator type is itself instantiated with the Java type `W` and the theory `th`. In the method postcondition the predicate `sorted` is then qualified with this theory.

3.4 Theory Instantiation

To deal with our client program, we need more annotations. First we add specifications to the `java.lang.Integer` class: Figure 7 shows an excerpt of it annotated in KML[1].

Then, we need to specify the `IntLtComparator` class of Figure 2. For that, we first provide a theory which instantiates the general comparison theory `ComparatorTheory`, in Figure 8. The goal is to formalize that we decided to compare with the "less-than" ordering. The `instantiates` declaration generates verification conditions: this is discussed in Section 3.5. The class `IntLtComparator` shown in Figure 9 implements the instantiation of the `Comparator` interface where the Java type is the `Integer` class and the theory is the comparison

[1] In KML, the private field is visible in the annotations of the public methods, whereas in JML, the field should be annotated with modifier `spec_public`.

```
class IntLtComparator
  implements Comparator<Integer> /*@ <IntLtComparatorTheory> */ {
    public int compare(Integer x, Integer y) { ... }
}
```

Figure 9: Specification of the `IntLtComparator` class of Figure 2

theory for this class, defined in Figure 8.

Finally, notice that when checking the specific call to method `sort` in the client program, the "implicit" parameters `V`, `W` and also the theory `Th` must be guessed. The value of `V` comes as usual with the Java typing, the value of `W` comes from the type of the `Comparator`. Then the theory `th` must be inferred from the theory argument of `cmp` and it must be checked whether it really instantiates a convenient comparison theory.

3.5 Verification Conditions for Soundness

The soundness conditions to generate are as follows. First, the theory `IntLtComparatorTheory` should be a valid instantiation of the theory `ComparatorTheory<Integer>`. This amounts to check that the definitions of the predicates `eq` and `lt` given in `IntLtComparatorTheory` satisfy the axioms given in `ComparatorTheory<T>` when the type variable `T` is instantiated with `Integer`. This condition is easily discharged by SMT provers. Second, the class `IntLtComparator` should correctly implement the interface `Comparator<Integer>`, which requires that the method `compare` in the `IntLtComparator` class should satisfy the specification of the method `compare` declared in the interface `Comparator<U>`, when the type parameter `U` is instantiated with `Integer` and the theory parameter `Th` is instantiated with `IntLtComparatorTheory`. This condition is again easily proved by SMT provers. Finally, checking the assertion in our client code can be done, by instantiating the generic postcondition `th.sorted`(...) of `sort` with the `IntLtComparatorTheory`. Substituting the `th.lt` predicate with its actual definition does the job. It is important to notice here that this final substitution is necessary, so it justifies why we initially parameterized the `sort` method with a theory parameter: otherwise, we would know that the array is sorted w.r.t some order, without knowing precisely which one.

4 Generic Hash Maps

We present additional constructs needed when specifying generic *hash maps*. These are data types which build finite mappings from indexes of some type *key* to values of some other type *data*. Finding the value associated to a given index is made efficient by use of classical hashing techniques.

A simple but illustrating example of use of hash maps is a method for com-

```
class Fib {
    HashMap<Integer,Long> memo;

    Fib() { memo = new HashMap<Integer,Long>(); }

    public long fib(int n) {
        if (n <= 1) return n;
        Integer n_obj = new Integer(n);
        Long x = memo.get(n_obj);
        if (x == null) {
            x = new Long(fib(n−1)+fib(n−2));
            memo.put(n_obj,x);
        }
        return x.longValue();
    }
}
```

Figure 10: Java source for Fib class

puting *Fibonacci numbers*[2]: $F(0) = 0$, $F(1) = 1$, and $F(n+2) = F(n+1)+F(n)$ for $n \geq 0$. To avoid the exponential complexity of the naive recursive algorithm, we apply the general technique of *memoization*. A Java `Fib` class with a `fib` method computing Fibonacci numbers with memoization is shown in Figure 10.

4.1 Specification of the Fibonacci sequence

A mathematical definition of the Fibonacci sequence as a theory is given below.

```
theory Fibonacci {
  logic integer math_fib(integer n);
  axiom fib0: math_fib(0) == 0;
  axiom fib1: math_fib(1) == 1;
  axiom fibn: \forall integer n; n >= 2 ==>
       math_fib(n) == math_fib(n−1) + math_fib(n−2);
}
```

The expected behavior of the `fib` method is specified as follows.

```
//@ requires n >= 0; assigns \nothing; ensures \result == math_fib(n);
public long fib(int n);
```

Notice that issues related to arithmetic overflow are ignored. We just assume for simplicity that computations are made on unbounded integers.

4.2 Theories for hashable objects and hash maps

The first step is to define a theory which provides a predicate for testing equality, and a hash function. This theory is given in Figure 11. The dots are for the

[2]From CeProMi collection of challenging examples, `http://www.lri.fr/ceprom`i. This is only for illustration, since there exist other efficient ways to compute Fibonacci numbers.

```
theory HashableTheory<T> {
  ... // equality theory as in Figure 4
  logic integer hash{L}(T x);
  axiom hash_eq{L}: \forall T x,y;
    eq{L}(x,y) ==> hash{L}(x) == hash{L}(y);
}
```

Figure 11: Theory of hashable objects

```
theory Map<K><Th instantiating HashableTheory<K> > {
  type t<V>;
  logic <V> V acc{L}(t<V> m, K key);
  logic <V> t<V> upd{L}(t<V> m, K key, V value);

  axiom <V> acc_upd_eq{L}: \forall t<V> m, K key1 key2, V value;
    Th.eq{L}(key1,key2) ==> \at(acc(upd(m,key1,value),key2) == value,L);

  axiom <V> acc_upd_neq{L}: \forall t<V> m, K key1 key2, V value;
    ! Th.eq{L}(key1,key2) ==>
      \at(acc(upd(m,key1,value),key2) == acc(m,key2),L);
}
```

Figure 12: Theory of maps

same axiomatization of the `eq` predicate as in Figure 4. The important part of this theory is the axiom `hash_eq` specifying the expected property for the hash function: two equal objects must have the same hash code.

Then, we provide a theory for maps, as shown in Figure 12. This theory is parameterized by both a type `K` for the keys and a theory for equality and hashing of `K` objects. The type of data is not given as a parameter to the theory itself, but as a parameter `V` of the type of maps. This allows to use the same theory of maps for several instances of `V`. This theory is indeed the classical *theory of arrays* which is a typical theory supported by SMT provers. It is defined by a function `acc` to access the element indexed by some key, and a function `upd` which provides a so-called *functional update* of a map, returning a new map in which the element associated to some key is changed. The behavior of these two functions is axiomatized by the two axioms in Figure 12, which makes use of the equality predicate on keys. It has to be noticed that specifying the proper equality relation on keys is one of the issues in this specification, and our proposal to use parameterized theories is an answer to this issue.

The resulting specification of the generic `java.util.HashMap` class is shown in Figure 13. Since the type variable `K` of keys in `HashMap<K><V>` implicitly extends `Object`, it inherits the `equals` and `hashCode` methods defined in the `java.lang.Object` class. These two methods should be specified in the `Object` class with some theory instantiating `HashableTheory<K>`, as shown in Figure 14. A new issue arises here from dynamic dispatch: the instance of `Object` satisfying the `HashableTheory` is not known yet. Our proposal is to introduce another

```
class HashMap<K,V> /*@ <Th instantiating HashableTheory<K> >
                      @ constraint: K extends Object<K><Th> */
{
 //@ theory M = Map<K><Th>;
 //@ model M.t<V> m;

  /*@ requires x instanceof K; assigns \nothing;
    @ ensures \result != null ==> \result == M.acc(m,(K)x) ; */
  V get(Object x);

  /*@ requires k != null; assigns m;
    @ ensures m == M.upd(\old(m),k,v); */
  void put(K k, V v);
}
```

Figure 13: Specification of the `HashMap` class

```
public class Object /*@ <T><H instantiating HashableTheory<T> > */ {
 /*@ requires this instanceof T && o instanceof T;
   @ ensures \result == true <==> H.eq((T)this,(T)o); */
 public boolean equals(Object o) { ... }

 /*@ requires this instanceof T;
   @ ensures \result == H.hash((T)this); */
 public int hashCode(){ ... }
}
```

Figure 14: Specification of two methods in the `Object` class

```
theory HashableInteger instantiates HashableTheory<Integer> {
  predicate eq{L}(Integer x, Integer y) = \at(x.value == y.value, L);
  logic integer hash{L}(Integer x) = \at(x.value, L);
}
```

Figure 15: Theory of equality and hashing of `Integer`s

type parameter T in the specification, to be bound later. To ensure that the theory given as argument to `HashMap` class is an `HashableTheory` on the right type, a *constraint* is posed (Figure 13) on the type `K`. Notice also the use of local naming of a particular instance of a theory: the name `M` is given to the theory of `Map`s instantiated on the type of keys and on its theory of equality and hashing.

4.3 Instantiating generic hash maps

The generic `HashMap` class being specified, we can use it in the `Fib` class. The first step is to provide an instance of the theory of equality and hashing on `Integer`s. This is done in Figure 15. A proper implementation of the `Integer` class is then given in Figure 16.

```
class Integer extends Object /*@ <Integer><HashableInteger> */ {
    boolean equals(Object o) {
        if (o instanceof Integer) return this.value == ((Integer)o).value;
        return false;
    }
    int hashCode() { return this.value; }
}
```

Figure 16: Implementation of hashable `Integer`s

```
class Fib {
  HashMap<Integer,Long> /*@ <HashableInteger> */ memo;
  /*@ invariant memo_fib: memo != null && \forall Integer x, Long y;
    @   x != null && y == memo.M.acc(memo,x) && y != null ==>
    @     y.value == math_fib(x.value);   */
  ... // fib as specified in Section 4.1
}
```

Figure 17: Class invariant of the `Fib` class

In order to prove the `fib` postcondition, it is mandatory to provide a class invariant which, informally, states that for any pair (x, y) stored in the `memo` map, $y = F(x)$. The class invariant for the `Fib` class can be written as in Figure 17.

4.4 Verification Conditions for Soundness

Verification conditions come first from the `instantiates` declaration of Figure 15. It should be proved that the given definitions satisfy the axioms given in `HashableTheory<T>` (Figure 11), when the type variable `T` is instantiated with `Integer`, which is straightforward. The constraint of Figure 13 comes immediately from the declaration of `Integer` class.

Other verification conditions come classically: invariant should be established by the `Fib()` constructor, which comes from a simple specification of `HashMap()` left to the reader; it should be preserved by any call to `fib()`, which comes from a simple reasoning by case distinction. The preconditions to calls to `get()` and `put()` are straightforward, and the postcondition of `fib()` follows from the invariant. However, one verification condition cannot be discharged, the one from the ***assigns*** clause: the contract says that there are no side-effects at all, whereas in reality the private `memo` hash map can be modified. See below for a discussion.

5 Related Work and Perspectives

Extensions of JML to Java 5 features are proposed by Cok [?]. Similarly, Ulbrich [?] analyzes what would be the consequences of adding support for generics

in KeY. Both of them do not address the issue of higher-order methods such as the `sort()` method, for which we propose theory parametricity. Stenzel *et al.* [?] propose another approach to support generics, which they implemented in the KIV tool. They focus on the *heap pollution* issue, arising at runtime when a dynamic type of a parametric type object does not match its static type. Their approach amounts to change the underlying program logics, which for them is defined directly in Isabelle. Since KIV does not have its own specification language at the Java source level, they do not address the issues of theory parameters and instantiation. Nevertheless, heap pollution is an issue that we should address too.

Shaner *et al.* [?] address the issue of higher-order method independently of generics. Their approach is based on *model programs* and is certainly suitable for runtime checking, but it is not for deductive verification: because every instance of, say the `sort()` method, would need to be re-proved, thus loosing modularity of the verification, hence loosing the advantage of genericity.

Supporting higher-order methods in deductive verification is particularly difficult when functional parameters do have side-effects, an issue we avoid in this paper since our `compare()` method is pure. Higher-order functions and side-effects issues are mainly studied in the context of functional languages, e.g. [?]. Indeed our approach mainly comes from the domain of functional languages, where parametricity in module systems has been largely studied [?]. Extensions of modules with subtyping and inheritance were proposed, e.g [?], but in a context of side-effect-free languages.

This is on-going work and it clearly remains to formalize the proposed constructions, to express what are the necessary verification conditions in general, and to show a soundness result. At the end of previous section we met an open issue known as hidden side-effects [8]: side-effects may occur in the internal state of an object, that one wants to hide in its public interface. Implementing an abstract view of a component using hidden concrete variables is the general approach of *refinement* which is not yet deeply investigated in the context of object-oriented programming [10]. Another future work is to apply a similar approach to the formal specification of `C` programs, i.e by extending the ACSL language [?].

Acknowledgments We would like to thank C. Paulin, A. Paskevych, W. Urribarrí, A. Tafat, R. Bardou and J. Kanig for helpful discussions and suggestions.

References

[1] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An Overview of JML Tools and Applications. In *FMICS 03*, volume 80 of *ENTCS*, pages 73–89. Elsevier, 2003.

[3] J. Chrzaszcz. Implementing modules in the Coq system. In *TPHOLs'03*, volume 2758 of *LNCS*, pages 270–286. Springer, 2003.

[4] D. R. Cok. Adapting JML to generic types and Java 1.6. In *SAVCBS '08*, pages 27–34, 2008.

[5] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the Coq system. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[6] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV'07*, volume 4590 of *LNCS*, pages 173–177. Springer, July 2007.

[7] J. Kanig and J.-C. Filliâtre. Who: A Verifier for Effectful Higher-order Programs. In *ACM SIGPLAN Workshop on ML*, Aug. 2009.

[8] G. T. Leavens and Y. Cheon. Design by Contract with JML. Available from `http://www.jmlspecs.org`, 2006.

[9] C. Marché. Towards modular algebraic specifications for pointer programs: a case study. In *Rewriting, Computation and Proof*, volume 4600 of *LNCS*, pages 235–258. Springer, 2007.

[10] C. Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, Jan. 2009. `http://krakatoa.lri.fr/ws/`.

[11] J. Mitchell, S. Meldal, and N. Madhav. An extension of standard ML modules with subtyping and inheritance. In *POPL '91*, pages 270–278. ACM, 1991.

[12] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of OOPSLA'07*, pages 351–368. ACM, 2007.

[13] K. Stenzel, H. Grandy, and W. Reif. Verification of Java programs with generics. In *AMAST'08*, number 5140 in LNCS, pages 315–329. Springer, 2008.

[14] A. Tafat, S. Boulmé, and C. Marché. A refinement methodology for object-oriented programs. `http://www.lri.fr/cepromi/`, 2009.

[15] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, July 2006. `http://coq.inria.fr`.

[16] M. Ulbrich. Software verification for Java 5. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, 2007.