Modular Specification of Java Programs

Elena Tushkanova, Alain Giorgetti, Claude Marché, Olga Kouchnarenko

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Modular Specification of Java Programs

Elena Tushkanova — Alain Giorgetti — Claude Marché — Olga Kouchnarenko

N° 7097

November 2009

ISSN 0249-6399 ISRN INRIA/RR--7097--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SACLAY - ÎLE-DE-FRANCE

Modular Specification of Java Programs

Elena Tushkanova , Alain Giorgetti , Claude Marché , Olga Kouchnarenko

Thème : Programmation, vérification et preuves
Équipes-Projets PROVAL et CASSIS

Rapport de recherche n° 7097 — November 2009 — 26 pages

Abstract: This work investigates the question of modular specification of generic Java classes and methods. The first part introduces a specification language for Java programs. In the second part the language is used to specify an array sorting algorithm by selection. The third and the fourth parts define a syntax proposal for the specification a generic Java programs, through two examples. The former is the specification of the generic method for sorting arrays which comes in the `java.util.Arrays` class of the Java API. The latter is the specification of the `java.util.HashMap` class and its use for memoization.

Key-words: Formal Specification, Verification, Proof, Automated Reasoning, SMT Provers, Krakatoa Modeling Language

This work was partly supported by the INRIA National Action “CeProMi”, `http://www.lri.fr/cepromi/`, and by the ANR National Project “CAT” (ANR-05-RNTL-0030x).

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université
4, rue Jacques Monod, 91893 ORSAY Cedex
Téléphone : +33 1 72 92 59 00

Spécification modulaire de programmes Java

Résumé : Ce travail cherche à répondre à la question de spécification modulaire de classes et de méthodes génériques en Java. La première partie présente un langage de spécification pour Java. Dans la seconde partie ce langage est utilisé pour spécifier un tri par sélection. Les troisième et quatrième parties proposent une syntaxe pour spécifier des programmes Java génériques, à travers deux exemples. Le premier exemple est une spécification de la méthode générique pour trier des tableaux définie dans la classe `java.util.Arrays` de l'API Java. Le deuxième exemple est une spécification de la classe `java.util.HashMap` et son utilisation pour la mémorisation des résultats d'une fonction.

Mots-clés : Spécification formelle, Vérification, Preuve, Démonstration automatique, Prouveurs SMT, Krakatoa Modeling Language

Contents

List of Figures

1 Introduction

The work presented in the present report has been done in the framework of the INRIA CeProMi[1] "Action de Recherche Collaborative" (ARC). One of the objectives of the ARC is modular specification and proof of properties of Java or C programs. A well conceived program is developed in a modular way, that is by the structured assembly of simpler components. The goal is also to get modularity to prove modular programs. Some members of the ARC project develop a specification language for Java programs called the Krakatoa Modeling Language (KML).

This work especially addresses the question of modular specification of the so-called generic Java classes and methods. It proposes extensions of the KML language to allow specifications of these, which essentially amount to

- add type parametricity in that specification language;
- add a notion of instantiation of *theories* used to model programs.

Our proposal is illustrated on two examples. The former is the specification of an algorithm to sort a Java array. The latter is the specification of the `java.util.HashMap` class and its use for memoization.

Existing works [3, 6] on sorting algorithm specification handle a particular instance of an array of integers. They use a permutation datatype to prove that the resulting array is a permutation of the initial array. Our suggestion is to re-use the bag datatype defined in [8] to say that the initial array and the sorted array have the same content.

We go further by specifying a generic sorting algorithm, where array elements are of any type T and the ordering is given as a parameter, under the form of a comparison function on T.

This document is organized as follows. Section 2 presents the Krakatoa Modeling Language, a new specification language for Java programs. Section 3 proposes original specifications for a sorting algorithm and discusses their automatic proof. Section 4 presents new specification constructs for specifying a generic Java method for sorting arrays. Section 5 presents additional constructs needed when specifying generic hashmaps.

2 Specification Language

A specification language is a formal language used in computer science during requirement analysis and system design. Most programming languages are directly executable formal languages. They are used to implement a system. Specification languages are generally not directly executed. They describe the system at a much higher level than a programming language. There are many specification languages like CASL, JML, Spec#, Z, B, etc.

This section describes a specification language for the Java programming language, named Krakatoa Modeling Language (KML). KML is a new specification language for Java programs. It is designed to reduce the distance between programming and proving activities.

Why is a generic platform for program verification [4]. From a source program annotated by definite specifications, the Why platform extracts the proof obligations and transmits them to provers like Simplify, Yices, Alt-Ergo, etc. The Krakatoa tool is a part of the Why platform. Krakatoa expects a Java source file as input, annotated with the Krakatoa Modeling Language. KML is largely inspired from the Java Modeling Language [2, 5]. KML specifications are given as annotations in the source code, in a special style of comments after `//@ ...` or between `/*@` and `@*/`. KML also shares many features with the ANSI/ISO C Specification Language [9].

2.1 Method contracts

Method contracts are made of a precondition and a set of behaviors. The precondition is a proposition introduced by `requires` keyword which is supposed to hold in the pre-state of the method, i.e. when it is called. It must be checked valid by the caller.

A normal behavior has the form:

[1] `http://www.lri.fr/cepromi`

```
/*@ requires R;
  @ behavior b:
  @  assigns L;
  @  ensures E;
  @*/
```

`R` and `E` are logical assertions, `R` is a precondition and `E` is a postcondition, `L` is a set of memory locations, that may be modified by the method. In `E`, the notation `\result` denotes the returned value.

An exceptional behavior as the form

```
/*@ requires R;
  @ behavior b:
  @  assigns L;
  @  signals (Exc x) E;
  @*/
```

The semantics is similar to normal behaviors, but here properties must hold when the method terminates abruptly with exception `Exc`.

2.2 Statement annotations

A **loop annotation** can be given just in front of a loop construct (while, for, etc.). It has the form

```
/*@ loop_invariant I
  @ for b: loop_invariant Ib;
  @ loop_variant V;
  @*/
```

It states that `I` is an inductive invariant: it must hold at loop entry and be preserved by any iteration of the loop body. The loop invariant `Ib` must also be an inductive invariant. The loop variant, if given, must be an expression of integer type, which must decrease at each loop iteration, and remain non-negative.

2.3 Class invariants

A class `invariant` is a property attached to a class. It has the form

```
/*@ invariant id: e; @*/
```

This property must be established by constructors, and preserved by each method of the class.

2.4 Logic functions and predicates

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

```
 //@ logic m id(m1 x1, .. , mn xn) = e;
 //@ predicate id(m1 x1, .. , mn xn) = p;
```

where `x1`, ..., `xn` are variables, `e` must have type `m`, and `p` must be a proposition. The types `m` and `mi` can be either Java types or purely logic types: integer, 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 several labels to them. A hybrid function and a predicate definition are of the following general form

```
//@ logic m id{L1, .. , Ln}(m1 x1, .. , mn xn) = e;
//@ predicate id{L1, .. , Ln}(m1 x1, .. , mn xn) = p;
```

where `L1`, ..., `Ln` are memory state labels on which the predicate or function depends, and `m`, `m1`, ... , `mn`, `e`, `p` have the same definition as presented before.

2.5 Lemmas

Lemmas are user-given propositions, a facility that might help theorem provers to establish validity of KML specifications. A lemma is declared as

```
//@ lemma id: p;
```

Obviously, a complete verification of a KML specification must provide a proof for each lemma.

2.6 Inductive definitions

A predicate may also be defined by an *inductive* definition.

```
/*@ inductive P(x1, ..., xn) {
  @   case c1 : p1;
  @   ...
  @   case cn \verb:pn;
  @ }
  @*/
```

where `c1`, ..., `cn` are identifiers and `p1`, ..., `pn` 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 `p1`, ..., `pn`. To ensure existence of a least fixpoint, it is required that each of these propositions is of the form

`\forall y1, ..., ym, h1 ==> ... ==> hl ==> P(t1, ..., tn)`

where `P` occurs only positively in hypotheses `h1`, ..., `hl`.

2.7 Theories

Logical specifications were supported by Krakatoa/Why under the form of axiomatic blocks in Java source files within specification comments by declaring a set of types, a set of predicates and functions with expected profiles, and a set of axioms. Now it is defined in a separated file with the ".spec" extension. The syntax for defining a `theory` is the same as for an axiomatic block but without comments, as in the following example:

```
theory Th {
  type new_type;
  logic new_type func1;
  logic integer func2(new_type v, integer k);
  axiom axiom_name: axiom_body;
}
```

where `Th` is the theory name and `axiom_body` is a closed formula.

Unlike inductive definitions, there is no syntactic conditions which would guarantee axiomatic definitions to be consistent. It is usually up to the user to ensure that the introduction of axioms does not lead to a logical inconsistency.

2.8 Construct \at and default logic labels

Construct `\at(e,id)` refers to the value of the expression `e` in the state of label `id`. There are four predefined logic **labels**: `Pre`, `Here`, `Old` and `Post`. `\old(e)` is in fact syntactic sugar for `\at(e,Old)`.

1. 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).
2. The label `Old` is visible in `assigns` and `ensures` and refers to the pre-state of the method's contract.
3. The label `Pre` is visible in all statement annotations, and refers to the pre-state of the function it occurs in.

More details could be found in [7].

This section has shortly described the Krakatoa Modeling Language. In the next section this specification language is used to prove a sorting algorithm.

3 Specification of a Sorting Algorithm

A sorting algorithm is an algorithm that puts elements of an array in a certain order. The resulting array must satisfy the following two properties:

1. The elements are in increasing order with respect to some ordering relation.
2. The elements in the sorted array are a permutation of the elements of the initial array.

Filliâtre and Magaud [3] study several algorithms for sorting. They specify and prove them correct within the Why tool, but only on the particular instance of an array of integers and the usual "less-than" order. The first condition is specified by a predicate `(sorted t i j)` which expresses that array `t` is sorted in increasing order between the bounds `i` and `j`. The second condition is specified by a predicate `(permut t tt)` where `t` and `tt` are permutations of each other. They describe many ways to define such a predicate, but the best solution is to express that the set of permutations is the smallest equivalence relation containing the transpositions, i.e. exchanges of two elements. The predicate `(exchange t tt i j)` is defined for two arrays `t` and `tt` and two indexes `i` and `j`, and the predicate `(permut t tt)` is defined inductively for the following properties: reflexivity, symmetry and transitivity. The proofs are performed within the Coq proof assistant [1].

A selection sorting algorithm is written in Java by Marché [6] with a similar specification in KML. It is also specific to integers and the usual less-than order. The proof is done fully automatically within SMT solvers (namely Simplify and Alt-Ergo provers).

Our proposal is to re-use the bag datatype defined in [8] and to rewrite the second condition by saying that the initial array and the resulting array have the same content.

This section is organized as follows: Section 3.1 presents the sorting algorithm by selection in Java, Section 3.2 presents this algorithm completed with a specification in KML. In Section 3.3 we specify the array content with a bag and try to prove the sorting algorithm automatically.

3.1 Selection sort in Java

The sorting algorithm by selection in Figure 1 is written in Java. There are two methods: `swap` method just exchanges two array elements of given indexes. In `selectionSort` method the integers `i` and `mi` are indexes for the current element and the minimal element respectively. The integer `mv` serves to store this minimal element. The minimal element is found in the remainder of the array and swapped with the current element.

This algorithm can be tested with different array examples, but it is not sure that it is always correct, i.e. it satisfies properties 1 and 2 for any array. A formal specification of these two properties constitutes the first step towards a formal proof of its correctness.

```
1. class Sort {
2.   /** method swapping 2 elements */
3.   void swap(int t[], int i, int j) {
4.     int tmp = t[i];
5.     t[i] = t[j];
6.     t[j] = tmp;
7.   }
9.   void selectionSort(int t[]) {
10.    int i, j;
11.    int mi, mv;
12.    for (i = 0; i < t.length - 1; i++) {
13.      mv = t[i];
14.      mi = i;
15.      for (j = i + 1; j < t.length; j++) {
16.        if (t[j] < mv) {
17.          mi = j;
18.          mv = t[j];
19.        }
20.      }
21.      swap(t, i, mi);
22.    }
23.  }
24.}
```

Figure 1: Selection sort in Java

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

Figure 2: Specification of the first property

3.2 Sorting algorithm with a KML specification

In [6] two postconditions for method `selectionSort` are proved:

```
behavior sorts:
   ensures Sorted(t,0,t.length-1);
```

which means that the resulting array is in increasing order, and

```
behavior permuts:
   ensures Permut{Old,Here}(t,0,t.length-1);
```

which means that the resulting array is a permutation of the initial array.

The `Sorted` predicate is presented in Figure 2. It is a hybrid predicate. It means that its value depends on the memory heap in some state `L`. The `Permut` predicate presented in Figure 3 has two labels. This predicate is true whenever the slice of the array `a` from lower bound `l` to upper bound `h` in the state `L1` is a permutation of the same slice in the state `L2`. The predicate defines four properties: reflexivity, symmetry, transitivity and swap. The last case tells us that swapping two elements in the slice is a permutation. The `Swap` predicate is reproduced in Figure 4. `Swap{L1,L2}(a,i,j)` is true if and only if the value of `a[i]` in the state of label `L2` equals the

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

Figure 3: Inductive predicate `Permut`

```
predicate Swap{L1,L2}(int 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);
```

Figure 4: Predicate `Swap`

value of `a[j]` in the state of label `L1`, the value of `a[j]` in the state of label `L2` equals the value of `a[i]` in the state of label `L1`, and the value of `a[k]` is the same in both states, if `k` is different from `i` and `j`.

Figure 5 presents two loop invariants for the two loops in Figure 1. The loop invariants for the `sorts` behavior tell that the array is sorted up to index `i` in the external loop and that `mv` is a minimal element between `a[i]` and `a[j]` in the internal loop. The loop invariant for the `permuts` behavior is the same for the external and internal loops. It tells that the current array is a permutation of the initial array.

The algorithm with this specification is proved within the Simplify prover, except the postcondition for the `selectionSort` method which tells that the resulting array is a permutation of the initial array. This postcondition is proved within the Alt-Ergo prover. So, this algorithm is proved within the Simplify and the Alt-Ergo provers.

The proof results are satisfactory. However, we want to explore another way for the second property by reusing a bag datatype. More precisely, we try to prove the same algorithm but with a property 2 saying that the initial array and the resulting array have the same content.

3.3 Specifying the sorting algorithm by selection with a bag

A bag (or multiset) is a collection without order. We want to associate to each array the bag of its elements, and to express that the output array is a permutation of the input array by writing that the corresponding bags are the same. It is a new way to prove property 2.

The type of bags is described by the functions on Figure 6 and the set of first-order axioms on Figure 7 that present some properties of bags. The first four axioms tell that `union` is associative, commutative and that the

```
/*@ loop_invariant 0 <= i;
  @ for sorts:
  @  loop_invariant Sorted(t,0,i) &&
  @    (\forall integer k1 k2 ;
  @       0 <= k1 < i <= k2 < t.length ==> t[k1] <= t[k2]);
  @ for permuts:
  @  loop_invariant
  @   Permut{Pre,Here}(t,0,t.length-1);
  @*/
for (i = 0; i < t.length-1; i++) {
    mv = t[i];
    mi = i;
    /*@ loop_invariant
      @   i < j &&
      @   i <= mi < t.length &&
      @   mv == t[mi];
      @ for sorts:
      @  loop_invariant
      @   (\forall integer k; i <= k < j ==> t[k] >= mv);
      @ for permuts:
      @  loop_invariant
      @   Permut{Pre,Here}(t,0,t.length-1);
      @*/
    for (j = i + 1; j < t.length; j++) {
     ...
    }
 ...
}
```

Figure 5: Loop invariants

```
type ibag;

// empty bag
logic ibag empty_bag();

// singleton(n)
logic ibag singleton(integer n);

// remove element n from bag b
logic ibag remove(integer n, ibag b);

// union b1 and b2
logic ibag union(ibag b1, ibag b2);
```

Figure 6: Signature for bags

`empty_bag` is a neutral element for the union of bags. The last axiom shows a relation between `union` and `remove`. When an element is removed from the union of a bag `b` and the bag containing only this element, the result is the bag `b`.

```
axiom union_assoc:
 \forall ibag b1 b2 b3;
   union(union(b1,b2),b3) == union(b1,union(b2,b3));
axiom union_comm:
 \forall ibag b1 b2; union(b1,b2) == union(b2,b1);
axiom union_empty_id_left:
 \forall ibag b; union(empty_bag(),b) == b;
axiom union_empty_id_right:
 \forall ibag b; union(b,empty_bag()) == b;

axiom remove_union:
 \forall ibag b, integer x;
   remove(x,union(singleton(x),b)) == b;
```

Figure 7: Algebraic specification of bags

```
logic ibag boundContent{L1}(int[] a,
                            integer i, integer j) reads a[i..j];

axiom emptyContent{L1}:
  \forall int[] a; \forall integer i j;
      (i > j ==> boundContent{L4}(a,i,j) == empty_bag());

axiom nonemptyContent{L1}:
  \forall int[] a, integer i j;
      i <= j ==> boundContent{L4}(a,i,j) ==
            union(boundContent{L4}(a,i+1,j),singleton(a[i]));
```

Figure 8: Hybrid function for array content

```
..
 ensures boundContent{Old}(a,0,a.length-1) ==
           boundContent{Here}(a,0,a.length-1);
..
```

Figure 9: Postcondition for `selectionSort` and `swap` methods

Figure 8 declares a hybrid function which takes an array, a lower and an upper bounds as parameters and returns a bag. The KML `reads` keyword says that `boundContent` just reads the array between `i` and `j`, it does not modify it. The first axiom says that `boundContent` returns the empty bag if the lower bound is greater than the upper bound. The second axiom says that, otherwise, the resulting bag is the union of the singleton bag containing the first array element and the content of the remaining part of the array.

It should be proved that the `swap` and `selectionSort` methods do not change the content of the slice as shown in Figure 9.

This postcondition is proved for the `selectionSort` method, but is not proved for the `swap` method. The `selectionSort` method depends on the `swap` method, therefore, it is easy to prove, but proving the `swap` method requires induction because `boundContent` is inductively defined. To prove the `swap` method we must guide provers step by step with assertions which are presented in Figure 10. The first assertion tells that the new

```
void swap(int a[], int i, int j) {
  int tmp = a[i];
  a[i] = a[j];
  /*@ for cont: assert
    @   boundContent{Here}(a,0,a.length-1) ==
    @     union(remove(\at(a[i],Pre),
    @            boundContent{Pre}(a,0,a.length-1)),
    @      singleton(\at(a[j],Pre)));
    @*/
  /*@ for cont: assert
    @    a[j] == \at(a[j],Pre);
    @*/
    Middle: {
  a[j] = tmp;
  /*@ for cont: assert
    @   boundContent{Here}(a,0,a.length-1) ==
    @     union(remove(\at(a[j],Middle),
    @            boundContent{Middle}(a,0,a.length-1)),
    @      singleton(tmp));
    @*/
  }
}
```

Figure 10: Assertions to guide provers step by step

content is obtained from the old content by replacing the old value of `a[i]` by the value of `a[j]` in the old content. The second assertion tells that the value of `a[j]` has not changed. Then the memory state between states `Old` and `Here` is labelled `Middle`. The last assertion tells that the new content is obtained from the previous content by removing the previous value of `a[j]` and adding the value of the local variable `tmp` (which is the old value of `a[i]`).

Moreover, the lemma presented in Figure 11 is added. It says that whenever the elements of an array are the same at two states, except in some position `k`, then the array content at the second state can be obtained from its content at the first state by removing the element at position `k` in the first state and adding the element at position `k` in the second state.

Table 1 presents proof obligations (POs) proved by five provers: Alt-Ergo 0.8, Simplify 1.5.4, Yices 1.0.21, Z3 2.2 and CVC3 2.1. There are nine POs: one PO for the `lemma`, four POs for the `selectionSort` method and four POs for the `swap` method. Since the lemma itself is not provable without induction it is proved by none of these provers. All POs are proved by the Simplify prover. Unlike Simplify the SMT solvers Alt-Ergo,Yices, Z3 and CVC3 fail to prove some of the POs.

As a conclusion, the sorting algorithm is proved for array elements with the `int` Java type. Nevertheless, we would like to prove this algorithm for every Java type, that is as a generic sorting algorithm. It is the matter of the next section.

4 Generic Sorting

Java generics are a language feature that allows definition and use of generic types and methods. Generics are needed for implementing a generic class that can be instantiated for a variety of types.

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

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

```
lemma UpdateContent{L1,L2}:
  \forall int[] a, integer i j k;
     // update of a[k]
     i <= k <= j &&
     (\forall integer l;
         i <= l <= j && k != l ==>
           \at(a[l],L1) == \at(a[l],L2))
     ==> boundContent{L2}(a,i,j) ==
           union(
             remove(\at(a[k],L1),
                    boundContent{L1}(a,i,j)),
             singleton(\at(a[k],L2)));
```

Figure 11: New lemma

Table 1: Result table

	Proof obligations	Alt-Ergo 0.8	Simplify 1.5.4	Yices 1.0.21	Z3 2.2	CVC3 2.1
	Lemma	−	−	−	−	−
POs for the selectionSort method	Loop invariants	+	+	−	+	−
	Postcondition presented in Figure 9	+	+	+	+	−
	Postcondition using the `Sorted` predicate	−	+	−	−	−
	Pointer dereferencing	+	+	−	+	−
POs for the swap method	Postcondition presented in Figure 9	−	+	−	−	−
	PO for the `assigns` clause	−	+	−	+	−
	Postcondition using the `Swap` predicate	+	+	−	+	+
	Pointer dereferencing	+	+	−	+	+
	Number of proved POs	5	8	1	6	2

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). Notice that the `java.util.Comparator<T>` interface imposes a total ordering on some collection of objects.

```
class Main {
 public static void main(String[] args) {
  IntComparator intc = new IntComparator();
  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 12: Main class where the sorting method is called

```
class IntComparator 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 13: IntComparator class

```
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` compares its two arguments for order. It returns a negative integer, zero, or a positive integer when the first argument is less than, equal to, or greater than the second one.

The simple program given on Figure 12 illustrates an instance of use of this `sort` method. In the `main` method, `intc` is an instance of the class `IntComparator` (Figure 13), which implements the interface `java.util.Comparator<T>` instantiated with the class `Integer` which wraps a value of the primitive type `int` in an object.

The client code ends with a simple assertion which we expect to be able to prove, as a consequence of the generic specification we will provide.

In the following, we propose a set of specifications for the classes and interfaces involved in this example, in ordre to be able to prove the assertion of the main program.

4.1 Specification of `Integer` class

An excerpt of the `Integer` class of package `java.lang` annotated in KML is given on Figure 14. Notice that the private field is visible in the annotations of the public methods. In JML, the field should be annotated with modifier `spec_public` to allow that. In KML, private fields are automatically visible in annotations.

4.2 Specification of the `Comparator` interface

Figure 15 presents our interface `Comparator` where the `compare` method is specified with a postcondition. This postcondition is based on two predicates `sto` and `eq`. The first one means a strict total order and the second one means equality.

```
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 14: Annotated `Integer` class

```
interface Comparator<U> {
   /*@ ensures (\result == -1 <==> sto(x,y)) &&
     @          (\result == 0 <==> eq(x,y)) &&
     @          (\result == 1 <==> sto(y,x));
     @*/
  public int compare(U x, U y);
}
```

Figure 15: Specification of the `Comparator` interface, first version

But where are `sto` and `eq` defined? Figure 16 shows a theory named `ComparatorTheory` which defines these two predicates. Equality is reflexive, symmetric and transitive. The strict total order satisfies four properties: irreflexivity, antisymmetry, totality and transitivity.

4.3 Specifying the sorting behavior

Figure 17 presents the specification of the method `sort` which tells that the array is sorted.

The predicate `sorted` is defined in Figure 16. It means that the array is sorted in increasing or decreasing order. Two array elements are compared with the total order `to` also defined in Figure 16 from `sto` and `eq`.

From the specification of the `compare` method (Figure 15) provers cannot understand where `sto` and `eq` predicates are. So, these predicates have to be qualified with the theory where they are declared as shown in Figure 18.

However, when the Comparator interface is instantiated with the `Integer` Java type, for example, we expect the predicates to be instantiated with an instance of the `ComparatorTheory`. For instance, when the `Comparator` interface is instantiated with the `Integer` Java type, the `eq` and `sto` predicates can be defined as in the `IntComparatorThery` reproduced in Figure 19.

The `Comparator` interface should take some comparison theory as a parameter. We suggest to replace Figure 18 with Figure 20. Now, the `Comparator` interface has two parameters: a Java type `U` and a theory `Th`.

```
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 sto{L}(T x, T y);
  axiom sto_irref{L}:
     \forall T a; ! sto{L}(a,a);
  axiom sto_antisym{L}:
     \forall T a1 a2; ! (sto{L}(a1,a2) && sto{L}(a2,a1))
  axiom sto_totality{L}:
     \forall T a1 a2; eq{L}(a1,a2) || sto{L}(a1,a2) ||
       sto{L}(a2,a1);
  axiom sto_trans{L}:
     \forall T a1 a2 a3; sto{L}(a1,a2) &&
       sto{L}(a2,a3) ==> sto{L}(a1,a3);

  predicate to{L}(T x, T y) = eq{L}(x,y) || sto{L}(x,y);

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

Figure 16: General theory for `Comparators`

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

Figure 17: Specification of the generic sorting method, first version

The syntax `Th instantiating ComparatorTheory<U>` says that `Th` is an instance of the general theory defined in Figure 16.

The class `IntComparator` shown in Figure 21 implements the instantiation of the `Comparator` interface where the Java type is the `Integer` class and the theory is the comparison theory for this class, defined in Figure 19.

Now the complete sorting method specification should be as shown in Figure 22. The sorting method takes an array `a` and a comparator `cmp` as parameters. The comparator type has itself two parameters: a Java type `W` which is a supertype of `V` and a theory `Th` instantiating the general comparison theory `ComparatorTheory`. In the method postcondition the predicate `sorted` is qualified with this theory.

```
interface Comparator<U> {
 /*@ ensures \result == −1 <==> ComparatorTheory(U).sto(x,y) &&
   @         \result == 0 <==> ComparatorTheory(U).eq(x,y) &&
   @         \result == 1 <==> ComparatorTheory(U).sto(y,x);
   @*/
  public int compare(U x, U y);
}
```

Figure 18: Specification of `Comparator` interface, second version

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

Figure 19: Theory for Integer comparison

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

  /*@ ensures \result == −1 <==> Th.sto(x,y) &&
    @         \result == 0 <==> Th.eq(x,y) &&
    @         \result == 1 <==> Th.sto(y,x);
    @*/
  public int compare(U x, U y);
}
```

Figure 20: Specification of `Comparator` interface, final version

```
class IntComparator
  implements Comparator<Integer> /*@ IntComparatorTheory */ {

  public int compare(Integer x, Integer y) {
    ...
  }
}
```

Figure 21: Specification of the `IntComparator` class

4.4 Specifying the permutation behavior

We specify the second expected property, that the resulting array is a permutation of the original one. Following [6], we introduce a hybrid predicate, defined inductively on Figure 23. The new and simple extension we need is the addition of a type parameter T to denote the type of array elements.

```
/*@ behavior sorts:
  @    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 22: Specification of the generic sorting method, final version

```
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 23: Permutation predicate

The second behavior of `sort` method is then given on Figure 24.

4.5 Verification conditions for soundness

The soundness condition for the theory `IntComparatorTheory` to instantiate the theory `Comparator-Theory<Integer>` is that the definitions of `eq` and `sto` given in `IntComparatorTheory` satisfy the ax-

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

Figure 24: Permutation behavior of `Arrays.sort`

ioms given in `ComparatorTheory<Integer>` when the type variable `T` is instantiated with `Integer`. This condition is easily discharged by SMT provers.

Another verification condition is generated for the implementation `IntComparator` (see Figure 21) of the interface `Comparator<U>`. The method `compare` defined in the `IntComparator` class should satisfy the specification of the method `compare` declared in the interface `Comparator<U>` when the theory parameter `Th` is instantiated with `IntComparatorTheory`. This condition is again easily proved by SMT provers (up to the question of null pointers that is not addressed here).

5 Generic Hashmaps

We investigate now the specification of generic *hashmaps*. 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.

Among the CeProMi collection of challenging examples[2], there is a simple but illustrating example of use of hashmaps: a method for computing *Fibonacci numbers*:

$$\begin{aligned} F(0) &= 0, \\ F(1) &= 1, \text{ and} \\ F(n+2) &= F(n+1) + F(n) \text{ for } n \geq 0. \end{aligned}$$

To avoid the exponential complexity of the naive recursive algorithm, we apply the general technique of *memoization*. Notice that there exists other efficient ways to compute Fibonacci numbers, this example is just considered as a simple illustration of memoization techniques in general.

A Java `Fib` class with a `fib` method computing Fibonacci numbers with memoization is shown on Figure 25.

5.1 Specification of the Fibonacci sequence

A mathematical definition of the Fibonacci sequence as a theory is given on Figure 26. The expected behavior of the `fib` method is specified as on Figure 27. Notice that issues related to arithmetic overflow are ignored. We just assume for simplicity that computations are made on unbounded integers.

5.2 A theory for hashable objects

The first step is to define a theory which provides a predicate for testing equality, and a hash function. This theory is given on Figure 28. The essential part of this theory is the lemma `hash_eq` which specifies the expected property for the hash function: two equal objects must have the same hash code.

We can then provide a theory for maps, as shown on Figure 29. This theory is parametrized 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

[2] `http://www.lri.fr/cepromi`

```
import java.util.HashMap;

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 25: Java source for Fibonacci sequence

```
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);

}
```

Figure 26: Theory for the Fibonacci sequence

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

Figure 27: Specification of the `fib` method

the theory itself, but as a parameter `V` of the type of maps. This allows using the same theory of maps for several instances of `V`.

```
theory HashableTheory<T> {

  predicate eq{L}(T x, T y);

  axiom eq_refl{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);

  logic integer hash{L}(T x);

  axiom hash_eq{L}: \forall T x,y;
    eq{L}(x,y) ==> \at(hash(x) == hash(y),L);

}
```

Figure 28: Theory for 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) == access(m,key2),L);

}
```

Figure 29: Theory of maps

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 of Figure 29, which make an essential 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 of use of theories is an answer to this issue.

```
interface Hashable /*@ as T */
    /*@ <Th instantiating HashableTheory<T> > */ {

    /*@ requires (o instanceof T);
      @ ensures \result == true <==> Th.eq(this,(T)o);
      @*/
    boolean equals(Object o);

    /*@ ensures \result == Th.hash(this);
      @*/
    int hashCode();
}
```

Figure 30: Interface for `Hashable` objects

```
class HashMap<K,V> /*@ <Th instantiating HashtableTheory<K> >
                      @ constraint: K implements Hashable<Th>
                      @*/
{

 //@ theory M = Map<K>(Th);

 //@ model M.t<V> m;

  /*@ requires x instanceof K;
    @ 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 31: Specification of the `HashMap` class

Building HashMap for type K of keys should be allowed only if K implements methods `equals` and `hashCode` in a way compatible with some theory instantiating `HashableTheory<K>`. To express that we introduce an interface for hashable objects as on Figure 30. The specification of the generic `java.util.HashMap` class is shown on Figure 31. A *constraint* is posed on the type of keys to relate it with a proper `HashableTheory`. Notice the use of local naming of a particular instance of a theory: the name `M` is given to the theory of `Maps` instantiated on the type of keys and on its theory of equality and hashing. This naming mechanism is certainly a construction that we should offer in practice.

```
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 32: Theory of equality and hashing of `Integers`

```
class Integer implements Hashable /*@ <HashableInteger> */

    boolean equals(Object o) {
        if (o instanceof Integer)
            return this.value == ((Integer)o).value;
        return false;
    }

    int hashCode() { return this.value }
}
```

Figure 33: Implementation of hashable `Integers`

5.3 Instantiating generic `HashMaps`

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 `Integers`. This is done on Figure 32. A proper implementation of the `Integer` class is then given on Figure 33.

Specification of the Fib invariant In order to prove the `fib` method behavior, it is mandatory to provide a class invariant which, informally, states that for any pair (x, y) stored in the `memo` map, $y = fib(x)$. The class invariant for the `Fib` class can be written as in Figure 34.

The verification conditions corresponding to the "instantiates" declaration amount to prove that the given definitions satisfy the axioms of reflexivity, symmetry and transitivity, which are obvious in that case; and the `hash_eq` axiom which is obvious too.

Hopefully, the given constructions in the specifications should allow to prove formally the expected behavior of the `fib` method. However, there is a final missing element that should be investigated further: the contract says that there is no side-effects at all, whereas in reality the private `memo` hashmap can be modified. This problem is known as the issue of hidden side-effects [8].

6 Conclusion

We have described the specification language KML for the Java programming language. The sorting algorithm by selection is proved by using a **hybrid function** which takes an **array** as a parameter and returns a **bag**. A bag is a collection without order. Given an array, this function returns the bag of its elements. We have expressed that the output array is a permutation of the input array by writing that the corresponding bags are the same.

```
class Fib {
  HashMap<Integer,Long> /*@ <HashableInteger> */ memo;

  /*@ invariant memo_fib:
    @  \forall Integer x, Long y;
    @   y == memo.M.acc(memo,x) && y != null ==>
    @     y.value == math_fib(x.value);
    @*/

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

Figure 34: Class invariant of the `Fib` class

The works in [3] and [6] suggest to define an inductive predicate to axiomatize the property that the output array is a permutation of the input array. Specifying with bags is another way to prove this property. But this new way of specifying a sorting algorithm leads to some difficulties. For instance, the `swap` method cannot be proved automatically without additional assertions and lemma which itself is unprovable without induction. These assertions are required by provers to succeed their proofs.

During this work the support of theories and its definition were implemented within the WHY platform. The theories are defined in a separated file with the ".spec" extension. The syntax for defining a theory is the same as the one for an axiomatic block, but without comments.

We explained that in order to formally specify generic methods and classes, it is necessary to extend existing notion of theories in specification languages like KML. It is not only mandatory to add type parametricity in theories, but also to provide a notion of parametricity of theories and a corresponding notion of theory instantiation.

The extensions we propose are essentially inspired by existing notion of languages implementing higher-order logic, namely the notion of modules and functors as they exist for example in Coq, or the notion of type classes [10].

This is an on-going work and it clearly remains to formalize the proposed constructions, to express what are the necessary proof obligations in general, and to show a soundness result. Also, a relation must be established between our approach and similar approaches based on refinement techniques [11].

Another future work is to apply a similar approach to the formal specification of C programs. We plan to propose similar constructs for extending the ACSL language [9]. An issue is that C type system is even weaker than Java generics, for example the `qsort()` function in the C standard library is made generic via the use of `void` pointers, and function pointer to pass the comparison function as argument. `void*` plays the same role as the `?` in Java, which means that the annotation language will need to annotate each occurrence of `void*` by a regular type variable.

Acknowledgments

We would like to thank Christine Paulin, Andriy Paskevych, Wendi Urribarri, Asma Tafat, Romain Bardou and Johannes Kanig for helpful discussions and suggestions.

References

[1] The Coq Proof Assistant. Available at `http://coq.inria.fr/`.

[2] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. 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.-C. Filliâtre and N. Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, 1999.

[4] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, July 2007.

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

[6] C. Marché. Winter School: instruction for lab session/Krakatoa tool. Available at `http://krakatoa.lri.fr/ws/`.

[7] C. Marché. The Krakatoa tool for Deductive Verification of Java Programs. Available from `http://krakatoa.lri.fr/ws/`, 2009.

[8] Claude Marché. Towards modular algebraic specifications for pointer programs: a case study. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 2007.

[9] J.-C. Filliâtre P. Baudin and C. Marché. ACSL: ANSI/ISO C Specification Language. Available from `http://frama-c.cea.fr/acsl.html`, 2008.

[10] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *21th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science. Springer, August 2008.

[11] Asma Tafat, Sylvain Boulmé, and Claude Marché. A refinement methodology for object-oriented programs. `http://www.lri.fr/cepromi/index.php/Publications`, 2009. Submitted.

Centre de recherche INRIA Saclay – Île-de-France
Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 Orsay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399