Verification of Java programs with generics

Kurt Stenzel, Holger Grandy, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Stenzel, Kurt, Holger Grandy, and Wolfgang Reif. 2008. "Verification of Java programs with generics." In *Algebraic Methodology and Software Technology: 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, proceedings*, edited by José Meseguer and Grigore Roşu, 315–29. Berlin: Springer. https://doi.org/10.1007/978-3-540-79980-1_24.

Verification of Java Programs with Generics

Kurt Stenzel, Holger Grandy, and Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Augsburg
86135 Augsburg Germany
{stenzel,grandy,reif}@informatik.uni-augsburg.de

Abstract. Several proof systems allow the formal verification of Java programs, and a specification language was specifically designed for Java. However, none of these systems support generics that were introduced in Java 5. Generics are very important and useful when the collection framework (lists, sets, hash tables etc.) is used. Though they are mainly dealt with at compile time, they have some effect on the run-time behavior of a Java program. Most notably, *heap pollution* can cause exceptions. A verification system for Java must incorporate these effects. In this paper we describe what effects can occur at run time, and how they are handled in the KIV system [18] [2]. To the authors knowledge, this makes KIV the first verification system to support Java's generics.

1 Introduction

The Java programming language [9] was from the beginning very popular with respect to a formal treatment. Alves-Foss published early results (many of which dealt with Java's type system) in 1999 in [1]. Later work focused more on the specification and verification of Java programs. The *Java Modeling Language* JML [15] [19] allows the specification of Java programs in a language similar to Java itself and is supported by many tools [6]. Several tools support the formal verification of Java programs: the KeY tool [5], the LOOP compiler [14] using PVS, or Krakatoa [21], to name just three. ESC/Java2 [7] and Jack [4] are static checkers for Java that use underlying automated theorem prover(s) for their reasoning.

Impressive applications have been specified and verified. Many verification systems and case studies focus on Java Card [26] programs. This makes sense, because programs running on Smart Cards are typically security critical. They handle electronic cash (e.g. the Mondex card [22]), act as official documents, or contain important personal information like finger prints, or health records. A programming error could have serious consequences. And, from a verification point of view, the programs are small, and do not employ all features of the Java language. Examples are [13] [23] [11].

But there is a problem with 'normal' Java: The Java language evolves, and every few years new features are introduced that have a significant impact on a verification system. The same is true for C#. Java 1 [8] was released 1996,

Java 2 [16] in 2000, and introduced inner classes. Java 5 [9] in 2005 added generics, annotations, enums, autoboxing, and other features (Java 6 added Scripting). Experience shows that it is very difficult for the developers of verification systems to keep up with the new features. (The same is true for formal API specifications.) They were designed without formal methods in mind, and it is very hard to estimate how difficult it is to include these features in a prover, and what their effects on actual proofs are without actually doing it. Furthermore, one may feel that these features are not really necessary. They simplify programming in Java, but if a program is to be formally proved anyway, this may not seem important.

We do not feel this way. We believe that there are programs worth verifying that use generics, and that it is important to analyze the effects of specific language features on formal proof systems. These and other results should be taken into account in the design of future programming languages.

Several groups are currently trying to support Java generics for formal specification or verification purposes. The JML developers are "working on Java 1.5 (generics)" [20]; "reason about Java 1.5 source" is "ongoing work" for ESC/-Java3 [17], and the KeY group has evaluated the consequences of supporting generics in the KeY prover [27]. Spec# [3] supports generics for C#, but this is easier because no heap pollution can occur.

This paper describes how generics are incorporated in the KIV prover. The results can probably be adapted to other proof systems with little effort. It turns out that generics have only a slight impact on run-time verification, mainly because of heap pollution. The rest of the paper is organized as follows: Section 2 gives a short introduction to generics from a user's (i.e. a programmer's) point of view, and section 3 describes the phenomenon of heap pollution. Section 4 gives a short introduction to the Java calculus in KIV, and the next two sections describe in detail the effects of generics on run-time behavior. Section 7 reports on results, and concludes.

2 Generics in Java

Generics were introduced in Java 5; they are described in the third edition of the Java Language Specification (JLS 3) [9]. This section provides only a very cursory overview that is focused on the run-time behavior of generics. Wildcards, or bounds are omitted since they are relevant only when their type erasure is computed (see Sect. 6).

Generic types are very useful for collections, e.g. lists. In Java 4, nothing is known about the elements of a list. When an element is retrieved with `li.get(0)` the result is of type `Object`. If a programmer uses a list of integers (i.e. he knows that all elements will be integers) the result must be cast anyway: `(Integer)li.get(0)`. This can lead to an exception at run time if inadvertently a list of strings is supplied. The type system of Java without generics does not help in this case. With generics it is possible to declare a parameterized type `List<Integer>`. In this case the compiler will prevent the programmer from supplying a list of strings (of type `List<String>`) where a

list of integers is expected, or to add a string to a list of integers. Additionally, no cast is necessary. Listing 1 shows a small example. The example includes two other features that were introduced in Java 5: *autoboxing* (automatic conversion of primitive types into their object counterpart and back) in lines 6, 11, 12, and the *enhanced* `for` *statement* in line 6.

```
1  import java.util.*;
2  public class Example1 {
3
4      public int sum(List<Integer> li) {
5      int res = 0;
6      for(int i : li) res += i;
7      return res;
8      }
9      public void example1() {
10     List<Integer> li = new ArrayList<Integer>();
11     li.add(5);
12     li.add(7);
13     System.out.println(sum(li));
14     }
15 }
```

Listing 1. An example with parameterized lists

Trying to call the `sum` method with a list of strings (of type `List<String>`) does not compile. The `List` interface is generic; it is declared as

```
public interface List<E> extends Collection<E> {
```

Here, `E` is a type variable that is instantiated with `Integer` in the example. The `add` method used in lines 11 and 12 is declared as `boolean add(E e);` and the `get` method (that is used implicitly in the loop) as `E get(int index)`. This means the declared result type of the `get` method is the type variable `E`; if `E` is instantiated to `Integer` the compiler knows that the result will be of type `Integer`.

The most important aspect of generics with respect to formal verification is the fact that generics are "forgotten" at run time ("some type information is erased during compilation", JLS 3 p. 56). The reason is compatibility with existing code; see the discussion in JLS 3, p. 57. In fact, the byte code produced for the `sum` method in Listing 1 is identical to the byte code produced by the source code in listing 2.

This code does not use generics. Furthermore, the enhanced `for` loop has been replaced by a standard loop that uses an `Iterator` to access the list elements. Line 4 contains an explicit cast of the list element to `Integer`. Without it the code does not compile. However, the cast will produce a `ClassCastException` at run time if `sum` is called with a list of strings. The source code in Listing 2 compiles in Java 4, and also in Java 5. This may be unexpected because the parameterized interface `List` is used without an instance for the element type (A parameterized class or interface without its parameters is called a *raw* type).

```
1  public int sum(List li) {
2      int res = 0;
3      for (Iterator iter = li.iterator(); iter.hasNext(); ) {
4          int i = ((Integer)iter.next()).intValue();
5          res += i;
6      }
7      return res;
8  }
```

Listing 2. The same example without generics

But it is legal in Java 5 for compatibility reasons as mentioned above: Otherwise it would not be possible to reuse existing class files that were compiled with Java 4. Usage of the raw type (among others) gives rise to an *unchecked* warning by the Java 5 compiler:

> Note: Some input files use unchecked or unsafe operations.

The code in listing 1 does not produce any compilation warnings. Still the byte code for listing 1 is the same as for listing 2. Especially the cast to `Integer` is contained in the byte code. This has implications for a formal verification in the presence of *heap pollution*, and will be explained in detail in sections 5 and 6.

3 Heap Pollution

Heap pollution is described in JLS 3, 4.12.2.1:

> It is possible that a variable of a parameterized type refers to an object that is not of that parameterized type. This situation is known as *heap pollution*. This situation can only occur if the program performed some operation that would give rise to an unchecked warning at compile-time.

Heap pollution can lead to a `ClassCastException` at run time. Listing 3 shows a simple example.

```
1       public int sum(List<Integer> li) {
2       int res = 0;
3       for (int i : li) res += i; // throws
4       return res;
5       }
6       public void example3() {
7       List li = new ArrayList<String>(); // raw type
8       li.add("foo");
9       List<Integer> lii = (List<Integer>)li; // ok
10      System.out.println(sum(lii));
11      }
```

Listing 3. An example for heap pollution

In line 7 a raw type is used and a list containing a string is created. In line 9 this list is assigned to a variable of type `List<Integer>`, causing heap pollution. The code compiles, but running `example3()` causes a `ClassCastException` in line 3. Line 9 does not cause a `ClassCastException` because the cast effectively checks whether the argument is of type `List` – the type parameter is *erased* and not available (and hence not checked) at run time. The byte code for line 3 contains an explicit cast to `Integer` as described in the previous section, causing the exception. The behavior of the code is the same if `li` is passed directly to the `sum()` method.

Heap pollution can occur even without involvement of the heap, and it is easy to write very obfuscated programs where it is difficult to guess whether they will compile, and what their run time behavior will be. Listing 4 contains an example for this.

```
1  public class Example4<X>{
2
3      public X m(boolean flag){
4      if(flag) return (X) "string";
5      else return (X) Integer.valueOf(3);
6      }
7      public static void main(String[] args){
8      Example4 ex = new Example4<String>();
9      Example4<Integer> ex4 = ex; // raw type
10     Integer x = ex4.m(false);   // ok
11     Integer y = ex4.m(true);    // throws
12     }
13 }
```

Listing 4. Heap pollution without the heap

The example compiles. Method `m` in line 3 returns either a `String` or an `Integer` object. This is possible because the type variable `X` is erased at run time, and on the byte code level the method has the result type `Object`. Line 9 causes "heap pollution" because a raw type is used. In line 10 `m` returns an `Integer`, and the assignment to `x` works. Line 11 causes a `ClassCastException`, because a `String` is returned.

Of course, both examples produce "unchecked" warnings at run time. It is tempting to reason in the following manner: "Good programming practice will not create code that produces unchecked warning. Therefore, we exclude those programs from formal verification." However, that is not true. Except for very simple examples it is almost impossible to avoid unchecked warnings. For example, the Java Collection Framework produces 96 unchecked warnings. This means that a useful verification system must cope with them.

4 Java Verification in KIV

The KIV system has a calculus for the interactive verification of sequential Java programs. Before describing the specific features of generics concerning verification we give a short introduction to the KIV calculus.

The calculus is a sequent calculus for dynamic logic [12] based on algebraic specifications with a loose semantics. Dynamic logic extends predicate logic with two modal operators, box [.] and diamond ⟨ . ⟩. Box and diamond contain a context (a store) *st* and a Java program running in this context. The context contains the Java heap with the objects, and additional information about static fields, initialization of classes, and the execution state to model exceptions. It is specified algebraically. The class and interface declarations are part of a global environment. The intuitive meaning of $\langle st; \alpha\rangle\ \varphi$ is: with initial store *st* the Java statement α terminates, and afterwards φ holds. φ is again a formula of dynamic logic, i.e. it may contain boxes or diamonds. The meaning of $[st; \alpha]\ \varphi$ is: if α terminates then afterwards φ holds. A sequent $\varphi_1, \ldots, \varphi_m \vdash \psi_1, \ldots, \psi_n$ consists of two lists of formulas (often abbreviated by Γ and Δ) divided by $\vdash$ and is equivalent to the formula $\varphi_1 \wedge \ldots \wedge \varphi_m \rightarrow \psi_1 \vee \ldots \vee \psi_n$. The formulas $\varphi_1, \ldots \varphi_m$ can be thought of as preconditions, while one of $\psi_1, \ldots, \psi_n$ must be proved. A Hoare triple $\{\varphi\}\alpha\{\psi\}$ can be expressed as $\varphi \vdash [st;\alpha]\psi$ or $\varphi \vdash \langle st;\alpha\rangle\ \psi$ if termination is included. Java's type system is not built into the calculus, but rather specified algebraically. Logically, Java types in KIV are simple algebraic data types. This makes it trivial to incorporate parameterized types, type variables, wildcards, and bounds.

The calculus essentially has one rule for every Java expression and statement, plus some general rules. It works by symbolic execution of the Java program from its beginning to its end (i.e. computation of strongest postcondition). This means it follows the natural execution of the program, which is very important for interactive proofs. Nested expressions and blocks are flattened to a sequence of simple expressions and statements that can be executed directly. Obviously, this flattening must obey the evaluation order of Java. The result of a sub expression is 'stored' with an assignment to a local variable. This is shown in the following example.

$\Gamma \vdash \langle st;$ `x = m1(m2(y), m3());` $\rangle\ x = 5$

1. Here the arguments of the method call `m1` must be evaluated first. This is done by introducing a new local variable `x2`, and a new assignment to `x2`:
2. $\Gamma \vdash \langle st;$ `x2 = m2(y);` $\rangle\ \langle st;$ `x = m1(x2, m3());` $\rangle\ x = 5$
   The sub expression is replaced by `x2`. Since the argument to `m2` is a variable the method call can be evaluated. A proof rule for the method call basically replaces the method call by its body. If `m2` is declared as
   `int m2(int i) { return i + 1; }`
   the following goal is obtained:
3. $\Gamma, i = y \vdash \langle st;$ `return i + 1;` $\rangle\ \langle st;$ `target(x2)` $\rangle$
   $\langle st;$ `x = m1(x2, m3());` $\rangle\ x = 5$

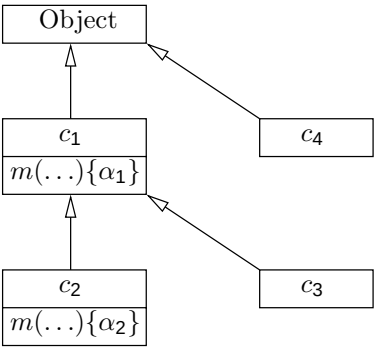
The formal parameter is bound to the actual parameter by the equation $i = y$. This is only possible if the actual argument is already fully evaluated, i.e. in KIV either a local variable or a literal.
The `target(x2)` statement (not part of Java, of course) acts as a catcher for the `return` statement, and assigns `x2` to the returned value. For `i + 1` another variable is introduced:

4. $\Gamma, i = y, i1 = i + 1 \vdash \langle st;$ `x2 = i1;` $\rangle \langle st;$ `x = m1(x2, m3());` $\rangle\ x = 5$
   The assignment becomes an equation, and then the next sub expression can be flattened. If variable conflicts occur, then a renaming will also take place:
5. $\Gamma, i = y, i1 = i + 1, x2 = i1 \vdash \langle st;$ `x = m1(x2, m3());` $\rangle\ x = 5$

And so on. After a finite number of applications of the flattening rule a list of assignments is returned where every sub expression is either a local variable or a literal. Then a rule for the main expression (e.g. a method call) or statement is applicable.

As a last example we show the rule for an instance method invocation `e.m(`e_1`,...,`e_n`);`. Figure 1 shows a class hierarchy where class c_1 contains a method declaration m with a body α_1 that is overridden in class c_2 with another body α_2.

The compiler determines at compile time a suitable method declaration, and the method call is annotated with the computed method signature, i.e. the method name m and the formal parameter types of the declaration. The argument types are needed because of overloading. Java verification in the KIV



1. $\ldots, \mathsf{mode}(st) \neq \mathsf{normal} \vdash \ldots$,
2. $\ldots, \mathsf{e} = \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal} \vdash$
   $\langle st; \mathsf{throw\ new}$ `NullPointerException();` $\rangle\ \ldots$,
3. $\ldots, \mathsf{e} \neq \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal} \vdash \mathit{classOf}(\mathsf{e}, st) \in \{\mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3\}$,
4. $\ldots, \mathsf{e} \neq \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal}, \mathit{classOf}(\mathsf{e}, st) \in \{\mathsf{c}_1, \mathsf{c}_3\},$
   $\mathsf{this}' = \mathsf{e}, \underline{\mathsf{z}} = \mathsf{e}_1, \ldots, \mathsf{e}_n \vdash \langle st; \ {}'_1 \rangle \langle st; \mathbf{target}(\mathsf{x}) \rangle\ \ldots$,
5. $\ldots, \mathsf{e} \neq \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal}, \mathit{classOf}(\mathsf{e}, st) \in \{\mathsf{c}_2\},$
   $\mathsf{this}' = \mathsf{e}, \underline{\mathsf{z}} = \mathsf{e}_1, \ldots, \mathsf{e}_n \vdash \langle st; \ {}'_2 \rangle \langle st; \mathbf{target}(\mathsf{x}) \rangle\ \ldots$,

$\vdash \langle st;$ `x = e.m(`e_1`,...,`e_n`);` $\rangle\ \ldots$,

Fig. 1. Example class hierarchy and rule for instance method invocation

system uses as input an annotated abstract syntax tree of the Java program and class declarations; so every method call is annotated with the computed method signature, too. The dynamic method lookup for an instance method then works as follows: The run-time class of the invoking object is determined, and the class declaration is searched for a method declaration with an identical signature as the annotated signature. If one is found this is the method to invoke. Otherwise the super class is searched and so on. The proof rule in the KIV system works in the same manner.

The proof rule is only applicable if e and the arguments $e_1, \ldots, e_n$ are fully evaluated, i.e. local variables or literals obtained by the flattening rule. This ensures that no side effects can occur. Premise 1 ensures that the method call is evaluated at all, and not skipped due to an exception. Premise 2 throws a `NullPointerException` if e is `null`. Premise 3 ensures that the type of e is either c_1, c_2, or c_3. If e is a reference to an object with type c_1 or c_3 then method $m(\ldots)\{\alpha_1\}$ is invoked (premise 4); if the type of e is c_2 then method $m(\ldots)\{\alpha_2\}$ is invoked (premise 5). In both premises the parameters $e_1, \ldots, e_n$ are bound to new variables $\underline{z}$, a new variable $this'$ is introduced for `this` and bound to e, in the method body the formal parameters are replaced with the new variables yielding α_i', and the new statement target(x) is added that will catch a return statement and bind x to the returned value.

Instead of expanding the method call, pre- and postconditions can be used (proof by contract). The calculus is well suited for interactive proofs because it follows the evaluation order of the Java statements and expressions as described in the Java language specification. Other proof rules modify or access the heap, but they are not relevant with respect to the formal verification of generics. We refer the reader to other literature [24] [25] [10].

5 Method Invocation

Method invocation is a situation where generics influence the run-time behavior of Java, for two reasons:

1. Dynamic method lookup is more complicated than before because of type variables and instantiation.
2. Heap pollution can cause an Exception during *method invocation conversion* (JLS 3, section 5.3).

Both items are described in turn.

Dynamic method lookup. In the presence of generics, it can be very complicated to compute the correct method signature that is associated with a method call at compile time. The description in JLS 3 is 32 pages long as compared to 9 pages in JLS 2. As mentioned in the previous section this annotation process is outside of KIV's formal framework. The dynamic method lookup also becomes more complicated if the types of the signature contain type variables. Listing 5 contains an example. (The heap pollution in the example can be ignored for the moment.)

```
1  interface I<T> { public int m(T x) ; }
2
3  class C<X> implements I<Integer> {
4     public int m(Integer x) { return 6; } // overrides m in I
5     public int m(String  x) { return 8; } // does not
6  }
7  public class Example5 {
8
9     public static void main(String[] args) {
10        I<Object> o = new C();          // raw type
11        System.out.println(o.m(5));     // prints 6
12        System.out.println(o.m("foo")); // throws
13    }
14 }
```

Listing 5. An example for dynamic method lookup

The program compiles. Since class `C` implements `I<Integer>` the method `m(Integer x)` in line 4 implements the method `m(T x)` in interface I. The two method calls in lines 11 and 12 are annotated with the method signature `m(T)` because the type of the invoking expression is `I`. At run time, it is not correct to search simply for a declaration with signature `m(T)`. Rather, it must be determined that the type variable `T` is instantiated with `Integer` in class `C`, so in class `C` a method with signature `m(Integer)` must be searched. In other classes that implement `I` the type variable may be instantiated with another class, or not at all. The same is possible for subclasses of `C` that override `m`.

The proof rule for dynamic method lookup in KIV now works as follows:

- ... The annotation for the method call must also include the name of the class or interface containing the suitable method declaration, not only the method signature. In the example this is interface `I`.
- ... Given a run-time class `C`, searching for the method declaration is done as follows: It is computed in which manner `C` inherits the method `m(T)` from `I`. This is done by following the chain of `implements` (or `extends`) clauses *downward* from `I` to `C`. In this process a substitution for the type variable `T` is computed. Because `C<X> implements I<Integer>`, and `I` has type variable `T`, the substitution is `T` ← `Integer`.

It may be noted that the description of the dynamic method lookup in JLS 3 (15.12.4.4) is identical to JLS 2, and the complication with instantiation of type variables is not mentioned. Another possibility for a proof rule would be to annotate every method declaration with those method signatures it overrides or implements. Then searching a class of a matching method signature would mean to search the annotations as well. This is comparable to using a dispatch table in a real implementation.

Method invocation conversion. JLS 3 states that the arguments of the method call are evaluated from left to right (15.12.4.2), then method invocation

conversion takes place (15.12.4.3), then an additional check is performed that may throw a `ClassCastException` (15.12.4.3). This is not correct, because after evaluation of one argument unboxing takes place (if necessary), and only then the next argument is evaluated. The evaluation order is significant, because unboxing can cause a `NullPointerException` if the reference to unbox is `null`. However, the additional checks (casts) are performed only after all arguments have been evaluated and converted.

Listing 5 is also an example why an additional cast may be necessary. In line 12 the method `m` is called with a string. The actual method to invoke depends on the run-time class of the invoking expression and the method signature as computed at compile time. In the example it is `m(T)` in interface `I` which is overridden by the method `m(Integer)` in class `C` as described above. Since a string cannot be converted to an Integer, a `ClassCastException` must be thrown. The method `m(String)` does not implement `m(T)` in `I`, and is not used even though the argument is a `String`. On the other hand, the incompatibility cannot be determined at compile time because another class `D` could implement `I<String>`, and for an invoker of class `D` a `String` argument is perfectly valid.

1. $, \mathsf{mode}(st) \neq \mathsf{normal} \vdash \quad ,$
2. $, \mathsf{o} = \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal} \vdash$
   $\langle st; \mathsf{throw\ new\ NullPointerException();} \rangle \quad ,$
3. $, \mathsf{o} \neq \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal} \vdash \mathit{classOf}(\mathsf{o}, st) = \mathsf{C},$
4. $, \mathsf{o} \neq \mathsf{null}, \mathsf{mode}(st) = \mathsf{normal}, \mathit{classOf}(\mathsf{o}, st) = \mathsf{C}, \mathsf{this} = \mathsf{o}, \vdash$
   $\langle st; \mathtt{x = (Integer)e;} \rangle \langle st; \mathbf{return}\ 6; \rangle \langle st; \mathbf{target} \rangle$

$\vdash \langle st; \mathtt{o.m(e);} \rangle$

Fig. 2. Method call for `o.m(e)`

The proof rule for a method call now works as follows (Fig. 2):

1. All arguments must be either local variables or literals, and autoboxing has been applied if necessary. This guarantees that no side effects occur.
2. For a given run-time class of the invoker the correct method declaration to invoke is determined as described above, in the example `m(Integer x) {...}`.
3. For every parameter of this declaration and actual argument:
   (a) If the static type of the actual argument is a subtype of the parameter type, then simply an equation *formal parameter variable* = *actual argument* is generated.
   (b) Otherwise an assignment to the formal parameter variable is generated, and the actual argument is cast to the formal parameter type. In the example the assignment `x = (Integer)e;` (premise 4 in Fig. 2) is generated.
4. The method call is replaced by its body, and all generated assignments are added before the body, in the example `x = (Integer)e; return 6;` (premise 4 in Fig. 2)

The Java compiler introduces a so called *bridge method* (JLS 3, 15.12.4.5) at compile time that is called at run time and performs the casts. In the example the bridge method is added to class `C`: `m(Object o){return m((Integer)o);}`. The proof rule has the identical behavior, but without additional transformations that are outside the formal framework.

6 Invalid Result Values

Section 15.5 of JLS 3 states rather cryptically

> *A run-time type error can occur only in these situations: [...] • In an implicit, compiler-generated cast introduced to ensure the validity of an operation on a non-reifiable type. [...]*

Because of heap pollution, the result of a method call (or a field access) can return a value that is not a subtype of (the erasure of) its static type (JLS 3, section 5.2). In these cases sometimes a `ClassCastException` is thrown. Listing 6 shows an example.

```
1  class Bag<E> {
2      public E content;
3      public Bag(E val) { content = val; }
4      public E get() { return content; }
5  }
6  public class Example6 {
7
8      static void mo(Object o) {
9      if (o instanceof String) System.out.println(1);
10     else System.out.println(2);
11     }
12     static void mi(Integer i) { System.out.println(i); }
13
14     public static void main(String[] args) {
15     Bag<Integer> bi = new Bag("foo"); // raw type
16     mo(bi.get());    // ok, prints 1
17     mi(bi.get());    // throws
18     }
19 }
```

Listing 6. An example for invalid return values

Line 15 creates heap pollution. In the following lines the static type of the variable `bi` is `Bag<Integer>` which means that the `content` field of `bi` should hold an Integer. However, it holds the string `"foo"`. The method call `bi.get()` returns this string.

Line 16 does not raise an exception. This is surprising since the result of the method call `bi.get()` (the string) is not a subtype of its static type `Integer`.

However, the result value is used in a context where only an `Object` is required. Therefore it is not necessary to throw a `ClassCastException` (JLS 3 5.2 and 5.3). The string is passed to the method `mo`, and the method prints `1`. Line 17 throws a `ClassCastException` because the result is used in a context that requires an `Integer`. This is a different situation than the method calls in listing 5 where bridge methods are used. The Java compiler guarantees this behavior by simply inserting casts into the byte code. The byte code always contains the casts, even when no "unchecked" warnings are issued because the heap pollution could stem from already compiled code.

What are the implications for the formal verification? One possibility would be to modify the source code during parsing and annotation. However, this would be outside the formal framework, and in interactive proofs it is desirable to be as close to the source code as possible. Therefore, these implicit (or "unchecked") casts are included in the calculus:

1. A new proof rule is introduced for "unchecked" casts.
2. The proof rules for field access and method call are applicable only if no "unchecked" casts are required.
3. The flattening rule is modified to keep track of the type required by the context.

In the same manner, a proof rule for autoboxing has been included in the calculus. The new proof rule for an "unchecked" cast is applicable for a method call `x = e.m(`$e_1,\ldots,e_n$`);` (and similarly for a field access `x = e.f`) iff

- …`e` and $e_1,\ldots,e_n$ are local variables (possibly introduced by flattening), and
- …the declared return type of `m` (in constrast to the computed result type) is a type variable `T`, and
- …the erasure of `T` is not a subtype of the static type of `x`. For a type variable or wildcard without bounds this means that the type of `x` is neither `Object` nor an unbounded type variable. For bounded type variables `T extends C` or wildcards `? extends C` the erasure is `C`.

Then the proof rule simply introduces a cast to the static type of `x`. Fig. 3 shows the rules. Conversely, the proof rules for method calls and field access are only applicable if the "unchecked" rule is not applicable. The flattening rule guarantees that the "unchecked" rule is not applied a second time.

Line 16 in listing 6 contains an example that the context is used to decide whether a cast is added or not. Since the flattening rule transforms `mo(bi.get())` into `x = bi.get(); mo(x);` the static type of `x` must be `Object` (the context type), not `Integer` (the computed result type of `bi.get()`). The modified flattening rule must use the context type where necessary.

$$\frac{\vdash \langle st; \texttt{x = (Ty)e.m(}e_1,\ldots,e_n\texttt{);}\rangle}{\vdash \langle st; \texttt{x = e.m(}e_1,\ldots,e_n\texttt{);}\rangle} \qquad \frac{\vdash \langle st; \texttt{x = (Ty)e.f;}\rangle}{\vdash \langle st; \texttt{x = e.f;}\rangle}$$

Fig. 3. The new "unchecked" proof rules

This finishes the description of implicit casts for result values. JLS 3 does not mention it, but the Java compilers treat an array access for an array of type `E[]` with `E` a type variable in the same manner as a field access or method call. This means an array access `a[i]` can also cause a `ClassCastException`.

7 Verification with Generics in KIV

The examples in the paper can be verified in KIV with the modified calculus. It turns out that generics have almost no effect on the specification and verification methodology used in KIV which is based on algebraic specifications and the proof of functional properties. This is illustrated with the following example: We want to prove that the `sum` method from listing 1 correctly computes the sum of some integers, $\sum$ *ints*. *ints* is an algebraically specified list of integers, and $\sum$ an algebraic function. The proof can only succeed if we know (or assume) that the input `li` to the `sum` method represents this list of integers, *isList(ints, li, st)*. *isList* is a predicate that "looks" into the heap *st*. The goal to prove is therefore

$$\mathit{isList(ints,\ li,\ st)}, \ldots \vdash \langle \mathrm{st};\ \texttt{i = x.sum(li);}\ \rangle\ \mathrm{i} = \sum \mathit{ints}$$

This property is not trivial: The iterator (used by the enhanced `for` loop as in listing 2) must be implemented correctly; the `hasNext` method must eventually return `false` (for termination); the `next` method must successively return `Integer` objects (not `null`, and not other objects because of heap pollution) that represent the integers in *ints*; no integer over- or underflow may occur.

Essentially this means we need a very precise knowledge about the data structure represented by `li` in the heap – independent of whether generic types are used or not. Of course, we do not have this knowledge because it is not known which `List` implementation is used. Therefore, we must make assumptions about the methods `iterator`, `hasNext`, and `next`, and the predicate *isList* (proof by contract). Then, for a given `List` implementation we can specify *isList* and prove the assumptions. E.g., the assumption for `next` is:

$$\begin{aligned} &\mathit{isIterator(ints,\ iter,\ st)},\ \mathit{ints} \neq [], \ldots \\ \vdash\ &\langle \mathrm{st};\ \texttt{o = iter.next();}\rangle\ (\quad \mathit{isIntegerObject(ints.first,\ o,\ st)} \\ &\qquad\qquad \wedge\ \mathit{isIterator(ints.rest,\ iter,\ st)} \wedge \ldots) \end{aligned}$$

Two auxiliary predicates are used: *isIterator(ints, iter, st)* is true if *iter* is an `Iterator` object that represents the integers *ints*. Then the `next` method returns an `Integer` object that represents the first value of *ints* (*isIntegerObject(ints.first, o, st)*, and by side effect the iterator has been modified so that it represents the *remaining* integers *ints.rest* (*isIterator(ints.rest, iter, st)*). Again, this is completely independent of generic types.

For a given `List` implementation (for example, the `ArrayList` of the Java collection framework) the predicates must be specified. This requires a look into the actual implementation (how the `next` method accesses the element to return etc.), and a class invariant about the iterator (the `cursor` field used by the iterator is not greater than the `size` field of the `ArrayList` which in turn

is less than the length of the array holding the elements etc.). And it must be specified that all list elements are indeed `Integer` objects.

8 Conclusion

We have extended KIV's Java calculus to support the formal verification of Java generics. It may be mentioned that JLS 3 is sometimes cryptic, unclear, and possibly wrong for generics. The main design decision was not to modify the source code to verify, but to include the run-time effects of generics dynamically into the appropriate proof rules. The effects are: First, dynamic method lookup is more complicated than before (Listing 5); second, because of the possibility of heap pollution (Sect. 3) a method invocation may require additional checks (Listing 5); third, because of heap pollution the result of a method call or a field access can cause a `ClassCastException` (Listing 6). This has the subtle consequence that in addition to the static type of an expression the context (the expected or required type) of the expression becomes important. It is now possible to verify programs with "unchecked" warnings, and in the presence of heap pollution (for example, the programs in this paper).

Experience shows that the effects of generics on proofs in KIV are small because the additional casts cause little overhead, and because KIV's methodology relies on algebraic properties where static types play a negligible role. A possible direction for future work is a formal specification of a type correct Java program with generics, and a proof of type soundness.

References

1. Alves-Foss, J. (ed.): Formal Syntax and Semantics of Java. LNCS, vol. 1523. Springer, Heidelberg (1999)
2. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783. Springer, Heidelberg (2000)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362. Springer, Heidelberg (2005)
4. Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., Requet, A.: JACK: a tool for validation of security and behaviour of Java applications. In: FMCO: Proceedings of 5th International Symposium on Formal Methods for Components and Objects, vol. 4709. Springer, Heidelberg (2007)
5. Beckert, B., Hähnle, R., Schmitt, P. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3) (2005)
7. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Huisman, M. (eds.) CASSIS 2004. LNCS, vol. 3362. Springer, Heidelberg (2005)
8. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley, Reading (1996)

9. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java (tm) Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
10. Grandy, H., Bertossi, R., Stenzel, K., Reif, W.: ASN1-light: A Verified Message Encoding for Security Protocols. In: Software Engineering and Formal Methods, SEFM. IEEE Press, Los Alamitos (2007)
11. Grandy, H., Bischof, M., Schellhorn, G., Reif, W., Stenzel, K.: Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In: FM 2008: 15th Int. Symposium on Formal Methods, vol. 5014. Springer, Heidelberg (2008)
12. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
13. Jacobs, B., Marché, C., Rauch, N.: Formal verification of a commercial smart card applet with multiple tools. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116. Springer, Heidelberg (2004)
14. Jacobs, B., Poll, E.: Java Program Verification at Nijmegen: Developments and Perspective. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 134–153. Springer, Heidelberg (2004)
15. JML home page, `http://www.jmlspecs.org/`
16. Joy, B., Steele, G., Gosling, J., Bracha, G.: The Java (tm) Language Specification, 2nd edn. Addison-Wesley, Reading (2000)
17. Kiniry, J.: Recent advances in extended static checking. Technical report, KeY Symposium 2007 (2007), `http://www.key-project.org/keysymposium07/slides/kiniry-esc.pdf`
18. KIV homepage, `http://www.informatik.uni-augsburg.de/swt/kiv`
19. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
20. Leavens, G.T., Kiniry, J., Poll, E.: A JML tutorial. Technical report, CAV 2007 Tutorial (2007), `http://cav2007.org/Docs/Leavens.JML.ps4.pdf`
21. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/Javacard programs annotated in JML. Journal of Logic and Algebraic Programming 58(1-2) (2004)
22. MasterCard International Inc. Mondex, `http://www.mondex.com`
23. Schmitt, P.H., Tonin, I.: Verifying the Mondex case study. In: Software Engineering and Formal Methods, SEFM. IEEE Press, Los Alamitos (2007)
24. Stenzel, K.: A formally verified calculus for full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)
25. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik (2005)
26. Sun Microsystems Inc. Java Card 2.2 Specification (2002), `http://java.sun.com/products/javacard/`
27. Ulbrich, M.: Software verification for Java 5. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (in English, 2007)