Integrating ADTs in KeY and their Application to History-based Reasoning

Jinting Bian[1][0000−0001−5003−598X], Hans-Dieter A. Hiep[1][0000−0001−9677−6644], Frank S. de Boer[1], and Stijn de Gouw[2][0000−0003−2964−6844]

[1] CWI, Science Park 123, 1098 XG Amsterdam, The Netherlands
{j.bian,hdh,frb}@cwi.nl
[2] Open University, The Netherlands
sdg@ou.nl

Abstract. We discuss integrating abstract data types (ADTs) in the KeY theorem prover by a new approach to model data types using Isabelle/HOL as an interactive back-end, and translate Isabelle theorems to user-defined taclets in KeY. As a case study of this new approach, we reason about Java's `Collection` interface using histories, and we prove the correctness of several clients that operate on multiple objects, thereby significantly improving the state-of-the-art of history-based reasoning.
Open Science. Includes video material [4] and a source code artifact [5].

Keywords: Formal verification · abstract data type · program correctness · Java Collection Framework · KeY · Isabelle/HOL.

1 Introduction

The overall aim of this paper is to put formal methods to work by the verification of software libraries which are the building blocks of millions of programs, and which run on the devices of billions of users every day. Our research agenda is to verify heavily used software libraries, such as the Java Collection Framework, since the verification effort weighs up against the potential impact of errors.

In [11] the use of formal methods led to the discovery of a major flaw in the design of TimSort, the default sorting method in many widely used programming languages, e.g. Java and Python, and platforms like Android. An improved version of TimSort was proven correct with the state-of-the-art theorem prover KeY [1]. The correctness proof of [11] convincingly illustrates the importance and potential of formal methods as a means of rigorously validating widely used software and improving it. In [17] this line of research has been further successfully extended by the verification of the basic methods of (a corrected version of) the `LinkedList` implementation of the Java Collection Framework, laying bare an integer overflow bug, using again the KeY theorem prover.

KeY is tailored to the verification of Java programs. In a proof calculus based on sequents, it symbolically executes fragments of the loaded program which are represented by modal operators of the underlying dynamic logic. KeY is based

on the Java Modeling Language [7], JML for short, for the specification of class invariants, method contracts, and loop invariants. This specification language is intrinsically *state-based* and as such is not directly suitable for the specification of state-hiding *interfaces*. As such, the work described in [17] excludes the specification and verification of the `Collection#addAll(Collection)` method implemented by `LinkedList`, which adds all the elements of the collection that is passed as parameter. The difficulty lies in giving a specification of the interface, that works for all possible implementations (including `LinkedList` itself).

In recent previous work [16] the concept of a *history* as a sequence of method calls and returns has been introduced as a general methodology for specifying interfaces. As proof-of-concept, using the KeY theorem prover, this methodology has been applied to the core methods of Java's `Collection` interface and uses an encoding of histories as Java objects on the heap.

That encoding, however, made use of pure methods in specification and thus required extensive use of so-called *accessibility* clauses, which express the set of locations on the heap that a method may access during its execution. These accessibility clauses must be verified (with KeY). Furthermore, for recursively defined pure methods we also need to show *termination* and *determinism* [23]. Essentially, the associated verification conditions boil down to verifying that the method under consideration computes the same value starting in two heaps that are different except for the locations stated in the accessibility clause. To that end one has to symbolically execute the method more than once (in the two different heaps) and relate the outcome of the method starting in different heaps to one another. After such proof effort, accessibility clauses of pure methods can be used by the application of *dependency contracts*, that are used to establish that the outcome of a pure method in two heaps is the same if one heap is obtained from the other by assignments outside of the declared accessible locations.

The degree of automation in the proof search strategy with respect to pure methods, accessibility clauses and dependency contracts turned out to be rather limited in KeY. So, while the methodology works in principle, in practice, for advanced use, the pure methods were a source of large overhead and complexity in the proof effort.

This paper avoids this complexity by instead modeling histories as *Abstract Data Types*, ADTs for short. Elements of abstract data types are not present on the heap, avoiding the need to use dependency contracts for proving that heap modifications affect their properties. Since KeY has limited support for user-defined abstract data types, we introduce a general *workflow* which integrates the domain-specific theorem prover KeY and the general-purpose theorem prover Isabelle/HOL [25] for the specification of ADTs. We apply and discuss the application of this workflow to the Java `Collection` interface, including how we have now been able to specify and verify the `addAll` method. Furthermore, reasoning about advanced use cases involving multiple instances of the same interface is possible: we also have verified a method that compares *two* collections.

We assume the reader is somewhat familiar with KeY and/or Isabelle/HOL.

Related work The Java Collection Framework is among the most heavily-used software libraries in practice [8], and various case studies have focused on verifying parts of that framework [3,18,19]. Knüppel et al. [20] specified and verified several classes of the Java Collection Framework with standard JML state-based assertions, and found that specification was one of the main bottlenecks. One source of the complexity concerns framing: specifying and reasoning about properties and locations that do *not* change.

The idea presented in this paper of integrating Isabelle/HOL and KeY arises out of the need for user-defined data types usable within specifications. Other tools, such as Dafny [22] and Why3 [13], support user-defined data types in the specification language, contrary to JML as it is implemented by KeY. However, the former tools are not suitable to verify Java programs: for that, as far as the authors know, only KeY is suitable due to its modeling of the many programming features of the Java language present in real-world programs.

In state-based approaches, including the dynamic frames used in [20], frames inherently heavily depend on the chosen representation, i.e. at some point, the concrete fields that are touched or changed must be made explicit. The same holds for separation logic [26] approaches for Java [12]. Since interfaces do not have a concrete state-based representation, *a priori* specification of frames is not possible. Instead, for each class that implements the interface, further specifications must be provided to name the concrete fields. One can abstract from these concrete fields by using a footprint model method. However, the footprint model method itself also requires a dynamic frame, leading to recursion in dependency contracts [2]. Moreover, any specification that mentions (abstract or concrete) fields can be problematic for clients of the classes, since the representation is typically hidden from them (by means of an interface), which raises the question: how to verify clients that make use of interfaces?

The history-based approach in this paper (contrary to [16]) avoids framing entirely: there is no reference to an underlying state, as the complete behavior of an interface is captured by its history. Additionally, since we model such histories as elements of an ADT *separate from the sorts used by Java* in this paper, histories can not be touched by Java programs under verification themselves, and so we never have to use dependency contracts for reasoning about properties of histories. This allows us to avoid the verification bottleneck that arises in the approach of [16], which used an encoding of histories as ordinary Java objects living on the heap.

2 Integrating Abstract Data Types in KeY

Abstract data types were introduced in 1974 by Barbara Liskov and Stephen Zilles [24] to ease the programming task: instead of directly programming with concrete data representations, programmers would use a suitable abstraction that instead exposes an interface, thereby hiding the implementation details of a data type. In most programming languages, such interfaces only fix the signature of an abstract data type (e.g. Java's interface or Haskell's typeclass). Further

research has lead to many approaches for specifying abstract data types, e.g. ranging from simple equational specifications, to axiomatizations in predicate logic. See for an extensive treatment of the subject the textbook [27].

In the context of our work, we need to distinguish the two levels in which abstract data types can appear: at the programming level, and at the specification level. In fact, Java supports abstract data types by means of its interfaces, and for example the Java Collection Framework provides many abstractions to ease the programming task. The specification language JML does support reasoning about the instances of such interfaces, but does not allow user-defined abstract data types on the specification level only The reason is that JML is designed to be "easier for programmers to learn and less intimidating than languages that use special-purpose mathematical notations" [21]. There are extensions to JML, e.g. [9], but this has not been implemented in KeY.

KeY extends JML in an important way: a number of built-in abstract data types at the specification level are provided in a type hierarchy [1, Section 2.4.1]. There is the abstract type of *sequences* that consists of finite sequences of arbitrary elements. Further, KeY provides the abstract data type of *integers* that comprises the mathematical integers (and not the integers modulo finite storage, as used in the Java language) to interpret JML's `\bigint`. Elements of these abstract types are not accessible by Java programs, and are not stored on the heap. It is possible to reason about elements of such abstract data types, since the KeY theorem prover allows to define their *theories* implemented by inference rules for deducing true statements involving these elements.

For introducing user-defined abstract data types, KeY does allow the specification of abstract data types by adding new inference rules: but it provides no guarantee that such user-defined theory is *consistent*. Thus, a small error in a user-defined abstract data type specification could lead to unsound proofs.

In contrast, Isabelle/HOL (Isabelle instantiated with Church's type theory) includes a definitional package for data types [6]. The definition mechanism provides so-called *freely generated* data types: the user provides some signature consisting of constants and function symbols and their types, and the system automatically derives (rather than postulate) characteristic theorems. Under the hood, each data type definition is associated to a Bounded Natural Functor (BNF) that admits a non-trivial initial algebra [28], but for our purposes we may simply trust that the system maintains consistency.

Our approach is to integrate the Isabelle/HOL theorem prover as an interactive back-end for KeY: this allows us to make use of the advanced capabilities of Isabelle to define data types, define functions, and prove general properties thereof, all while preserving consistency. We can then import these results from Isabelle in KeY to use them for proving the correctness of Java programs. The soundness of our approach crucially relies on the consistency of the imported Isabelle theory, and the soundness of the translation from statements in Isabelle/HOL to KeY taclets.

The overall approach can be summarized by a workflow diagram, see Figure 1. Essentially, we will be following three steps:

1. We define data types and functions in Isabelle/HOL to logically model domain-specific knowledge of the Java program that we want to verify. These definitions can not refer to Java types directly, but can instead be defined using polymorphic type parameters.
2. We import the signature (sorts, function symbols) in KeY and write specifications of the Java program in JML. These specifications can make use of the imported function symbols by using a KeY-specific extension of JML.
3. We use the KeY system to perform symbolic execution of the Java program. This leads to proof obligations in which the imported symbols are uninterpreted, meaning that one is limited in reasoning about them in KeY. We then formulate a lemma that captures the expected property and prove it in Isabelle/HOL. If we succeed, the lemma is added to KeY by translating it back as an inference rule called a *taclet*. This step will be repeated many times, because we can not find all needed lemmas at once.

Below we give more detail on each of these main steps.

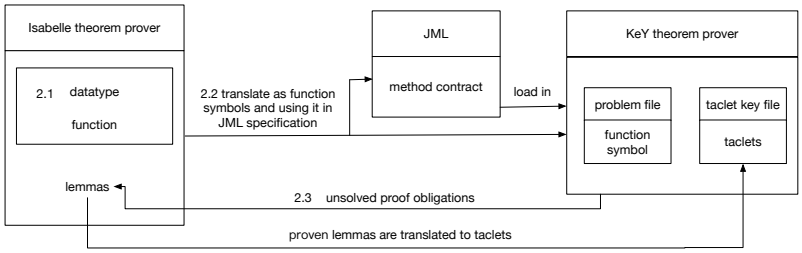

Fig. 1. The workflow of integrating ADTs in KeY.

Step 1. Formalizing ADTs in Isabelle/HOL One defines data types and recursive functions in Isabelle/HOL in the usual manner: using the **datatype** and **fun** commands. There are a number of caveats when working in Isabelle, to ensure a smooth transfer of the theory to KeY.

- For data types that contain nested Java objects, we have to work around the limitation that the Java types are not available in Isabelle. This is a design choice of our approach, to keep Isabelle pure: we can instead introduce a polymorphic type parameter. Below we show how in our translation back to KeY, we put back the original types by instantiating the polymorphic type parameters by Java types which are available in KeY.
- Isabelle/HOL allows higher-order definitions, whereas the dynamic logic of KeY is first-order. Thus, for function symbols that we wish to import, we limit ourselves to first-order type signatures.

As a simple example, an element of the data type definition

$$\textbf{datatype}\ \alpha\ option = None \mid Some(\alpha)$$

in Isabelle/HOL is either 'nothing' or an element of type α. This introduces also a constant *None* : α *option* and a unary function *Some* : $\alpha \Rightarrow \alpha$ *option*. One can then define functions recursively over the structure of a user-defined data type. This is illustrated in the next section.

Step 2. Using ADTs in JML specifications To declare new sorts and functions, KeY uses an extensible formalism called *taclets* [14,15]. Taclets in KeY are stored in plain-text files alongside Java sources and each contains the following sections:

- We introduce for each new type instantiation (where the type parameters are replaced by sorts) a corresponding logical sort with the desired name of the ADT. The syntactical notation uses a block section named `\sorts`.
- We declare the signatures of each function in a section named `\functions`. A function signature consists of the number and types of its parameters, and its return type. We erase the polymorphic type parameters, by replacing them by their instantiated sorts.
- We add axioms to specify properties of functions in a section named `\axioms`.

Listing 1 shows how to represent the above data type α *option* as a taclet. We have instantiated the type parameter α with the Java `Object` sort.

```
\sorts { option; }
\functions { option Some(java.lang.Object);
             option None; ... }
\axioms { ... }
```

Listing 1. Declaring sorts and function symbols for new ADTs with KeY.

The new functions can then be used in JML specifications (such as method contracts and class invariants) by prefixing their name with "`\dl_`". For example, the function symbol *None* can be referred to in a JML contract by writing it as `\dl_None`. To allow using the functions in JML specifications, axioms are not (yet) needed. Therefore, in step two of our workflow we do not specify any axioms. We describe adding axioms in more detail below, in step three.

Step 3. Translating Isabelle theorems to KeY taclets We now focus on using the new ADTs in proofs of Java programs with KeY. When one starts proving that a Java program satisfies its JML specification and that specification contains function symbols declared as above, KeY treats them as uninterpreted symbols (with unknown behavior, other than their signatures). Typically this is insufficient to complete the proof: one needs specific properties that follow from the underlying definition in Isabelle/HOL.

We can "import" such properties about the behavior of user-defined functions into KeY by defining inference rules in the `axioms` section of taclets. The `axioms` are intended for basic facts that KeY can not derive from any other inference rules. We leverage Isabelle/HOL to prove the soundness and consistency of the imported axioms. In essence, this provides a way to use Isabelle/HOL as an interactive back-end to KeY. Our workflow supports a lazy approach that

minimizes the amount of work. We only add axioms about functions *when they are necessary*, i.e., when we are stuck in a proof situation that requires more knowledge of the function behavior.

Let us consider a concrete example that illustrates the above concepts. Suppose we have a proof obligation in KeY in which $Some(x) = Some(y)$ appears as an assumption (it occurs as an antecedent of an open goal sequent), and has $x = y$ as one of its conclusions (it appears as a succedent of the sequent). Without any axioms, KeY can not proceed in proving the goal. We thus formulate in Isabelle/HOL, abstracting from the particular sorts as they appear in KeY, the following lemma[3]

$$\textbf{lemma}\ \mathit{Some\_injective} :: \mathit{Some}(a) = \mathit{Some}(b) \leftrightarrow a = b$$

which we easily verified using one of the characteristic theorems of the data type.

Our next objective is to import this lemma to KeY to make it available during the proving process. We do this by formulating the lemma as a taclet in the block `axioms`, see Listing 2.

```
\axioms {
  Some_injective {
    \schemaVar \term java.lang.Object o1, o2;
    \find(Some(o1) = Some(o2)) \replacewith(o1 = o2)
  }; }
```

Listing 2. Adding a taclet to KeY that expresses injectivity of the function *Some*.

This taclet states that the name of the inference rule is `Some_injective`. The keyword `find` states to which expressions or formulae the rule can be applied (on either side of the sequent). Two placeholder symbols called schema variables are used to stand for, in this case, the parameters of the `Some` function. These placeholders are instantiated when the inference rule is applied in a concrete proof. The `replacewith` clause states that the expression or formula in the find clause to which the rules is applied, is replaced after application by a new formula (in this case `o1 = o2`) in the resulting sequent. The `heuristics` clause indicates which of KeY's internal automated proof search strategies may use the rule.

One may also express side conditions on other formulas that need to be present in the sequent with the clause `assumes`, and variable conditions with the clause `varcond` (not shown in this example). A full exposition of the taclet language is out of scope of this article, we instead refer to [1, Section 4.3].

3 Case study: History-based Reasoning about `Collection`

In our case study, we apply histories to the specification of the `Collection` interface and verify clients of that interface with respect to this specification. Histories are defined formally below as sequences of events, and a single event

[3] A taclet which uses `find` and `replacewith` corresponds to a bi-implcation in Isabelle, since the term can appear on both sides of the sequent in KeY.

models a method call and its return. For each collection one may create new iterator objects as a view of its contents. Iterators require special treatment because their behavior relies on the history of other objects, such as the enclosing collection that *owns* the iterator. We therefore model iterators as sub-objects so that their history is recorded by the associated owning collection.

We thus introduce (in Isabelle/HOL) the following event data type:

$$\begin{aligned}\textbf{datatype}\ (\alpha,\beta)\ \mathit{event} = {} & \mathit{Add}(\alpha,\textbf{bool}) \mid \mathit{AddAll}(\alpha\ \mathit{elemlist},\textbf{bool}) \mid \\ & \mathit{Remove}(\alpha,\textbf{bool}) \mid \mathit{Contains}(\alpha,\textbf{bool}) \mid \mathit{IsEmpty}(\textbf{bool}) \mid \\ & \mathit{Iterator}(\beta) \mid \mathit{IteratorNext}(\beta,\alpha) \mid \mathit{IteratorRemove}(\beta) \mid ...\end{aligned}$$

We focus here on the most important events only. The type parameters α and β correspond to (type abstractions of) the Java types `Object` and `Iterator`, respectively. We now introduce histories as sequences of events:

$$\textbf{datatype}\ (\alpha,\beta)\ \mathit{history} = \mathit{Nil} \mid \mathit{Cons}((\alpha,\beta)\ \mathit{event},(\alpha,\beta)\ \mathit{history})$$

As above, the type parameters α and β correspond to (type abstractions of) the Java types `Object` and `Iterator`.

Listing 3 illustrates the use of histories modeled as abstract data types and their functions in the interface specification of part of the **ensures** clause of the `addAll` method. It relates the multiplicities of elements of the argument collection with that of the receiving collection. Here, `\dl_multiset(c.history(),o))` and `\dl_multiset(history(),o)`, defined below, denote the multiplicity of an element o in the argument and receiving collection, respectively. The list *el* of type (where we overload *Nil* and *Cons*)

$$\textbf{datatype}\ \alpha\ \mathit{elemlist} = \mathit{Nil} \mid \mathit{Cons}(\alpha \times \textbf{bool},\alpha\ \mathit{elemlist})$$

associates a status flag to each occurrence of an element of the argument collection. This flag indicates whether the *receiving* collection's implementation actually *does* add the supplied element (e.g., a `Set` filters out duplicate objects but a `List` does not). Consequently, the multiplicity of the elements of the receiving collection is updated by how many times the object is actually added, denoted by `\dl_multisetEl(el,o)` (also defined below). The existential quantification of this list allows both for abstraction from the particular enumeration order of the argument collection and the implementation of the receiving collection as specified by the association of the Boolean values.

```
/*@ ...
  @ ensures (\exists elemlist el;
        (\forall Object o;
          \dl_occurs(el,o) == \dl_multiset(c.history(),o) &&
          \dl_multiset(history(),o) ==
          \dl_multiset(\old(history()),o) + \dl_multisetEl(el,o)));
  @*/
boolean addAll(Collection c);
```

Listing 3. The use of *multiset* and *elemlist* in the specification of `addAll`.

Using the data types above, we can now recursively define the crucial function *multiset*, that computes the multiplicity of an object given a particular history. Intuitively it represents the 'contents' of a collection at a particular instant.

$$
\begin{aligned}
&\textbf{fun } \mathit{multiset} : (\alpha, \beta)\ \mathit{history} \times \alpha \Rightarrow \textbf{int} \\
&\quad \mathit{multiset}(\mathit{Nil}, x) = 0 \\
&\quad \mathit{multiset}(\mathit{Cons}(\mathit{Add}(y, b), h), x) = \mathit{multiset}(h, x) + (x = y \wedge b\ ?\ 1 : 0) \\
&\quad \mathit{multiset}(\mathit{Cons}(\mathit{AddAll}(xs, b), h), x) = \mathit{multiset}(h, x) + \mathit{multisetEl}(xs, x) \\
&\quad \mathit{multiset}(\mathit{Cons}(\mathit{Remove}(y, b), h), x) = \mathit{multiset}(h, x) - (x = y \wedge b\ ?\ 1 : 0) \\
&\quad \mathit{multiset}(\mathit{Cons}(\mathit{IteratorRemove}(i), h), x) = \\
&\qquad\qquad \mathit{multiset}(h, x) - (\mathit{last}(h, i) = \mathit{Some}(x)\ ?\ 1 : 0) \\
&\quad \mathit{multiset}(\mathit{Cons}(e, h), x) = \mathit{multiset}(h, x)
\end{aligned}
$$

The function *multisetEl* is defined as follows.

$$
\begin{aligned}
&\textbf{fun } \mathit{multisetEl} : \alpha\ \mathit{elemlist} \times \alpha \Rightarrow \textbf{int} \\
&\quad \mathit{multisetEl}(\mathit{Nil}, x) = 0 \\
&\quad \mathit{multisetEl}(\mathit{Cons}((y, b), t), x) = \mathit{multisetEl}(t, x) + (x = y \wedge b\ ?\ 1 : 0)
\end{aligned}
$$

For the full Isabelle theory of our case study (including the definition for *last*) we refer the reader to the artifact accompanying this paper [5]. This artifact includes the translation of the Isabelle theory to a signature that can be loaded in KeY (version 2.8.0), so that its function symbols are available in the JML specifications we gave for `Collection`. It also includes the taclets we imported from Isabelle, that we used to close the proof obligations generated by KeY.

3.1 Significant improvement in proof effort

Using ADTs instead of encoding histories as Java objects results in significantly lower effort in defining functions for use in contracts and giving correctness proofs. This can be best seen by revisiting an example of our previous work [16] and comparing it to the proof effort required in the new approach using ADTs.

```
/*@ ...
  @ ensures (\forall Object o1; \dl_multiset(x.history(),o1) ==
       \dl_multiset(\old(x.history()),o1)); @*/
public static Object add_remove(Collection x, Object y) {
    if (x.add(y)) {
        x.remove(y);
    }
    return y;
}
```

Listing 4. Adding an object and if successful removing it again, leaves the contents of a `Collection` the same. See Table 1 for proof statistics and a link to the video.

The client code and its contract is given Listing 4, which has the same contract as in previous work, except we now use the imported functions we have defined in Isabelle instead of using pure methods and their dependency contracts.

In both the previous and this work, we specify the behavior of the client by ensuring that the 'contents' of the collection remains unmodified: we do so in terms of the multiset of the old history and the new history (after the `add_remove` method). During verification we make use of the contracts of methods `add(Object)` and `remove(Object)`. These contracts specify their method behavior also in terms of the old and new history, relative to each call. Let h be the old history (before the call) and h' be the new history (after the call). Let y be the argument, the `remove` method contract specifies that $multiset(h', y) = multiset(h, y) - 1$ if the return value was **true**, and $multiset(h', y) = multiset(h, y)$ otherwise. Further, it ensures the return value is **true** if $multiset(h, y) > 0$. Also, $multiset(h', x) = multiset(h, x)$ holds for any object $x \neq y$. In similar terms, a contract is given for `add` that specifies that the multiplicity of the argument is increased by one, in case **true** is returned, and that regardless of the return value the multiplicity of the argument is positive after `add`.

We need to show that the multiplicity of the object y after the `add` method and the `remove` method is the same as before executing both methods. At this point, we can see a clear difference in verification effort required between the two approaches. In the previous approach, multiplicities are computed by a pure Java method `Multiset` that operates on an encoding of the history that lives on the heap. Since Java methods may diverge or use non-deterministic features, we need to show that the pure method behaves as a function: it terminates and is deterministic. Moreover, since we deal with effects of the heap, we also need to show that the computation of this pure method is not affected by calls of `add` or `remove`, which requires the use of a accessibility clause of the multiset method.

To make this explicit, Listing 5 shows a concrete example of a proof obligation from KeY that arose in the previous approach.

```
...
History.Multiset(h,y)@heap2 + 1 = History.Multiset(h,y)@heap1,
History.Multiset(h,y)@heap1 = History.Multiset(h,y)@heap + 1,
...
==>
History.Multiset(h,y)@heap2 = History.Multiset(h@heap,y)@heap2
```

Listing 5. Simplified proof obligation with histories as Java objects showing evaluation of the multiset function as a pure (Java) method in various heaps.

Informally, the proof obligation states that we must establish that the multiplicity of `y` after adding and removing object `y` (resulting in the heap named `heap2`) is equal to the multiplicity of `y` before both methods were executed (in the heap named `heap`). So we have to perform proof steps relating the result/behavior of the multiset method in different heaps. In practice, heap terms may grow very large (i.e. in a different, previous case study [10] we encountered heap terms that were several pages long) which further complicates reasoning.

By contrast, in the new ADT approach of this paper, we model multiset as a function without any dependency on the heap, and so we do not have to perform proof steps to relate the behavior of multiset in different heaps (the interpretation of multiset is fixed and does not change if the heap is modified). While the arguments of multiset may still depend on the heap (such as the history associated to an interface that lives on the heap), when we evaluate the argument to a particular value (such as an element of the history ADT) the behavior of the multiset function when given such values does not depend on the heap.[4] Moreover, by defining the function in Isabelle/HOL, we make use of its facilities to show that the function is well-defined (terminating and deterministic). In our case-study this is done fully automatically: contrary to the proofs of the same property in KeY in the previous approach. Thus, the new approach significantly reduces the total verification effort required.

The proof statistics of verifying this example are shown in Table 1. The previous approach (encoding histories as Java objects [16]) is marked, and includes the verification that the multiset pure method is terminating and deterministic and satisfies its equational specification (first row). This effort is completely eliminated in the new approach, since it can be done automatically using Isabelle/HOL. Furthermore, comparing the verification of the `add_remove` method in both approaches, it can be immediately seen that we no longer have to apply any dependency contract in the new approach. This improves two important factors of the total proof effort. Moreover, the previous approach was studied in the context of a simpler definition for histories (without modeling the `addAll` event), thus favoring the new approach even more. The non-marked rows, i.e. the new approach, are part of the accompanying artifact [5], and video files (no sound!) show a recording of the interactive proof sessions [4].

Name	**Nodes**	**Branches**	**I.step**	**Q.inst**	**Contract**	**Dep.**	**Loop inv.**	**Time**
Multiset†	54,857	1,053	52	476	39	0	0	72 min
add_remove†	3,936	79	44	5	2	23	0	11 min
add_remove	2,606	14	4	7	2	0	0	1 min
iterate_only†	8,549	58	53	0	4	12	1	15 min
iterate_only	3,447	15	0	2	3	0	1	1 min
iter_remove	5,003	20	5	0	4	0	1	3 min
all_contains	12,310	77	119	39	5	0	1	35 min
compare_two	28,585	147	371	84	6	0	1	75 min

Table 1. Summary of proof statistics. **Nodes** and **Branches** measure the size of the proof tree, **I.step** counts the number of interactive steps performed by the user, **Q.inst** is the number of quantifier instantiations, Contract is the number of contracts applied, **Dep.** is the number of dependency contracts applied, **Loop inv.** is the number of loop invariants applied, and **Time** is the estimated time of completing the proof in the KeY theorem prover. The rows marked † come from the previous approach.

[4] This can be compared to the expression $x + y$ in Java where x and y are fields: the value of x and y depends on the heap but the meaning of the '+' operation does not.

3.2 Advanced use cases

In this subsection we illustrate the benefits of our new approach in the verification of more complicated client programs that make use of the `Collection` interface. First we consider the following use case: iterating over the elements of a collection. An iterator is obtained as a view of the elements that a collection contains. It is possible to obtain multiple iterators, each with their own local view on a collection. The question arises: what happens when using an iterator when the collection it was obtained from is modified after its creation? In practice, a `ModificationException` is thrown. To ensure that the iterator methods are only called when the backing collection is not modified in the meantime, we introduce the notion of validity of an iterator. As already discussed above, we record the events of the iterators in the owning history, alongside other events that signal whether the owning collection is modified, so that indeed we *can* define a recursive function that determines whether an iterator is still valid. Another complex feature of the iterator is that it provides a parameterless `Iterator#remove()` method, producing no return value. Its intended semantics is to delete from the backing collection the element that was returned by a previous call to `Iterator#next()`, and invalidating all other iterators.

In the previous work [16], we did verify a client of iterator and showed its termination as shown in Listing 6: but we did not verify the pure methods (termination, determinism, equational specification) used in the specification that modeled the behavior of iterators. Functions like *size* of a history which computes the total number of elements contained by the collection, *iteratorSize* of a history and an iterator which computes the total number of elements already seen by the iterator, and the function *isIteratorValid* which depends on the functions *iteratorLast* (called *last* above), *iteratorHasNext*, and *iteratorVisited*.

The large number of auxiliary functions needed to model the behavior of iterators shows a verification bottleneck we encountered in the previous approach: modeling these as pure methods and verifying their properties takes roughly the same effort as required for *multiset*, per function! Using our new approach and the workflow above, we are able to define all functions and verify the properties necessary in completing the proof, thereby eliminating much proof effort.

```
public static boolean iterate_only(Collection x) {
    Iterator it = x.iterator();
    /*@ ...
      @ decreasing \dl_size(it.owner().history()) -
            \dl_iteratorSize(it.owner().history(),it); @*/
    while (it.hasNext()) {
        it.next();
    }
    return true;
}
```

Listing 6. Iterating over the collection. Why does it terminate?

Advancing further, we can verify a modifying iterator as is shown in Listing 7. This example shows an important aspect of our new approach: being able to use Isabelle/HOL to derive non-trivial properties of the functions we have defined.

```
/*@ ...
  @ ensures \dl_size(x.history()) == 0; @*/
public static boolean iter_remove(Collection x) {
    Iterator it = x.iterator();
    /*@ ...
      @ loop_invariant \dl_iteratorSize(it.owner().history(),it) == 0;
      @ decreasing \dl_size(it.owner().history()); @*/
    while (it.hasNext()) {
        it.next();
        it.remove();
    }
    return true;
}
```

Listing 7. Iterating over the collection and removing all its elements.

We iterate over a given collection and at each step we remove the last returned element by the iterator from the backing collection. Thus, after completing the iteration, when there are no next elements left, we expect to be able to prove that the backing collection is now empty. The crucial insight here is that, after we exit the loop, we know that `hasNext()` returned **false**. We modeled the outcome of the `hasNext()` method by defining a function *iteratorHasNext* of a history and an iterator. We established in Isabelle/HOL the (non-trivial) fact that if a valid iterator has no next elements then *iteratorSize* and *size* must be equal.

Following our workflow, we have proven this fact and imported it into KeY as a taclet, which is shown in Listing 8. Since it is a loop invariant that the size of the iterator remains zero (each time we remove an element through its iterator, it is not only removed from the backing collection but also from the elements seen by the iterator), we can thus deduce that finally the collection must be empty!

```
HasNext_size {
\schemaVar \term history h;
\schemaVar \term Iterator it;
\assumes(isIteratorValid(h,it) = TRUE, ... ==>)
\find(iteratorHasNext(h,it) = FALSE)
\replacewith(size(h) = iteratorSize(h,it))
\heuristics(concrete)
};
```

Listing 8. Taclet for showing the equality between *size* and *iteratorSize*.

Advancing even further, we have verified clients that operate on two collections at the same time. This is interesting, since both collections can be of a different implementation, and can potentially interfere with each other. The technique we applied here is to specify what properties remain *invariant* of histories of all other collections, e.g. that a call to a method of one collection does

not change the history of any other collection. Since histories are not part of the heap, that a history remains invariant implies that all its (polymorphic) properties are invariant too. However, if a history contains some reference to an object on the heap, it can still be the case that properties of such an object have changed.

```
/*@ ...
  @ ensures \result = true; @*/
public static boolean all_contains(Collection x, Collection y) {
    x.addAll(y); Iterator it = y.iterator();
    /*@ ...
      @ loop_invariant (\forall Object o1;
            \dl_multiset(y.history(),o1) > 0 ==>
            \dl_multiset(x.history(),o1) > 0); @*/
    while(it.hasNext()) {
        if (!x.contains(it.next())) { return false; }
    }
    return true;
}
```

Listing 9. Using the `addAll` method and checking inclusion.

In the example given in Listing 9, we make use of the `addAll` method of the collection, adding elements of one collection to another. Clearly, during the `addAll` call, the collections interfere: collection `x` could obtain an iterator of collection `y` to add all elements of `y` to itself. So, in the specification of `addAll` we have not history invariance of `y`. Instead, we specify what properties of `y`'s history remain invariant: in this case its multiset must remain invariant (assuming `x` and `y` are not aliases). The program first performs such `addAll`, and then iterates over the collection `y` that was supplied as argument. For each of the elements in the argument collection `y`, we check whether `x` did indeed add that element, by calling `contains`. We expect that after adding all elements, that all elements must (already) be contained. Indeed, we verified this propery.

The crucial property in this verification is shown as the loop invariant: all objects that are contained in collection `y` are also contained in collection `x`. This can be verified initially: the call to iterator does not change the multisets associated to the histories of `x` and `y`, and after the `addAll` method is called this inclusion is true. But why? As already explained above, in the specification of `addAll`, we state the existence of an element list: this is an enumeration of the contents of the argument collection `y`, but for each element also a Boolean flag that states whether `x` has decided to add those elements. Since this flag depends on the actual implementation of `x`, which is inaccessible to us, the contract of `addAll` existentially quantifies such element list. Thus, under the condition that for any element that was not yet contained in `x` at least one of the elements in the element list must have a **true** flag associated, we can deduce that the loop invariant holds initially. From the loop invariant, we can further deduce that the `contains` method never returns **false**, so the then-branch returning **false**

is unreachable. Termination of the iterator can be verified as in the previous example. Hence, the whole client returns **true**.

The last example is the most complex: it is a method that compares two collections. Two collections are considered equivalent whenever they have the same multiplicities for all elements. The example shown in Listing 10 performs a destructive comparison: the collections are modified in the process. Thus, we have formulated in the contract that this method returns **true** if and only if the two collections were equivalent before calling the method.

```
/*@ ...
  @ requires x != y;
  @ ensures \result == true <==> (\forall Object o1;
        \dl_multiset(\old(x.history()),o1) ==
          \dl_multiset(\old(y.history()),o1)); @*/
public static boolean compare_two(Collection x, Collection y) {
    Iterator it = x.iterator();
    /*@ ...
      @ loop_invariant (\forall Object o1;
            \dl_multiset(\old(x.history()),o1) ==
              \dl_multiset(\old(y.history()),o1) <==>
            \dl_multiset(x.history(),o1) ==
              \dl_multiset(y.history(),o1)); @*/
    while (it.hasNext()) {
        if (!y.remove(it.next())) { return false; }
        else { it.remove(); }
    }
    return y.isEmpty();
}
```

Listing 10. Client side example for binary method.

We assume the collections are not aliases. The verification goes along the following lines: it is a loop invariant that the two collections were equivalent at the beginning of the method `compare_two` *if and only if* the two collections are equivalent in the current state. The invariant is trivially valid at the start of the method, and also at the start of the loop since the iterator does not change the multisets of either collection: the call on `x` explicitly specifies that `x`'s multiset values are preserved, but moreover specifies the invariance of the history of any other collection (so also that of `y`). For each element of `x`, we try to remove it from `y` (which does not affect the iteration over `x`, since the removal of an element of `y` specifies that the history of any other collection is invariant). If that fails, then there is an element in `x` which is not contained in `y`, hence `x` and `y` are not equivalent, hence they were not equivalent. If removal did not fail, we also remove the element from `x` through its iterator: hence `x` and `y` are equivalent iff they were equivalent. At the end of the loop we know `x` is empty (a similar argument as seen in a previous example). If `y` is not empty then it had more elements than `x`, otherwise both are empty and were also equivalent at the start.

4 Conclusion

We showed how ADTs externally defined in Isabelle/HOL can be used in JML specifications and KeY proofs for Java programs. This provides a way to use Isabelle/HOL as an additional back-end for KeY, but also to enrich the specification language. We successfully applied our approach to define an ADT for histories of Java programs and specified and verified several client programs that use core methods of the main interface of the Java Collection Framework. Our method is sufficiently powerful to support programming to interfaces, binary methods, and iterators, the latter of which requires a notion of ownership as iterator behavior depends on the history of other objects, i.e. the enclosing collection and other iterators over that collection.

In a previous paper [16], we modeled the history as an ordinary Java class. That worked, but the modeling of histories in this paper, as an external ADT with functions, offers numerous benefits. Here, we avoid pure methods that rely on the heap, which give rise to additional proof obligations every time these pure methods are used in JML specifications. Also we significantly simplified reasoning about properties of user-defined functions themselves. For example, in our case study, we reduced proofs from the previous paper (in KeY) about *multiset* modeled as a pure method from 72 minutes of work, to a fully automated verification in Isabelle/HOL with *multiset* modeled as an ADT function.

This work has opened up the possibility to define many more functions on histories, thus furthering the ability to model complex object behavior: this we demonstrated by verifying complex client code using collections. Further, while KeY is tailored for proving properties of concrete Java programs, Isabelle/HOL has more powerful facilities for general theorem proving. Our approach allows leveraging Isabelle/HOL to guarantee, for example, meta-properties such as the consistency of axioms about user-defined ADT functions. Using KeY alone, this was problematic or even impossible.

A further next step in the history-based specification of interfaces and its application to the Java Collection Framework is the development of a general history-based *refinement* theory which allows to formally verify that a class *implements* a given interface, and, more specifically, that inherited methods are correct with respect to refinements of overridden methods. For example, the class `LinkedList` inherits from `AbstractSequentialList` which inherits from `AbstractList` and thus from `AbstractCollection`, that provides a partial implementation of the `Collection` interface. Thus, not all methods of the `Collection` interface are directly implemented by `LinkedList`, but inherited along the class hierarchy.

Acknowledgements The authors wish to thank Mattias Ulbrich for pointing out a soundness issue in the taclet translation (see footnote 3) and his useful reference suggestion [9], and the anonymous referees for their comments and suggestions on how to improve this paper.

References

1. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich. *Deductive software verification—the KeY book*, volume 10001 of *LNCS*. Springer, 2016.
2. A. Banerjee, D. A. Naumann, and M. Nikouei. A logical analysis of framing for specifications with pure method calls. *ACM Trans. Program. Lang. Syst.*, 40(2), May 2018.
3. B. Beckert, J. Schiffl, P. H. Schmitt, and M. Ulbrich. Proving JDK's dual pivot quicksort correct. In *9th Conference on Verified Software, Theories, Tools, and Experiments (VSTTE)*, volume 10712 of *LNCS*, pages 35–48. Springer, 2017.
4. J. Bian and H. A. Hiep. *Integrating ADTs in KeY and their Application to History-based Reasoning: Video Material*. FigShare, 2021. Available at: https://doi.org/10.6084/m9.figshare.c.5413263.
5. J. Bian, H. A. Hiep, F. S. de Boer, and S. de Gouw. *Integrating ADTs in KeY and their Application to History-based Reasoning: Proof Files*. Zenodo, 2021. Available at: https://doi.org/10.5281/zenodo.4744268.
6. J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. Defining (co)datatypes and primitively (co)recursive functions in Isabelle/HOL, 2016. Available at: https://isabelle.in.tum.de/doc/datatypes.pdf.
7. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.
8. D. Costa, A. Andrzejak, J. Seboek, and D. Lo. Empirical study of usage and performance of Java collections. In *8th Conference on Performance Engineering*, pages 389–400. ACM, 2017.
9. A. Darvas and P. Müller. Faithful mapping of model classes to mathematical structures. In *2007 Conference on Specification and Verification of Component-Based Systems (SAVCBS)*, page 31–38. ACM, 2007.
10. S. de Gouw, F. S. de Boer, and J. Rot. Proof pearl: The key to correct and stable sorting. *J. Autom. Reason.*, 53(2):129–139, 2014.
11. S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. OpenJDK's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *27th Conference on Computer Aided Verification (CAV)*, volume 9206 of *LNCS*, pages 273–289. Springer, 2015.
12. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 213–226. ACM, 2008.
13. J.-C. Filliâtre and A. Paskevich. Why3: where programs meet provers. In *22nd European Symposium on Programming*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
14. M. Giese. Taclets and the KeY prover. *Electronic Notes in Theoretical Computer Science*, 103:67–79, 2004.
15. E. Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theorie spezifische Regeln*. PhD thesis, University of Karlsruhe, 2000.
16. H. A. Hiep, J. Bian, F. S. de Boer, and S. de Gouw. History-based specification and verification of Java collections in KeY. In *16th International Conference on Integrated Formal Methods*, pages 199–217. Springer, 2020.
17. H. A. Hiep, O. Maathuis, J. Bian, F. S. de Boer, M. C. J. D. van Eekelen, and S. de Gouw. Verifying OpenJDK's LinkedList using KeY. In *26th Conference*

on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 12079 of *LNCS*, pages 217–234. Springer, 2020.
18. M. Huisman. Verification of Java's AbstractCollection class: A case study. In *6th Conference on Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 175–194. Springer, 2002.
19. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. *Int. J. Softw. Tools Technol. Transf.*, 3(3):332–352, 2001.
20. A. Knüppel, T. Thüm, C. Pardylla, and I. Schaefer. Experience report on formally verifying parts of OpenJDK's API with KeY. In *F-IDE 2018: Formal Integrated Development Environment*, volume 284 of *EPTCS*, pages 53–70. OPA, 2018.
21. G. T. Leavens and Y. Cheon. Design by contract with jml, 2006. Available at: http://www.cs.utep.edu/cheon/cs3331/data/jmldbc.pdf.
22. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
23. K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *17th European Symposium on Programming*, volume 4960 of *LNCS*, pages 307–321. Springer, 2008.
24. B. Liskov and S. Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59, 1974.
25. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.
26. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.
27. D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. Springer, 2012.
28. D. Traytel, A. Popescu, and J. C. Blanchette. Foundational, compositional (co)datatypes for higher-order logic: Category theory applied to theorem proving. In *27th Symposium on Logic in Computer Science (LICS)*, pages 596–605. IEEE, 2012.