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

Jinting Bian, Hans-Dieter Hiep,
Frank de Boer, Stijn de Gouw

(submitted to FM2021)

Centrum Wiskunde & Informatica

July 9th, 2021

Abstract

Part 1: Integrating ADTs in KeY

*We discuss integrating **abstract data types** (ADTs) in the KeY theorem prover by a novel approach to model data types using **Isabelle/HOL** as an interactive back-end, and translate Isabelle **theorems** to user-defined **taclets** in KeY.*

Abstract

Part 1: Integrating ADTs in KeY

*We discuss integrating **abstract data types** (ADTs) in the KeY theorem prover by a novel approach to model data types using **Isabelle/HOL** as an interactive back-end, and translate Isabelle **theorems** to user-defined **taclets** in KeY.*

Part 2: Application to History-based Reasoning

*As a **case study** of this 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*

Origin of this work

1st International HacKeYthon (6–7 December, 2018)

Team **Algebraic Data Types for KeY**:

- Dominic Steinhöfel (TU Darmstadt)
- Alexander Knüppel (TU Braunschweig)
- Asmae Heydari Tabar (TU Darmstadt)
- Hans-Dieter Hiep (CWI)

Origin of this work

1st International HacKeYthon (6–7 December, 2018)

Team **Algebraic Data Types for KeY**:

- Dominic Steinhöfel (TU Darmstadt)
- Alexander Knüppel (TU Braunschweig)
- Asmae Heydari Tabar (TU Darmstadt)
- Hans-Dieter Hiep (CWI)

Q: What is the difference between?

- Algebraic Data Types (ADTs)
- Abstract Data Types (ADTs)

Motivation

Abstract Data Types (ADTs):

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data, on a higher-level of abstraction

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data,
  on a higher-level of abstraction
- Useful for relating Java implementation to abstract type

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data,
  on a higher-level of abstraction
- Useful for relating Java implementation to abstract type

History-based reasoning for Java interfaces:

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data,
  on a higher-level of abstraction
- Useful for relating Java implementation to abstract type

History-based reasoning for Java interfaces:

- Java interface: no concrete implementation available,

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data, on a higher-level of abstraction
- Useful for relating Java implementation to abstract type

History-based reasoning for Java interfaces:

- Java interface: no concrete implementation available, so keep track of all method **calls** and **returns** in a history.

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data,
  on a higher-level of abstraction
- Useful for relating Java implementation to abstract type

History-based reasoning for Java interfaces:

- Java interface: no concrete implementation available,
  so keep track of all method **calls** and **returns** in a history.
- Model histories using ADTs

Motivation

Abstract Data Types (ADTs):

- Reasoning about *conceptual* data, on a higher-level of abstraction
- Useful for relating Java implementation to abstract type

History-based reasoning for Java interfaces:

- Java interface: no concrete implementation available, so keep track of all method **calls** and **returns** in a history.
- Model histories using ADTs
- Verification of the functional properties of clients

Part 1:
Integrating ADTs in KeY

Abstract Data Types

- Modelling abstract data types in KeY

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: ?

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

- Defining user-defined functions and predicates

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

- Defining user-defined functions and predicates

  e.g. *incr1*: **option** (**int**, `Object`) $\rightarrow$ **option** (**int**, `Object`)

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

- Defining user-defined functions and predicates

  e.g. *incr1*: **option** (**int**, `Object`) $\rightarrow$ **option** (**int**, `Object`)
  *is*Some? **option** (**int**, `Object`)

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

- Defining user-defined functions and predicates

  e.g. *incr1*: **option** (**int**, `Object`) $\rightarrow$ **option** (**int**, `Object`)
  *is*Some? **option** (**int**, `Object`)

- Reasoning about data types and their properties

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

- Defining user-defined functions and predicates

  e.g. *incr1*: **option** (**int**, `Object`) $\rightarrow$ **option** (**int**, `Object`)
  *is*Some? **option** (**int**, `Object`)

- Reasoning about data types and their properties

  e.g. $incr1(None) = None$

Abstract Data Types

- Modelling abstract data types in KeY

  Built-in: `\bigint`, `\seq`, `\locset`

  User-defined: e.g. an **option** of pairs of **int** and `Object`

- Defining user-defined functions and predicates

  e.g. *incr1*: **option** (**int**, `Object`) $\rightarrow$ **option** (**int**, `Object`)
  *is*Some? **option** (**int**, `Object`)

- Reasoning about data types and their properties

  e.g. $incr1(None) = None$
  $\neg isSome(None)$

Problem & Solutions

Problem:

KeY has no *direct* support for ADTs

Problem & Solutions

Problem:

KeY has no *direct* support for ADTs

Solution directions:

Problem & Solutions

Problem:

KeY has no *direct* support for ADTs

Solution directions:

1. Model ADTs using Java classes (previous work)

Problem & Solutions

Problem:

KeY has no *direct* support for ADTs

Solution directions:

1. Model ADTs using Java classes (previous work)

   Pros:

   - Specification at level of programming language
   - Computable functions, also available within Java
   - Only need to use and know Java & JML

Problem & Solutions

Problem:

KeY has no *direct* support for ADTs

Solution directions:

1. Model ADTs using Java classes (previous work)

   Pros:

   - Specification at level of programming language
   - Computable functions, also available within Java
   - Only need to use and know Java & JML

   Cons:

   - Lifting to specification language (pure functions)
   - Elements live on the heap, referential equality
   - Large verification overhead:
     termination, determinism modulo heap dependency

Problem & Solutions

Solution directions:

2. Add user-defined sorts and taclet rules to KeY (current)

Problem & Solutions

Solution directions:

2. Add user-defined sorts and taclet rules to KeY (current)

   Pros:
   - Declarative specification, no programming
   - Directly available in logical specifications
   - No interference by Java program execution

Problem & Solutions

Solution directions:

2. Add user-defined sorts and taclet rules to KeY (current)

   Pros:

   - Declarative specification, no programming
   - Directly available in logical specifications
   - No interference by Java program execution

   Cons:

   - KeY specific: need to know taclets & JavaDL
   - Not necessarily computable
   - Possibly inconsistent

Problem & Solutions

Solution directions:

2. Add user-defined sorts and taclet rules to KeY (current)

   Pros:
   - Declarative specification, no programming
   - Directly available in logical specifications
   - No interference by Java program execution

   Cons:
   - KeY specific: need to know taclets & JavaDL
   - Not necessarily computable
   - Possibly inconsistent

Our Approach

We use the **Isabelle/HOL** theorem prover to define **data types.**

Our Approach

We use the **Isabelle/HOL** theorem prover to define **data types.**

We **translate** data types into:

- sorts (one per data type)
- function symbols (constructors, functions)

Our Approach

We use the **Isabelle/HOL** theorem prover to define **data types.**

We **translate** data types into:

- sorts (one per data type)
- function symbols (constructors, functions)

We use the new function symbols in JML **specifications**:
using the escape hatch `\dl_`

Our Approach

We use the **Isabelle/HOL** theorem prover to define **data types.**

We **translate** data types into:

- sorts (one per data type)
- function symbols (constructors, functions)

We use the new function symbols in JML **specifications**:
using the escape hatch `\dl_`

We use KeY to generate (and prove) **proof obligations.**

Our Approach

We use the **Isabelle/HOL** theorem prover to define **data types.**

We **translate** data types into:

- sorts (one per data type)
- function symbols (constructors, functions)

We use the new function symbols in JML **specifications**:
using the escape hatch `\dl_`

We use KeY to generate (and prove) **proof obligations.**

When we are stuck on proof involving our logical symbols:
we **prove** a relevant and useful theorem **in Isabelle.**

Our Approach

We use the **Isabelle/HOL** theorem prover to define **data types.**

We **translate** data types into:

- sorts (one per data type)
- function symbols (constructors, functions)

We use the new function symbols in JML **specifications**:
using the escape hatch `\dl_`

We use KeY to generate (and prove) **proof obligations.**

When we are stuck on proof involving our logical symbols:
we **prove** a relevant and useful theorem **in Isabelle.**

We **import** the theorems of Isabelle as taclets **in KeY.**

Integrating ADTs in KeY

Our approach is:

- Lazy

Integrating ADTs in KeY

Our approach is:

- Lazy:
  - only verify theorems in Isabelle when needed

Integrating ADTs in KeY

Our approach is:

- Lazy:
  - only verify theorems in Isabelle when needed
- Consistent

Integrating ADTs in KeY

Our approach is:

- Lazy:
  - only verify theorems in Isabelle when needed
- Consistent, if:
  - Isabelle theory is consistent

Integrating ADTs in KeY

Our approach is:

- Lazy:
  - only verify theorems in Isabelle when needed
- Consistent, if:
  - Isabelle theory is consistent, and
  - the translation is sound

Example

Isabelle/HOL:

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

Example

Isabelle/HOL:

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

Import instantiated signature (.key file):

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

Example

Isabelle/HOL:

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

Import instantiated signature (.key file):

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

Use `\dl_Some` and `\dl_None` in specification language (JML):

```
//@ requires \dl_Some(x) = o;
...
```

Example

(KeY) We are stuck at a proof obligation:

```
...,   Some(x) = o,   o = Some(y)
==>
x = y
```

Example

(KeY) We are stuck at a proof obligation:

```
...,   Some(x) = o,   o = Some(y)
==>
x = y
```

(Isabelle) Prove the theorem:

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

Example

(KeY) We are stuck at a proof obligation:

```
...,   Some(x) = o,   o = Some(y)
==>
x = y
```

(Isabelle) Prove the theorem:

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

Import Isabelle theorem as KeY taclet:

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

Part 2:
Application to History-based Reasoning

Histories

To specify an interface, we use the **history-based approach**:

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**
2. Define **abstract properties** of histories

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**
2. Define **abstract properties** of histories
3. Specify **object behavior** in terms of its abstract properties

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**
2. Define **abstract properties** of histories
3. Specify **object behavior** in terms of its abstract properties

General approach applicable to all interfaces,
we focus here on `Collection` interface.

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**
2. Define **abstract properties** of histories
3. Specify **object behavior** in terms of its abstract properties

General approach applicable to all interfaces,
we focus here on `Collection` interface.

No built-in support for histories in KeY or JML!

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**
2. Define **abstract properties** of histories
3. Specify **object behavior** in terms of its abstract properties

General approach applicable to all interfaces,
we focus here on `Collection` interface.

No built-in support for histories in KeY or JML!

Our previous work:

H. A. Hiep, J. Bian, F. S. de Boer, and S. de Gouw. History-based specification and verification of Java Collections in KeY. In *International Conference on Integrated Formal Methods*, pages 199–217. Springer, 2020

Histories

To specify an interface, we use the **history-based approach**:

1. Record all method invocations on an interface in a **history**
2. Define **abstract properties** of histories
3. Specify **object behavior** in terms of its abstract properties

General approach applicable to all interfaces,
we focus here on `Collection` interface.

No built-in support for histories in KeY or JML!

Our previous work:
H. A. Hiep, J. Bian, F. S. de Boer, and S. de Gouw. History-based specification and verification of Java Collections in KeY. In *International Conference on Integrated Formal Methods*, pages 199–217. Springer, 2020

But now we can use ADTs!

Example program

```
Object add_remove(Collection x, Object y) {
  if (x.add(y)) {
    x.remove(y);
  }
  return y;
}
```

Q: What does this method do?

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Let *h* be a history of Collection events.
A *multiset* is defined inductively:

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Let h be a history of Collection events.
A *multiset* is defined inductively:

- $multiset(\varepsilon) = \emptyset$

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Let h be a history of Collection events.
A *multiset* is defined inductively:

- $\mathit{multiset}(\varepsilon) = \emptyset$
- $\mathit{multiset}(\texttt{add}(y) \mapsto \textbf{true} :: h) = \mathit{multiset}(h) \cup \{y\}$

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Let h be a history of Collection events.
A *multiset* is defined inductively:

- $multiset(\varepsilon) = \emptyset$
- $multiset(\texttt{add}(y) \mapsto \textbf{true} :: h) = multiset(h) \cup \{y\}$
- $multiset(\texttt{add}(y) \mapsto \textbf{false} :: h) = multiset(h)$

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Let h be a history of Collection events.
A *multiset* is defined inductively:

- $\mathit{multiset}(\varepsilon) = \emptyset$
- $\mathit{multiset}(\texttt{add}(y) \mapsto \textbf{true} :: h) = \mathit{multiset}(h) \cup \{y\}$
- $\mathit{multiset}(\texttt{add}(y) \mapsto \textbf{false} :: h) = \mathit{multiset}(h)$
- $\mathit{multiset}(\texttt{remove}(y) \mapsto \textbf{true} :: h) = \mathit{multiset}(h) - \{y\}$

Abstract properties of `Collection`

The contents of a collection is modelled as a multiset.

Let *h* be a history of Collection events.
A *multiset* is defined inductively:

- $\mathit{multiset}(\varepsilon) = \emptyset$
- $\mathit{multiset}(\texttt{add}(y) \mapsto \textbf{true} :: h) = \mathit{multiset}(h) \cup \{y\}$
- $\mathit{multiset}(\texttt{add}(y) \mapsto \textbf{false} :: h) = \mathit{multiset}(h)$
- $\mathit{multiset}(\texttt{remove}(y) \mapsto \textbf{true} :: h) = \mathit{multiset}(h) - \{y\}$
- $\mathit{multiset}(\texttt{remove}(y) \mapsto \textbf{false} :: h) = \mathit{multiset}(h)$

Example contract

```
/*@ ...
  @ ensures (\forall Object o1;
      \dl_multiset(x.history(),o1) ==
      \dl_multiset(\old(x.history()),o1)); @*/
Object add_remove(Collection x, Object y)
...
```

Example contract

```
/*@ ...
  @ ensures (\forall Object o1;
        \dl_multiset(x.history(),o1) ==
        \dl_multiset(\old(x.history()),o1)); @*/
Object add_remove(Collection x, Object y)
...
```

- `\dl_multiset`: escape hatch to function symbol

Example contract

```
/*@ ...
  @ ensures (\forall Object o1;
      \dl_multiset(x.history(),o1) ==
      \dl_multiset(\old(x.history()),o1)); @*/
Object add_remove(Collection x, Object y)
...
```

- `\dl_multiset`: escape hatch to function symbol
- `x.history()`: associated history to interface instance

Example contract

```
/*@ ...
  @ ensures (\forall Object o1;
       \dl_multiset(x.history(),o1) ==
       \dl_multiset(\old(x.history()),o1)); @*/
Object add_remove(Collection x, Object y)
...
```

- `\dl_multiset`: escape hatch to function symbol
- `x.history()`: associated history to interface instance

Moral: the contents of the collection remains the same.

What improvement?

Previous approach:

- Use Java objects to encode histories
- Use Java methods to define functions
- Verify pure functions (totality, determinacy, dependency)
- Total verification effort: est. **75 minutes**

What improvement?

Previous approach:

- Use Java objects to encode histories
- Use Java methods to define functions
- Verify pure functions (totality, determinacy, dependency)
- Total verification effort: est. **75 minutes**

New approach (using ADTs!):

- Define ADTs and functions in Isabelle/HOL
- Lazy approach to proving/importing theorems
- Total verification effort: **automatic**

What improvement?

Previous approach:

- Use Java objects to encode histories
- Use Java methods to define functions
- Verify pure functions (totality, determinacy, dependency)
- Total verification effort: est. **75 minutes**

New approach (using ADTs!):

- Define ADTs and functions in Isabelle/HOL
- Lazy approach to proving/importing theorems
- Total verification effort: **automatic**

What can we do with the gain?

Verify more advanced clients!

Example program

```
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;
}
```

Example program

```
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;
}
```

- `it.owner()`: all events recorded in *owning* collection

Example program

```
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;
}
```

- `it.owner()`: all events recorded in *owning* collection
- `\dl_size`: sum of all multiplicities

Example program

```
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;
}
```

- `it.owner()`: all events recorded in *owning* collection
- `\dl_size`: sum of all multiplicities
- `\dl_iteratorSize`: sum of all multiplicities of iterator

What improvement?

Previous approach:

- Verify pure functions (totality, determinacy, dependency):
  - size
  - iteratorSize
  - isIteratorValid
  - iteratorLast
  - iteratorHasNext
  - iteratorVisited
- Total verification effort: unknown. More than **8 hours?**

What improvement?

Previous approach:

- Verify pure functions (totality, determinacy, dependency):
  - size
  - iteratorSize
  - isIteratorValid
  - iteratorLast
  - iteratorHasNext
  - iteratorVisited
- Total verification effort: unknown. More than **8 hours?**

New approach (using ADTs!):

- Define functions in Isabelle/HOL
- Total verification effort: **automatic**

Isabelle functions

Function *iteratorSize*:

fun *iterSize: History* $\rightarrow$ *Iterator* $\rightarrow$ ***int where***
iterSize (ε) z = 0
iterSize (iterator()$\mapsto$*y :: h) z = (y = z ? 0 : iterSize h z)*
iterSize (y.next()$\mapsto$*x :: h) z = iterSize h z + (y = z ?1:0)*
iterSize (y.remove() :: h) z = (y = z ? iterSize h z - 1 : 0)
iterSize (e :: h) z = (modify e ? 0 : iterSize h z)

Isabelle functions

Function *iteratorSize*:

> ***fun*** *iterSize: History* $\rightarrow$ *Iterator* $\rightarrow$ ***int where***
> *iterSize* (ε) *z = 0*
> *iterSize (iterator()*$\mapsto$*y :: h) z = (y = z ? 0 : iterSize h z)*
> *iterSize (y.next()*$\mapsto$*x :: h) z = iterSize h z + (y = z ?1:0)*
> *iterSize (y.remove() :: h) z = (y = z ? iterSize h z - 1 : 0)*
> *iterSize (e :: h) z = (modify e ? 0 : iterSize h z)*

and *iteratorHasNext*:

> ***fun*** *iterHasNext: History* $\rightarrow$ *Iterator* $\rightarrow$ ***bool*** *where*
> *iterHasNext h z = (iterSize h z < size h)*

Imported theorem

Isabelle lemma:

lemma *HasNext_size:*
isValid h $\Longrightarrow$
isIteratorValid h z $\Longrightarrow$
$\neg$*iteratorHasNext h z* $\Longrightarrow$
size h = iteratorSize h z

Imported theorem

Isabelle lemma:

lemma *HasNext_size:*
isValid h $\Longrightarrow$
isIteratorValid h z $\Longrightarrow$
$\neg$*iteratorHasNext h z* $\Longrightarrow$
size h = iteratorSize h z

Imported taclet:

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

Example program

```
boolean compare(Collection x, Collection y) {
  Iterator it = x.iterator();
  /*@ ...
      ...
      ...
      ...
      ...
      ... @*/
  while (it.hasNext()) {
    if (!y.remove(it.next())) { return false; }
    else { it.remove(); }
  }
  return y.isEmpty();
}
```

Example program

```
boolean compare(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();
}
```

Example program

```
boolean compare(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();
}
```

New approach, total verification effort: **75 minutes**

Concluding

Summary

1. Integrating ADTs in KeY:
   - Isabelle/HOL as interactive back-end
   - A lazy approach that guarantees consistency

Summary

1. Integrating ADTs in KeY:
   - Isabelle/HOL as interactive back-end
   - A lazy approach that guarantees consistency

2. Reasoning about interface clients using histories:
   - Case study: `Collection`
   - Advanced clients: are two collections equal?

Summary

1. Integrating ADTs in KeY:
   - Isabelle/HOL as interactive back-end
   - A lazy approach that guarantees consistency

2. Reasoning about interface clients using histories:
   - Case study: `Collection`
   - Advanced clients: are two collections equal?

Paper submitted to FM2021

Open Science: video and source material available

Future work

- Continue with the specification of the Java Collection Framework:
  - `Collection`
  - `Map`
  - `Set`
  - `List`
  - etc.

Future work

- Continue with the specification of the Java Collection Framework:
  - `Collection`
  - `Map`
  - `Set`
  - `List`
  - etc.
- Reasoning about invariant properties
  - Problem: histories of objects not called may not remain the same

Future work

- Continue with the specification of the Java Collection Framework:
  - `Collection`
  - `Map`
  - `Set`
  - `List`
  - etc.
- Reasoning about invariant properties
  - Problem: histories of objects not called may not remain the same
- General history-based refinement theory
  - Formally verify that a class implements an interface
  - `LinkedList` :> `AbstractSequentialList` :> `AbstractList` :> `AbstractCollection`