Specification of Generic APIs, or: Why Algebraic May Be Better Than Pre/Post

Anya Helene Bagge
Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~anya/

Magne Haveraaen
Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~magne/

ABSTRACT

Specification based on Floyd-Hoare logic, using pre and postconditions, is common in languages aimed at high integrity software. Such pre/postcondition specifications are geared towards verification of code. While this specification technique has proven quite successful in the past 40 years, it has limitations when applied to API specification, particularly specification of generic interfaces.

API-oriented design and genericity is of particular importance in modern large-scale software development. In this situation, algebraic specification techniques have a significant advantage. Unlike pre/post-based specification, which deals with the inputs and outputs of one operation at a time, algebraic specification deals with the relationships between the different operations in an API, which is needed in the specification of generic APIs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Reliability, Verification, Languages

Keywords

Program specification; API specification; Generic programming; APIs; Axioms

1. INTRODUCTION

Modern interface-oriented and generics-heavy development methods [29, 35] focus on abstract APIs, rather than dealing directly with concrete data structures. In this setting, the implementation of a class will never deal directly with objects of other classes—rather, access is through a well-defined interface that hides the concrete class of each object. One implementation class is exchangeable for another, as long as it complies with the same interface.

Generic libraries for collection classes are prime exponents of this idea. The different collection implementations have specific requirements on the API of the elements. The collection class only accesses the elements through the API. When the API requirements are satisfied, the class provides the service expected of it. Generic programming extends this approach to any algorithm and data structure. The goal is to make the implementation as flexible as possible, stating the minimal requirements needed on the data in order to let the implementation work.

Many languages have mechanisms for syntactic requirements on generic parameters. Examples are interfaces in Java and packages in Ada. This is sufficient to ensure independently that the generic code is type correct and that the instantiation of the generic code will be type correct. In C++ the template mechanism is used for generic programming, but in itself offers no such guarantee, so every instantiation must be fully typechecked. An attempt at remedying this situation was made in the form of *concepts* [17], but the proposal was deemed to complex. Interestingly, the C++ concept proposal included *axioms* to provide semantic constraints as well as the syntactic requirements. Neither the Java nor the Ada mechanisms include any semantic constraints.

Due to the abstract nature of interfaces, semantics are more easily specified by relating the operations of the interface to each other, rather than by trying to specify the inputs and outputs of each operation separately. In particular, specifying the output of an operation may be impossible until we know which concrete data types are involved.

Consider the hash and equals methods of Java [15]:

```
1 public int hashCode();
  public boolean equals(Object other);
```

The behavior of hashCode is specified entirely by its relationship to equals: all equal objects must have the same hash value, i.e. a.equals(b) implies that a.hashCode() == b.hashCode() for all a and b. This is an *axiom*, in the algebraic specification sense. In Java, satisfying these properties is necessary in order for collections like HashMap and HashSet to work as specified.

In algebraic specification terminology, a *signature* defines the syntax of an API by listing its types and operations (with argument lists and result types). Together with a set of *axioms* describing the intended semantics, we get a complete API specification. The axioms normally take the form of universally quantified expressions in some logic. The axioms

HILT 2014, October 18–21, 2014, Portland, OR, USA.

ACM 978-1-4503-3217-0/14/10 ...$15.00.
http://dx.doi.org/10.1145/2663171.2663183.

relate the different operations of the API to each other. *Models* (implementations) of a specification can be constructed by associating concrete data structures with the types, and concrete implementation code with the operations, in such a way that the axioms hold (are *satisfied*). A typical abstract data type declaration corresponds to a signature; with axioms we get an API specification; and with implementation class(es) we get one (or more) models.

In contrast, the pre/post specification technique is based on Floyd-Hoare logic, using triples of the form $\{P\}\ C\ \{Q\}$, where the postcondition Q holds after the command C, if the precondition P holds before. This is typically realized in the form of a requires or pre clause for the precondition, and an ensures or post clause for the postcondition, as seen in e.g., Eiffel [27], JML [23] or SPARK [5].

Preconditions are particularly important in specifying *partiality* – the range of valid parameters of a function. In the case of hashCode and equals above, Java enforces an implicit precondition that this != null. For Java's equals, the semantics is normally such that passing null as the argument should give the result false, hence there is no precondition on the argument.

In algebraic specifications, handling of partiality and preconditions has traditionally been inelegant, often leading to significant clutter in the axioms, see [28] for an overview. This may have lead to an impression that algebraic specifications are difficult to use in practical settings. In *guarded algebras* [20], preconditions are stated at the level of the signature (similarly to pre/post specifications), and all axioms are written on the implicit assumption that the preconditions hold. The clutter of guarding against precondition problems thus disappears, while the benefit of specifying the abstract API is retained.

In this paper we do not delve into the theory of the specification technique. Rather we contribute a practical approach to API specifications, where the axioms are written as code using assertions. This allows direct integration with unit testing systems, if the axioms are written using unit testing assertions. Such axioms should also be readily exploitable by language related proof tools, e.g., the machinery used for proving pre/post specifications. The paper shows this technique for writing axioms in Java, C++ and Ada, and relates it to the QuickCheck [8] test framework for Haskell.

The paper is organized as follows. In section 2 we motivate axioms for API specifications, and show several examples related to the Java library. Then we discuss the approach for other languages (C++, Ada, Haskell) in section 3. Section 4 discusses axioms versus pre/post specifications, followed by the issue of purity and side effects (section 5). We will also briefly discuss the background and history of specification techniques (section 6) before we conclude (section 7).

2. API SPECIFICATIONS

Specifications of APIs differ from the specification of individual functions. In APIs there are interaction effects between functions, and the specification focuses on these.

Consider the Java collection classes. These exist in many variations, but the Collection interface defines the basic properties. The method add is in the Java 7 library given the following specification[1].

> Ensures that this collection contains the specified element (optional operation). Returns true if this collection changed as a result of the call. (Returns false if this collection does not permit duplicates and already contains the specified element.)
>
> Collections that support this operation may place limitations on what elements may be added to this collection. In particular, some collections will refuse to add null elements, and others will impose restrictions on the type of elements that may be added. Collection classes should clearly specify in their documentation any restrictions on what elements may be added.
>
> If a collection refuses to add a particular element for any reason other than that it already contains the element, it *must* throw an exception (rather than returning false). This preserves the invariant that a collection always contains the specified element after this call returns.

This textual description subtly relates the effect of the add method to the result of calling the contains method. Further, there are many conditions which need to raise exceptions. If the collection is immutable, the add method will not be supported and must raise UnsupportedOperationException. For some precondition violations, like adding a null reference to a collection that does not allow this, the NullPointerException should be thrown. For other violations, like adding a duplicate element to a collection which does not allow this, the add method should return normally with a return value of false, not modifying the collection. Thus any method that queries the status of the collection, e.g., the size method, should not observe any difference between the two states of the collection. These properties can be written down as normal Java code[2]. The following procedure captures the specification above.

```
public static <E>
void addEnsuresCollectionContainsElement(
    Collection<E> c, E t) {
  try {
  // Records some data on current status of c.
  int size = c.size();
  boolean contained = c.contains(t);

  // Attempt to add an element.
  if (c.add(t)) {
    // Element added to the collection.
    assertEquals(size + 1, c.size());
  } else {
    // Element already present.
    assertTrue(contained);
    assertEquals(size, c.size());
  }
  // Check that the element is present.
  assertTrue(c.contains(t));
  } catch (UnsupportedOperationException
    | ClassCastException
```

[1] From http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html#add(E)

[2] The Java based axioms used in this paper work with JAxT [19], an axiom based testing tool for Java. JAxT couples axioms and test data to the relevant interfaces and classes, and is integrated with JUnit through using JUnit assertions.

```
    | NullPointerException
    | IllegalArgumentException
    | IllegalStateException e) {
    // OK: precondition violation indicated.
  }
}
```

Note the interactions between methods add, contains and size. The specification is as much a description of how these methods interact, as a description of the add method itself. The interaction is somewhat overwhelmed by the code used to handle precondition situations, a problem also in regular Java code that tries to deal with all error situations.

This rather complex interaction between features and preconditions has readily been captured as Java code. We believe many API features can be specified as code, yielding several benefits.

- The axiom reads as normal code, a notation familiar to programmers.
- The axiom is executable as a parameterized unit test, and in fact here we use the JUnit assertion facility to make the claims in the axiom, hence supporting integration with unit testing tools.
- The semantics of the axiom is fully compatible with the semantics of the host language.

The latter implies that any proof tool for API specification needs to understand the programming language syntax and semantics, and *only* needs to understand the programming language's syntax and semantics since no additional specification notation is used.

We can further explore this style of axiom notation by providing specializations for a few of the cases covered above. The next two axioms are mutually exclusive and must be used for appropriate collections; the first for immutable collections where add is not supported, the second requires that the add method is implemented.

```
public static <E>
void add_unsupported(Collection<E> c, E t) {
  try {
    c.add(t);
    fail ("add() did not throw.");
  } catch (UnsupportedOperationException e) {
    // OK: intended behavior.
  } catch (ClassCastException
    | NullPointerException
    | IllegalArgumentException
    | IllegalStateException e) {
      fail ("add() throws wrong exception.");
  }
}
```

```
public static <E>
void add_supported(Collection<E> c, E t) {
  try {
    c.add(t);
    // OK: call succeeded.
  } catch (UnsupportedOperationException e) {
    fail ("add() is required for this class");
  } catch (ClassCastException
      | NullPointerException
      | IllegalArgumentException
      | IllegalStateException e) {
    // OK: call violated preconditions.
  }
}
```

The second axiom above implies that normal values added to the collection have the normal behavior described in the axiom addEnsuresCollectionContainsElement above.

A similar pattern can be used for dealing with collections that disallow, or allow, null references, respectively.

```
public static <E>
void add_null_invalid (Collection<E> c) {
  try {
    c.add(null);
    fail ("add() is required to refuse null elements.");
  } catch (UnsupportedOperationException e) {
    fail ("add() is supposed to be implemented.");
  } catch (NullPointerException e) {
    // OK
  } catch (ClassCastException
      | IllegalArgumentException
      | IllegalStateException e) {
    fail ("add() should throw NullPointerException.";
  }
}
```

```
public static <E>
void add_null_valid (Collection<E> c) {
  try {
    c.add(null);
    // OK
  } catch (UnsupportedOperationException
      | NullPointerException
      | ClassCastException
      | IllegalArgumentException e) {
    fail ("add() is required to accept null elements.");
    } catch (IllegalStateException e) {
    // OK, maybe a buffer overflow
  }
}
```

Axioms written as code are generally able to deal with all situations that the programmer must handle in the code.

2.1 Generic Requirement APIs

The purpose of generic programming is to reuse algorithms and data structures by parameterizing them on types and operations, i.e., the *generic requirement API*. The meaning of the parameterized data structures and algorithms very much depend on the semantics of the generic requirement API. For instance, an algorithm for sorting data in an array, requires that the elements have a total order operation available. When implementing a hash map, a hash function compatible with the equality function are the basic requirements, implementing a matrix package requires at least a commutative ring as the element type, and so forth.

Mainstream languages supporting generic programming include Ada, C++ and Java. They all provide means for declaring *syntactic* aspects of generic requirements, possibly bundling generic arguments together in packages, templates or interfaces.

Java is an illustrative example, where the requirements for their collection classes have received extensive documenta-

tion in the standard library. Consider the Comparable API from Java[3].

```
interface Comparable<T> {
  int compareTo(T o);
}
```

The Comparable API is one of the standard interfaces offered by Java, and must be implemented in order to use e.g. the sorted collection classes from the Java Library. Classes that implement this interface must provide a total order via the compareTo method. This method has a number of properties. Quoting from the documentation:

> The implementer must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)
>
> The implementer must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.
>
> Finally, the implementer must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.
>
> It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)).

The compareTo method combines several total comparison relations into one:

- x.compareTo(y) $<$ 0 is $x < y$,
- x.compareTo(y) <= 0 is $x \leq y$,
- x.compareTo(y) == 0 is $x = y$,
- x.compareTo(y) >= 0 is $x \geq y$, and
- x.compareTo(y) $>$ 0 is $x > y$.

The first property from the Java documentation states that $<$ and $>$, $\leq$ and $\geq$ are duals, and also defines *symmetry* for $=$. The second property is *transitivity* for $>$, and hence for $<$. The third gives transitivity and *reflexivity* for $=$, hence these properties follow for $\leq$ and $\geq$. The *antisymmetry* of $\leq$ and $\geq$ follow from the first property and antisymmetry for int comparisons. *Connectedness* follows from compareTo being a function (it has to give a verdict for every combinations of arguments). The fourth property encourages a weak form of *congruence*, ensuring that objects that are alike using compareTo also are alike using equals. These semantic requirements can be captured as axioms in Java code.

```
public static <T extends Comparable<T>>
void prop1(T x, T y) {
  try {
    assertEquals(Math.signum(x.compareTo(y)),
        -Math.signum(y.compareTo(x)));
  } catch (RuntimeException e) {
    // OK;
  }
}

public static <T extends Comparable<T>>
void prop2(T x, T y, T z) {
  try {
    if (x.compareTo(y) > 0 && y.compareTo(z) > 0) {
      assertTrue(x.compareTo(z) > 0);
    }
  } catch (RuntimeException e) {
    // OK;
  }
}

public static <T extends Comparable<T>>
void prop3(T x, T y, T z) {
  try {
    if (x.compareTo(y) == 0) {
      assertEquals(Math.signum(x.compareTo(z)),
          -Math.signum(y.compareTo(z)));
    }
  } catch (RuntimeException e) {
    // OK;
  }
}

public static <T extends Comparable<T>>
void prop4(T x, T y) {
  try {
    assertEquals(x.compareTo(y) == 0, x.equals(y));
  } catch (RuntimeException e) {
    // OK;
  }
}
```

The specification also makes strong statements on exceptions. The first property states that compareTo is symmetric also for exceptions. Our code-oriented style for writing axioms can deal with this.

```
public static <T extends Comparable<T>>
void strongSymmetry(T x, T y) {
  try {
    x.compareTo(y);
    y.compareTo(x);
    // OK: neither call throws an exception.
  } catch (RuntimeException e) {
    // at least one of the calls throws an exception
    try {
      x.compareTo(y);
      fail("x.compareTo(y) does not throw!");
    } catch (RuntimeException e1) {
      try {
        y.compareTo(x);
        fail("y.compareTo(x) does not throw!");
      } catch (RuntimeException e2) {
        // OK! Both calls fail symmetrically.
      }
    }
  }
}
```

This property is even more specific for comparisons with null, where another part of the specification states that since null.compareTo(x) will cause a NullPointerException, then x.compareTo(null) must also throw NullPointerException.

```
public static <T extends Comparable<T>>
void compareTo_null(T x) {
  try {
```

[3]From http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

```
    x.compareTo(null);
    fail ("Should throw a NullPointerException");
  } catch (NullPointerException e) {
    // As required
  }
}
```

Specifying that throwing of exceptions is symmetric, or when a specific exception is to be thrown, may seem strange from a pre/postcondition viewpoint, where throwing an exception may be thought of as checking a precondition (see section 4). However, considering the generic use cases, knowing that certain exceptions are thrown in specific circumstances makes sense. When writing a sorting algorithm or binary search, the strong symmetry ensures that it does not matter for the user of the compareTo method which element is used for which argument in the call. The method will behave symmetrically with respect to throwing in general. Specifically, if a collection allow null references as elements, its algorithms can possibly be simplified since null pointer exceptions occur consistently whether the null element is used as the left or the right argument of compareTo.

2.2 API Specifications and API Enrichment

Keeping the specification logic limited to program code obviously has limitations. Many properties cannot readily be expressed due to the weakness of code as a specification logic, e.g., lack of quantifiers, no ghost variables, no modal operators etc. Yet there are interesting interplays between specification language power and API complexity: a less powerful specification language can in many cases be compensated by extending the API.

The specification of a sorting algorithm has two components: the fact that the output is sorted, and the fact that the output is a permutation of the input. The sortedness condition is readily specified in a simple logic, but the permutation condition normally requires a more powerful specification logic.

We can stick with the simpler logic by expanding the API. In addition to the sorting method, let the API also contain a function that counts the occurrences of elements in the data structure.

```
/** Sorts the data in situ. */
abstract public void sort();
/** Counts the number of occurrences of t. */
abstract public int count(T t);
```

Now it becomes possible to write both the sortedness and the permutation condition in a few lines of code.

```
public static <T extends Comparable<T>>
void isSorted(MyArrayList<T> list, int i) {
  if (list.size() <= i) return;
  if (0 == i) return;
  list.sort();
  assertTrue(list.get(i-1).compareTo(list.get(i)) <= 0);
}

public <T extends Comparable<T>>
void isPermutation(MyArrayList<T> list, T t) {
  int precount = list.count(t);
  list.sort();
  int postcount = list.count(t);
  assertEquals(precount, postcount);
}
```

The first axiom states that the list is in the correct ordering for an arbitrary selected pair of neighboring indices. The second axiom ensures that for an arbitrary element, sorting the list does not change the number of times the element occurs in the list.

In our experience enlarging the API in order to simplify specifications seems to have several benefits.

- The specification becomes simpler.
- It becomes much easier to test the API as the need for test fixtures and mock objects is reduced.
- Often the richer API turns out to be more reusable than the leaner API.

These effects may be a consequence of the API becoming somewhat more complete and self contained when it is enriched in this way. On the other hand, enlarging the API induces the cost of developing the additional methods.

3. AXIOMS IN OTHER LANGUAGES

3.1 C++ Concepts and the Catsfoot Library

The failed *concepts* proposal for C++11 [17] came with builtin algebraic specification support with axiom definitions as part of the concept interface definitions. The constructs allowed fundamental algebraic concepts to be expressed and integrated with generic libraries [16].

Although the proposal was ultimately rejected, the specification features as well as most of the other features can be provided through libraries such as the Catsfoot C++ template library [2].

The following example illustrates how Catsfoot can be used to specify total orders, for any type T and associated relation Rel. Axioms are placed in specially-structured C++ classes inheriting from the Catsfoot class concept.

```
template <typename T, typename Rel>
struct total_order : public concept {
```

Each concept may be generic, and have a number of requirements. In this case, we require a relation, encoded as a *functor* (class with overloaded function call operator). It should take two arguments, be callable, and return something convertible to bool:

```
  typedef concept_list <
    is_callable <Rel(T, T)>,
    std :: is_convertible <typename is_callable<Rel(T, T)>
                           :: result_type , bool>,
    > requirements;
```

Next come the axioms. The operator we are specifying is provided as one of the parameters (this may seem counter-intuitive, but remember that the actual relation is defined by the functor class and not the object itself, which is typically a dummy object obtained through the default constructor):

```
  static void is_antisymmetric(const T& a, const T& b,
                               const Rel& rel) {
    if (rel(a,b) && rel(b, a))
      axiom_assert(a == b);
  }
  static void is_total(const T& a, const T& b,
                       const Rel& rel) {
    axiom_assert(rel(a,b) || rel(b,a));
```

```
  }
  static void is_transitive (const T& a, const T& b,
                             const T& c, const Rel& rel) {
    if (rel(a, b) && rel(b, c))
      axiom_assert(rel(a,c));
  }
```

Finally, the axioms are collected in a list—this allows the tooling to automatically call all axioms with randomized data when testing.

```
  AXIOMS(is_antisymmetric,
         is_total ,
         is_transitive )
};
```

A sample implementation (*model*) of a total order would be the less-than-or-equals operator for integers: total_order< int, op_lt> (this instantiation checks the syntactic requirements). We can state that <= satisfies the total_order concept by specializing the verified type trait (this instantiation records the semantic intent):

```
template <>
struct verified <total_order<int, op_lt>>
  : public std::true_type
{};
```

A sorting library may then state that it requires a verified total order to provide sorting.

3.2 Expressing Axioms in Ada

The same principle of encoding axioms in the Java and C++ programming languages can be applied to Ada. Below we show a generic specification of the less-than-or-equals operator, with the properties stated as assertions within procedures—one for each property.

The setup is similar to Catsfoot, specifying a type and an operation as parameters:

```
generic
  type T is private;
  with function "<="(A, B : T) return Boolean is <>;
```

With Ada's separation of specification and body, the axioms should have descriptive names, so that the behavior is apparent from reading the specification part:

```
package Total_Order is
   procedure LessEq_Is_Antisymmetric(A, B : T);
   procedure LessEq_Is_Total(A, B : T);
   procedure LessEq_Is_Transitive (A, B, C : T);
end Total_Order;
```

The package body provides the actual axioms, in the form of assertions:

```
package body Total_Order is
  procedure LessEq_Is_Antisymmetric(A, B : T) is begin
    if A <= B and B <= A then
      Assert(A = B, "LessEq_Is_Antisymmetric");
    end if;
  end;

  procedure LessEq_Is_Total(A, B : T) is begin
    Assert(A <= B or B <= A, "LessEq_Is_Total");
  end;

  procedure LessEq_Is_Transitive (A, B, C : T) is begin
    if A <= B and B <= C then
      Assert(A <= C, "LessEq_Is_Transitive");
    end if;
  end;
end Total_Order;
```

This generic description of how a less-than-or-equals operator should behave can then be reused in other specifications, such as the specification of sorted lists in the following example: First, a straight-forward description of the interface:

```
generic
  type Elt is private;
  with function "<="(A, B : Elt) return Boolean is <>;
package Sorted_Lists is
  type List is private;

  -- insert E into L
  procedure Insert(L : in out List; E : in Elt);
  -- get length of L
  function Length(L : in List) return Natural;
  -- get element at index I in L
  function Get(L : in List; I : in Natural) return Elt;
  -- count occurrences of E in L
  function Count(L : in List; E : in Elt) return Natural;
```

Next, the semantics. We have split the behavior specification into two parts, Requirements and Axioms, two specially named nested packages.[4] The former is used to specify required semantics for parameters to generic packages, and the latter for specifying the behavior of subprograms provided by the package. For the requirements, we reuse Total_Order:

```
  -- specification of what we require from parameters
  package Requirements is
     new Total_Order(T => Elt);

  -- specification of what we are providing
  package Axioms is
     -- for any I1, I2,  I1 <= I2 = Get(L,I1) <= Get(L,I2)
     procedure List_Is_Sorted (L : List; I1, I2 : Natural);
     -- result of Count increases by one after insert
     procedure Insert_Increases_Count (L : List; E : Elt);
     -- result of Length increases by one after insert
     procedure Insert_Increases_Length (L : List; E : Elt);
  end Axioms;
private
   -- ...
end Sorted_Lists;
```

Code for the axioms is provided in the package body, for example:

```
  -- sample axiom implementation
  procedure List_Is_Sorted (L : List; I1, I2 : Natural) is
    N : Natural := Length(L);
    X1 : Natural := I1 mod N;
    X2 : Natural := I2 mod N;
  begin
    Assert((X1 <= X2) = (Get(L, X1) <= Get(L, X2)));
  end;
```

Parameter behavior and provided behavior can then be tested by instantiating the generic package and calling the axiom procedures with representative data. Tool assistance

[4] This format is chosen for technical reasons related to the Ada package system.

may be helpful for this—in Java and C++, axioms can be tested automatically with random data, using reflection or meta-programming (respectively). A verification or testing tool should be able to find the associated algebraic specification by looking into the nested Requirements and Axioms packages.

The package organization shown here would likely not be supported in SPARK, due to limitations in dealing with generics [34]. Since several SPARK features may be helpful (including preconditions and dependencies), finding a SPARK-compatible axiom scheme would be useful.

3.3 Haskell and QuickCheck

QuickCheck [8] is a popular testing library for Haskell based on algebraic specification. Axioms are normal Haskell boolean functions (called *properties*), and QuickCheck can automatically determine appropriate arguments and call the functions with randomly generated data. For example, the less-than-or-equals specification looks like this (note that forward implication is, confusingly, written with a backwards arrow <=):

```
propLessEqIsAntisymmetric a b =
    (a <= b && b <= a) <= (a == b)

propLessEqIsTotal a b =
    a <= b || b <= a

propLessEqIsTransitive a b c =
    (a <= b && b <= c) <= (a <= c)
```

We can try the tests on various implementations, for example for integers and characters:

```
> quickCheck (propLessEqIsTotal :: Int->Int->Bool)
+++ OK, passed 100 tests.
> quickCheck (propLessEqIsTotal :: Char->Char->Bool)
+++ OK, passed 100 tests.
```

In this case, the properties are tied to the *type class* Ord of ordered types, which also contains other ordering operations, min/max operations and equality. A QuickCheck specification library should provide properties for the complete set of operations.

Properties and type classes are fully decoupled in Haskell, unlike how concepts in JAxT and Catsfoot (and our sketched Ada example) combine an interface with a specification. This means that building more complex specification from simpler ones becomes somewhat more difficult. For example, to deal with the sorted list which requires an element type with a less-than-or-equals operator, we would specify Ord as the type of elements (a purely syntactic requirement). Checking conformance with the axioms requires finding each instantiation and running QuickCheck on the appropriate properties.

4. API AND PRE/POST SPECIFICATIONS

Frameworks that rely solely on pre/post specification have problems capturing the properties of *Comparable*.

The JML [23] specification below[5] exposes these differences. It presents the compareTo example used in subsection 2.1 using pre/post specifications. The specification was written for an older, pre-generic version of Java, so needs to deal with dynamic typing of the second argument. It is also evident that the specification considers being called with null as a precondition violation, rather than being related to throwing a specific exception.

```
Specifications : pure
  public behavior
    requires o != null;
    ensures (* \result is negative
         if this is "less than" o *);
    ensures (* \result is 0 if this is "equal to" o *);
    ensures (* \result is positive
         if this is "greater than" o *);
    signals_only ClassCastException;
    signals (ClassCastException) (* ... *);
  also
  public behavior
    requires o != null &&
             o instanceof Comparable;
    ensures this.definedComparison((Comparable)o,this);
    ensures o == this ==> \result == 0;
    ensures this.sgn(\result ) ==
           -this.sgn(((Comparable)o).compareTo(this));
    signals (ClassCastException)
         !this.definedComparison((Comparable)o,this);
int compareTo(non_null Object o);
```

This specification is more monolithic than the algebraic approach. It is monolithic in the sense that all aspects of the description of the method is included, as opposed to in the axiom case where the specification is composed from individual axioms, each dealing with separate aspects of the library specification. Further, we see that the JML specification uses a more powerful logic than the Java only specification of axioms, e.g., the specification language has a built in equality. Yet there are important shortcomings in this specification: the transitivity property (property 2) is missing, so is property 3. Both of these properties relate three variables, one more than provided as arguments to the function call, going beyond what pre/post easily can express.

JML does have features for dealing with this, through universally quantified assertions at the class level, corresponding to axioms in the algebraic sense—which is how the missing properties are expessed in JML.

In Spec# [6], the specification of IComparable[6] deals only with data flow and purity, and does not attempt to specify the full behavior. This seems to be common for Spec# interface specification, and in general Spec# seems focused on specifying purity, object invariants, data flow and exception behavior.

We believe this difference in capturing API properties is archetypal for the two techniques.

- Algebraic specifications are composed of collections of axioms, each relating one or more operations with an unlimited use of free variables.

  They may be incomplete, i.e., important axioms may be missing from the current specification, yet the specification is still useful.

  This encourages modularity and reuse of specification components.

[5]From http://www.eecs.ucf.edu/~leavens/JML-release/specs/java/lang/Comparable.spec

[6]From https://specsharp.codeplex.com/SourceControl/latest#SpecSharp/Samples/OutOfBandContracts/Mscorlib/System.IComparable.ssc

- Pre/post specifications are extremal points, the first and the last, of assertions for an algorithm. Assertions play important roles in the verification of algorithms, thus making pre/post natural resources for verification of algorithms.

  In order to meet this goal, pre/post specifications tend to be monolithic and should encompass all relevant properties. They often need quite powerful specification logics to express properties.

These differences are amplified by the different goals: on the one side API specifications with axioms for unit testing, on the other specifications for proving algorithms correct.

However, there are many interesting properties that are less straight-forward to specify algebraically or at the API level. We have already mentioned how preconditions are necessary also in algebraic specification. Tools like JML, Spec#, SPARK and Dafny [24] support a number of such properties, e.g., termination, coarse or fine grained data modification, data dependency between inputs and outputs, data ownership, and algorithm-level assertions.

4.1 API Specifications Subsume Pre/Post Specifications

Using the same specification logic for axioms as for pre/post specifications, we find that axioms subsume pre/post specifications. Assume we have a pre/post specification of the following form in a suitable specification logic.

```
method m(...)
  requires Pre;
  ensures Post;
```

We can write this as an axiom in the following way.

```
axiom PrePost (...) {
  if ( ! Pre ) return;
  call m(...);
  assertTrue( Post );
}
```

The if statement checks for precondition violations in the test data, ensuring that the axiom does not commit the method to any specific behavior in such cases. After calling the method, the post condition is asserted to hold. Picking up the pre and postconditions in idioms like this should be straight-forward. Allowing the axioms to use the same specification power as the pre/post specifications, e.g., local quantifiers and ghost variables, we find that pre/post specifications are a special case of axioms.

Tools that support pre/post specifications often include features beyond just Pre and Post. For instance, Dafny [24] has a decreases clause for attaching termination related information to procedures. Such features are not captured by the translation into axioms sketched above.

5. PURITY AND MUTABILITY

A particular problem that arises when mixing specifications with code, is that of what happens when operations have side effects—i.e., they modify or use global data, access the outside world, or change the objects passed as arguments.

This creates a reasoning problem: *how do you reason about a series of assertions/axioms, if each assertion/axiom may have unknown side effects?*—and a programming problem: *what semantic effect does turning assertion checking on or off have if the assertions can have side effects (and similar for axiom checking)?*

Solutions to this problem include forbidding calls to arbitrary functions from specifications (e.g., as in ESC/Java [11]); introduce a notion of purity, and allow only calls to known pure functions (e.g., as in JML); tell the programmer sternly to avoid using non-pure functions and hope for the best (e.g., as in Eiffel).

In Spec# this problem is dealt with through a notion of *observational purity* [7], where side effects are allowed only as long as they can not be observed by the callers. A static analysis tool is available to determine the property.

For practical use in testing, side-effects are not a huge concern. It mostly affects the way axioms are written. The test framework can take care of ensuring that each axiom is evaluated in a fresh environment, and instantiated with suitable (perhaps aged) objects for its universally quantified variables. Some algebraic-style testing frameworks, such as ASTOOT [9], allows axioms with OO notation, and can also deal with side-effects.

The experimental Magnolia language [4] deals with the side-effect issue by avoiding aliasing (hence no globals) and mapping from procedures (which are allowed to have effects on the arguments) to functions (which are strictly pure). Side effects on the outside world are dealt with through updates on a 'world' object. Each procedure is mapped to a set of functions—one function for each possible output / updated argument of the procedure. Specifications (which are algebraic-style and integrated into the language) are written using functions only, and reasoning happens at the level of pure functions. Implementations, however, may be provided in the form of procedures, and through a process of *mutification* pure function-oriented code is rewritten to use procedures with in-place updates of arguments.

6. HISTORICAL BACKGROUND

Both pre/post specifications [12, 21] and algebraic specifications [26] have a long history in computer science.

Pre/post specifications are a natural extension of assertions used for proving and understanding algorithms. An assertion captures properties of the state of a program, e.g., a loop invariant. The last assertion in a procedure defines its postcondition: what holds when the procedure finishes. Symmetrically, the first assertion in the procedure defines its precondition: what needs to hold in order for the procedure to work properly. Already in the 1970s, languages like Gypsy [1], Euclid [30], CLU [25] and Alphard [33] picked up these ideas. Some, including Euclid and CLU, left the specification syntax and processing to external tools, while Alphard made it directly part of the language. Pre/post specifications were popularized as "design by contract" in Eiffel [27], and have gained popularity in recent languages as Spec# [6], JML [23] and Ada [36] with the SPARK [5] tool set for proving correctness.

CLU and Alphard also pioneered the use of abstract data types. ADTs have an internal implementation, a model, and an external API (in current terminology). Algebraic specifications focus on the interaction between functions, making them natural for specifying APIs [26]. Early work tied algebraic specifications closely to software, e.g., [13] using axioms as test oracles. The research on algebraic specifications soon took on a more mathematical approach, lead-

ing to a focus on initial specifications [14, 10]. These are good for building theoretical models, but only have an indirect relationship to software. The LARCH specification language [18] makes this explicit by having a separate algebraic specification language, though not with initial semantics, and explicit programming specific interface languages. Extended ML [31] provided a tight integration between an algebraic specification language and the programming language ML. Although positive experiences were reported [32], there were problems in integrating the semantics of the specifications and of ML.

The Tecton system [22] similarly attempted to leverage algebraic specification; those experiences were later used in the design of C++ concepts [17]—which, if successful, would have provided an industrial-strength language with algebraic specification support. Although axioms played little role in the proposed standard—compilers were for the most part supposed to ignore them, apart from basic syntax and type checking—several initiative attempted to exploit the specifications for optimization [37, 16, 3] and testing [3].

7. CONCLUSIONS

We have discussed the specification of generic APIs and how postconditions fail to deal with this case. Instead, we argue that APIs should be specified through algebraic specification, with axioms that relate the operations of the API to each other. Preconditions still have an important role to play in such specifications, for specifying constraints on arguments—guarded algebras provide a systematic approach to dealing with this.

Though our examples of generic API specifications are primarily related to collection classes, there are similar needs in other domains, e.g., in generic linear algebra codes [16].

An API may be implemented many times. A useful specification should cover all such implementations, and should be useful in determining the correctness of such implementations. This requires a tight integration between the notation and semantics of the specification language and the programming language. Such an integration is readily achieved when axioms are written as tests in the programming language. This provides an immediate approach to using algebraic style axioms for APIs in a high integrity setting.

Proof tools based on pre/post specifications have made huge improvements over that past few decades, so much so that we are approaching the goal of verifying realistic applications. The recent success of such tools gives a vision for developing similar automated proof tools for algebraic style specifications of APIs and the related implementations, possibly on top of existing tools for pre/post specifications.

Tools for proving generic programs seem to be lacking, as up to date tools such as SPARK or Dafny cannot tackle this when requirements on the generic API need to be taken into account. They are however able to prove the correctness of instantiated generic code.

Assertions and invariants in general are useful for reasoning about algorithms and concrete code—regardless of how APIs are specified. The success of assertion-based frameworks such as JML and SPARK are a testament to this.

Acknowledgments

This research is partially financed by the Research Council of Norway, under the DMPL project. Thanks to Eivind Jahren for help in understanding Haskell.

8. REFERENCES

[1] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells. Gypsy: A language for specification and implementation of verifiable programs. In *Proceedings of an ACM conference on Language design for reliable software*, pages 1–10, New York, NY, USA, 1977. ACM.

[2] A. H. Bagge, V. David, and M. Haveraaen. Testing with axioms in C++ 2011. *Journal of Object Technology*, 10:10:1–32, 2011.

[3] A. H. Bagge and M. Haveraaen. Axiom-based transformations: Optimisation and testing. In J. J. Vinju and A. Johnstone, editors, *Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *Electronic Notes in Theoretical Computer Science*, pages 17–33, Budapest, Hungary, 2009. Elsevier.

[4] A. H. Bagge and M. Haveraaen. Interfacing concepts: Why declaration style shouldn't matter. In T. Ekman and J. J. Vinju, editors, *Proceedings of the Ninth Workshop on Language Descriptions, Tools and Applications (LDTA '09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 37–50, York, UK, 2010. Elsevier.

[5] J. Barnes. *SPARK – The Proven Approach to High Integrity Software*. Altran Praxis Ltd, 2012.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[7] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *6th Workshop on Formal Techniques for Java-like Programs (FTfJP'2004)*, 2004.

[8] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.

[9] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[10] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[12] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[13] J. D. Gannon, P. R. McMullin, and R. G. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[14] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–149. Prentice Hall, 1978.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[16] P. Gottschling and A. Lumsdaine. Integrating semantics and compilation: Using C++ concepts to develop robust and efficient reusable libraries. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 67–76. ACM, 2008.

[17] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM.

[18] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Softw.*, 2(5):24–36, 1985.

[19] M. Haveraaen and K. T. Kalleberg. JAxT and JDI: the simplicity of JUnit applied to axioms and data invariants. In *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 731–732, New York, NY, USA, 2008. ACM.

[20] M. Haveraaen and E. G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In *Recent Trends In Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 3–11. Springer-Verlag, 2000.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[22] D. Kapur, D. R. Musser, and A. A. Stepanov. Tecton: A language for manipulating generic objects. In J. Staunstrup, editor, *Program Specification, Proceedings of a Workshop*, Lecture Notes in Computer Science, pages 402–414, Aarhus, Denmark, Aug. 1981. Springer-Verlag.

[23] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[24] 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 - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[25] B. Liskov, R. R. Atkinson, T. Bloom, J. E. B. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[26] B. Liskov and S. Zilles. Specification techniques for data abstractions. In *Proceedings of the international conference on Reliable software*, pages 72–87, New York, NY, USA, 1975. ACM.

[27] B. Meyer. *Eiffel: The language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[28] P. D. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *COMPASS/ADT*, volume 655 of *Lecture Notes in Computer Science*, pages 66–92. Springer, 1991.

[29] D. R. Musser and A. A. Stepanov. Generic programming. In P. M. Gianni, editor, *Symbolic and Algebraic Computation, International Symposium ISSAC'88, Rome, Italy, July 4-8, 1988, Proceedings*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 1988.

[30] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell, and R. L. London. Notes on the design of Euclid. In *Proceedings of an ACM conference on Language design for reliable software*, pages 11–18, 1977.

[31] D. Sannella and A. Tarlecki. Extended ML: An institution-independent framework for formal program development. In *Proceedings of the Tutorial and Workshop on Category Theory and Computer Programming*, pages 364–389, London, UK, 1986. Springer-Verlag.

[32] D. Sannella and A. Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Comput. Surv.*, page 10, 1999.

[33] M. Shaw, W. A. Wulf, and R. L. London. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM*, 20(8):553–564, 1977.

[34] SPARK Team. SPARK Generics – A User View. Technical Report S.P0468.42.25, Altran, January 2012. Draft.

[35] A. Stepanov and P. McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[36] S. T. Taft, R. A. Duff, R. Brukardt, E. Plödereder, P. Leroy, and E. Schonberg. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652/2012 (E)*, volume 8339 of *Lecture Notes in Computer Science*. Springer, 2013.

[37] X. Tang and J. Järvi. Concept-based optimization. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 97–108, New York, NY, USA, 2007. ACM.