Contract-Based Software Development: Immutable Lists and Quantifiers

Kongshøj, Simon

Publication date:
2016

Document Version
Other version

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Download policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Summary

This note explains how contracts can be written to express properties of entire data structures, when methods mutate the contents of those structures. We will see two mechanisms that serve this purpose: Immutable lists, which are derived from functional programming concepts and are used to express data structure contracts in Eiffel, and shallow copies, which are used in many modern contract development tools. Examples are provided in JML.

Introduction

Recall that commands modify the state of some object, whereas queries merely return some state. In good DbC design, the postconditions of a command specify which modifications are done to the object in question. In our stack example, we specified as postconditions for the push operation that the stack size was to grow by one, and that the new top element of the stack was to be the object given as parameter to the push operation.

However, does that tell the whole story of what happens when we push an object onto a stack? In fact it does not: A broken implementation could perhaps destroy the element immediately under the new top element, and our contract would still consider it a correct implementation. Worse, none of the conditions on any other operation would be able to discover this behaviour either, so a rather serious bug could be undetected by our DbC tools.

We need a way to model the state of entire data structures, allowing us to examine elements inside the data structure from our contracts. However, because we don’t want to expose any implementation details in our specifications, we want to do this in a manner that is implementation independent as well. There are several possible mechanisms
that provide such a tool, and they rely on having access to some secondary representation of the data structure in question.

**Immutable Lists**

A simple mechanism for supporting queries on data structure contents is the *immutable list*, a construct that is ubiquitous in functional programming languages such as Lisp, Haskell and F#. In contrast to the more typical object-oriented conception of lists (such as `ArrayList`s and `LinkedList`s from the Java Collections API), immutable lists *cannot be changed*. New lists can be created, perhaps with old lists as sublists, but once a list has been constructed, it can never be modified. In DbC terms, there are thus no *commands* defined on immutable lists, only a variety of *queries*. This makes them quite useful for defining contracts: Recall that the first of the six fundamental principles required us to separate commands from queries specifically because only queries are safe for use in pre- and postconditions. Furthermore, recall that all our pre- and postcondition assertions are *expressions*: They return values, but don’t have side effects. Immutable lists (and other immutable data structures) are specifically designed such that their entire interface is based on expressions.

One way to conceptualize immutable lists comes from the Lisp language, which builds lists from a notion of “cons cells”; pairs of objects which may contain other cons cells — in other words, they can be recursive. For example, consider the 4-element list `(42 69 666 7)`. Although this list can be treated as a single object, it can also be viewed as a pair, consisting of a head and a tail:

\[
H: 42 \quad \rightarrow \quad T: (69 \ 666 \ 7).
\]

The tail of that list is itself a list, which in turn has the head 69 and the tail `(666 7)`, and so forth. The last pair in such a list has a *null object* as its tail (in Lisp, the null object is conventionally written `()`, denoting the empty list), just like linked lists in Java. In fact, if we wish to implement immutable lists in Java, a linked list is often the best data structure to do so with. We will now see how we can make a specification for such a list.
Here is a JML-based interface for an immutable list:

```java
public interface ImmutableList {

    // Basic queries
    public /*@ pure @*/ boolean isEmpty();
    public /*@ pure @*/ Object head();
    public /*@ pure @*/ ImmutableList tail();

    // Derived queries
    public /*@ pure @*/ int size();
    public /*@ pure @*/ ImmutableList precededBy(Object o);
    public /*@ pure @*/ boolean equals(ImmutableList l);
    public /*@ pure @*/ Object item(int i);
    public /*@ pure @*/ ImmutableList sublist(int from, int to);
}
```

The three basic queries should be self-explanatory. They also have very simple specifications:

```java
public /*@ pure @*/ boolean isEmpty();
/*@ requires !isEmpty(); @*/
public /*@ pure @*/ Object head();
/*@ requires !isEmpty(); @*/
public /*@ pure @*/ ImmutableList tail();
```

In other words, the only preconditions on head and tail is that they are not defined on empty lists. This is a typical property of immutable lists in functional languages as well; it is usually an error to attempt to get the head or tail of an empty list. The derived queries are somewhat more involved, and — as is common in constructs from functional programming — make heavy use of recursion. The size query has a straightforward recursive specification:
In other words: The size of an empty list is 0, the size of a non-empty list is one higher than the size of its tail. Since lists are always terminated by the empty list, this recursion is guaranteed to end\(^1\).

The precededBy() query can be used to construct new lists. In other words, assuming that empty is an empty list, we could construct the list from the earlier example thus:\(^2\):

empty.precededBy(7).precededBy(666).precededBy(69).precededBy(42)

Note that this is an expression! The empty list isn’t changed, nor is any of the intermediate lists. If we wanted to put another element into the list, such as the number 0, we could simply chain .precededBy(0) to the end.

The precededBy() query has a very simple specification:

```java
/*@ ensures !(esult).isEmpty();
@ ensures (esult).tail() == this;
@ ensures (esult).head() == o;
@*/
public /*@ pure @*/ ImmutableList precededBy(Object o);
```

The equals operation also has a recursive specification:

```java
/*@ requires l != null;
@ ensures (l.isEmpty() != isEmpty()) ==> !(result);
@ ensures (!isEmpty()) ==> (result == (l.head()==head() &&
@ l.tail().equals(tail())));
@*/
public /*@ pure @*/ boolean equals(ImmutableList l);
```

\(^1\)Lists that are terminated by the empty list are sometimes called *proper lists*, whereas an *improper list* is a list that either has some non-list object in its final tail, or is a circular list. Improper lists make reasoning about (and using!) lists more complicated, and most modern functional programming languages simply don’t have them.

\(^2\)For the paranthetically inclined, in Lisp it would be written as (cons 42 (cons 69 (cons 666 (cons 7 ())))) — or, equivalently, (list 42 69 666 7). Java is a bit more cumbersome for working with immutable lists.
Informally, this specification declares that it's invalid to try to compare the Java null object to an immutable list, and then specifies that two lists are equal if and only if they are both empty, or they are both non-empty and their heads are equal and their tails are equal.

Likewise, we can make a recursive specification for item(), a query which returns the item at some given position in the list:

```java
/*@ requires 0<=i && i<size();
@ ensures (i==0) ==> \result==head();
@ ensures (i>0) ==> \result==tail().item(i-1);
/*@
public /*@ pure @*/ Object item(int i);
```

Can you see from the specification which of L.item(0) or L.item(1) would be equivalent to L.head()?

Finally, the sublist query returns a sublist consisting of all the elements between two positions in the list, in sequence. Here is its specification:

```java
/*@ requires 0<=from && from<=to && to<=size();
@ ensures (from==from) ==> \result.isEmpty()==(to==from);
@ ensures (from!=to) ==> ((\result).head()==item(from));
@ ensures (from!=to) ==> ((\result).tail().equals(sublist(from+1,to)));
@*/
public /*@ pure @*/ ImmutableList sublist(int from, int to);
```

Giving an informal English description of this specification is left as an exercise for the reader — reading specifications and describing them in one's own words is often a good way of learning DbC.

**Example: Queues**

Armed with our newly specified (and, hopefully, implemented!) implementation of immutable lists, we are ready to apply what we know to a slightly more complicated data structure than stacks: Queues. A typical queue presented in an algorithms textbook might have the following basic operations:
This looks quite similar to a basic stack, except that enqueueing an element puts it at the tail of a queue rather than at the top of a stack, giving a FIFO access discipline rather than a LIFO one.

Applying the first of the six principles immediately reveals that the dequeue operation is problematic: It is neither a query nor a command. We thus separate it into two operations: A query head() and a command remove() — just like we did with the stack. To be consistent with remove(), we rename enqueue() to put().

Now we design the pre- and postconditions of the resulting operations. First, consider the newly-introduced remove() operation. It probably makes no sense to remove anything from an empty queue, so we ought to put that in as a precondition. Furthermore, removing something from a queue reduces its size by one, so that ought to be a postcondition. However, is this specification sufficient? In fact it is not: We also want to be able to specify that after removing an element from the queue, the formerly-second element is now the first element, the formerly-third element is now the second, and so forth. We can specify this using an immutable list.

We introduce a new basic query, items(), which returns an immutable list of all the items in a queue:

```java
class Queue {
    public ImmutableList items();
}
```

Using this, we are now able to express a contract for remove():

```java
/*@ requires size()>0;
@ ensures size()==\old(size())-1;
@ ensures \old(items()).tail().equals(items());
@*/
public void remove();
```

Using the immutable list, the second postcondition succinctly expresses the FIFO behaviour! It simply states that the tail of the old queue (that is, the queue before running remove) is equal to the entirety of the new queue, which is equivalent to saying that only the front element has been removed. Because all operations on immutable lists are queries, we can freely use all of them in our specs.

The specification for head() follows directly, given how immutable lists work:
Now that we have `items()`, a question arises: Is `size()` a basic query at all? In fact, it isn’t: It can be expressed in terms of `items()`, since the size of a queue is precisely equivalent to the length of the immutable list of all its items. In other words, it is a derived query, and it has the following contract:

```java
/*@ ensures \result == items().size(); @*/
public /*@ pure @*/ int size();
```

With this change, a buggy `size` that returned a wrong result can be detected.

Finally, we can write a contract for `put()`:

```java
/*@ ensures size()==\old(size())+1;
 @ ensures items().item(size()-1)==o;
 @*/
public void put(Object o);
```

Note that, given `items()`, we don’t have to resort to introducing an `itemAt()` operation like we did with the stack in our previous session.

**Discussion**

A performance-minded programmer will have noticed one distinct drawback to this design: If `items()` constructs an immutable list consisting of all the elements in our queue, then it is a quite expensive operation. In fact, it is at best $O(n)$ in the size of the queue. Since we use it in the contracts of all our basic queue operations, it can be readily concluded that the queue will have **terrible** performance: An operation as simple as `put()` which ought to be $O(1)$, has become an $O(n)$ operation. Clearly, this is unacceptable for any production use of our queue.

However, notice that `items()` is only used in the **postconditions** of our specifications; there is no precondition that refers to it directly\(^3\). This is an important point, due to the role pre- and postcondition checking plays in Dbc-based software development. Checking **postconditions** serves to expose errors in the **implementation** of a class, and is important while a class is being developed. On the other hand, checking **preconditions**

\(^3\)There is an **indirect** use of it because we made `size()` a derived query which uses `items()` in its own postcondition, and several of our preconditions use `size()`
serves to expose errors in the use of a class, on the client’s side. This means that it is.common practice to have precondition checking enabled for all classes during development, but to only have postcondition checking enabled for the class currently under development. If postcondition checking is disabled, then there are no checks that involve items() in our contracts, and then they will have good performance.

This is in fact a useful rule of thumb: Make sure that preconditions can be checked efficiently! In our example, this would mean that size() should have a very efficient implementation, for example simply returning a count attribute. On the other hands, your postconditions can be arbitrarily expensive; they’ll only be checked while you’re developing the class they’re specified for.

Both CodeContract and JML allow disabling postcondition checking, and both tools can also disable all contract checking (in the case of JML, this is done simply by not running the JML preprocessor). Common DbC methodology is to have precondition checking enabled for all classes during development, postcondition checking enabled only for the class currently being developed, and sometimes to have all checking disabled for production code.

Quantifiers and Shallow Copies

Some tools (including both JML and CodeContract) support quantifiers, based on the existential and universal quantifiers from formal logic that you saw in the beginning of the course. When we have quantifiers, immutable lists are not needed. It’s still useful to understand immutable lists, both because they’re often used in DbC literature (they are used in the Eiffel language) and because they’re widely used in functional programming, a programming paradigm currently on the rise.

Quantifiers allow us to express properties on sets of elements, but we still have to be able to have a notion of expressing properties both before and after executing some command. This can be achieved by taking a shallow copy: A copy of the structure, but without copying the actual elements (in other words, simply using references). Taking a shallow copy is also an O(n) operation, but it often has substantially less overhead than constructing a new immutable list.

We can express a contract for taking a shallow copy of our queue thus:

```java
/*@ ensures \result.size()==size();
```

*Programmers familiar with lowlevel languages might complain that it’s inefficient to have a size() method that simply returns the contents of an attribute, and that it’d be better to just use the attribute directly. However, we can’t write contracts for attributes — and there would be no performance benefits of doing so in Java or C# anyway, since the VMs of both languages inline such small methods.*
@ ensures (\forall int i; 0\leq i && i\leq size()-1;
  @   \ (\text{result.get}(i)==\text{get}(i)));
@*/
public Queue shallowCopy();

Note that we use simple == to check for equality, in other words we’re comparing object \textit{references} rather than object \textit{contents}. This contract guarantees that \texttt{shallowCopy()} returns a correct copy of the queue — and you can hopefully clearly see how the above could be expressed in propositional logic.

Given this operation, we can now use the JML \texttt{forall} quantifier to express our contracts. The contracts for the queries are the same as in the version based on immutable lists, but those for the commands are expressed a bit differently. Using \texttt{remove()} as an example:

/*@ requires size()>=1;
@ ensures size()==\texttt{old(size())}-1;
@ ensures (\forall int i; 0\leq i && i\leq size()-2;
  @   \ \texttt{old(shallowCopy())}.get(i+1) == \texttt{get}(i));
@*/
public void remove();

In this contract, we compare against a copy taken \textit{before} the execution of \texttt{remove()}, using \texttt{old} and \texttt{shallowCopy()}. We then use the \texttt{forall} quantifier to express that every element has been moved one position ahead in the queue after the old head was removed.

\textbf{Literature}

Mitchell & McKim: \textit{Design by Contract, by Example}, Addison-Wesley 2002