

Abstraction Functions and Representation Invariants

6.170 Lecture 7

Spring 2004

In this lecture, we describe two important tools for understanding abstract data types: the representation invariant and the abstraction function. The representation invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it. Representation invariants can amplify the power of testing. It's impossible to code an abstract type or modify it without understanding the abstraction function at least informally. Writing it down is useful, especially for maintainers, and crucial in tricky cases.

1 Representation Invariants

A representation invariant, or rep invariant for short, is a constraint that characterizes whether an instance of an abstract data type is well formed, from a representation point of view. Mathematically, it is a formula over the representation of an instance; you can view it as a function that takes objects of the abstract type and returns true or false depending on whether they are well formed:

$RI : \text{Object} \rightarrow \text{boolean}$

Let's consider a version of the Account data type that we discussed in an early lecture. Here's the code:

```
// Account represents a mutable bank account.
class Account {
    private ArrayList transactions;
    private int balance;

    public Account () {
        // effects: makes a new account
        transactions = new ArrayList ();
        balance = 0;
    }

    boolean isAllowed (Trans t) {
        // requires: t != null
        // effects: returns true iff t can be posted to this account
        return (balance + t.getAmount() >= 0);
    }
}
```

```

boolean post (Trans t) {
    // requires: t != null
    // effects: adds t to this account and returns true if t is allowed;
    //           otherwise does nothing and returns false
    if (!isAllowed (t)) return false;
    transactions.add (t);
    balance += t.getAmount();
    return true;
}
}

```

The representation invariant (often called *rep invariant* for short) is a constraint that should hold for every instance of the type. What is the rep invariant for Account?

- First, the bank doesn't want the balance of an account to drop below 0. That's the whole point of the `isAllowed` method. So one of the terms in the rep invariant should be

$$\text{balance} \geq 0$$

- The balance should also correctly reflect all the transactions that have been posted to the account; we don't want the auditors to come through at the end of the month and find a discrepancy. This term might be written:

$$\text{balance} = \sum_i \text{transactions.get}(i).\text{amount}$$

- So much for semantic constraints (constraints imposed by the domain, the world of banking). We also have some implementation-related constraints — constraints that ensure that the code we've written will work correctly, without producing any unexpected errors. For one thing, the `post` method freely calls a method on the `transactions` object. If we want to be sure that `post` never throws a `NullPointerException`, then we need

$$\text{transactions} \neq \text{null}$$

A rep invariant often includes a term like this for every object reference in the representation. It's far easier to code if you don't have to worry about checking for null all the time.

- Finally, we should also say something about the contents of the `transactions` list: that each object in the list should be a `Trans`. It isn't strictly needed for this code, since we never actually look at the objects we put in `transactions`. But clearly this `Account` isn't an adequate abstract data type. We'll eventually have to add some observers for retrieving the transactions, and those observers will care about this invariant:

$$\forall i, \text{transactions}[i] \text{ instanceof } \text{Trans}$$

Incidentally, this requirement also covers the case of null references in the `transactions` list, since in Java, `null instanceof C` is false for any class `C`.

Notice that we didn't bother listing `transactions instanceof List` in the rep invariant. We've already said that in the declaration of `transactions`, and furthermore Java's static typing rule guarantees that it will be true, so we don't have to check it. But static typing doesn't help with the elements of the `transactions` list, because the elements of a `List` have declared type `Object`. We have to use the rep invariant to say something stronger – not only are those elements `Objects`, they should in fact be `Trans` objects.

Let's summarize our rep invariant informally, the way you might see it in a source code comment:

```
/*
  Rep invariant is:
    balance >= 0
    balance = sum (for all i) of transactions[i].amount
    transactions != null
    for all i, transactions[i] instanceof Trans
*/
```

Here's a more mathematical way to write the invariant, that makes it very clear that the rep invariant is a boolean function on a particular Account object:

$$\begin{aligned} RI(a) = & a.balance \geq 0 \\ & \wedge a.balance = \sum_i transactions.get(i).amount \\ & \wedge a.transactions \neq null \\ & \wedge \forall i, transactions[i] \text{ instanceof } Trans \end{aligned}$$

Finally, here's a method that actually reduces the rep invariant to code:

```
private void checkRep ()
// effects: nothing if this satisfies rep invariant;
//          otherwise throws an exception

if (balance < 0) throw new RuntimeException ();

if (transactions == null) throw new RuntimeException ();

int b = 0;
for (int i = 0; i < transactions.size(); i++) {
    Trans t = (Trans) transactions.get(i);
    b += t.getAmount();
}
if (balance != b) throw new RuntimeException ();
}
```

This checkRep method examines the representation at runtime and checks that it satisfies the rep invariant. This is even more effective than writing the rep invariant down in a comment, if we judiciously sprinkle calls to this method throughout the implementation. Calls to checkRep can be placed at the start and end of every public mutator or producer and at the end of every creator.

Checking rep invariants is especially useful because it helps to localize bugs. Suppose you forget to update the balance field in a new mutator operation in Account. This bug may not cause problems until a

subsequent operation tries to make use of `balance`, and even then some operations may succeed. When an operation finally fails, it may be much much later, and the bug will be obscure. It will take some debugging to discover that there is no fault with the operation that finally revealed the faulty `balance`, but that it was using a badly formed object. With checks on the rep invariant inserted, however, the bug would be noticed as soon as the offending operation executed, and the programmer's attention would be drawn to the operation that actually contains the error, not the operation that first fails.

2 Modular Reasoning

The rep invariant makes modular reasoning possible. To check whether an operation is implemented correctly, we don't need to look at any other methods. Instead, we appeal to the principle of induction. We ensure that every creator creates an object that satisfies the invariant, and that every producer and mutator preserves the invariant: that is, if given an object that satisfies it, it produces one that also satisfies it. Now we can argue that every object of the type satisfies the rep invariant, since it must have been produced by a constructor and some sequence of mutator or producer applications.

To see how this works, let's look at the operations of the `Account` class. Here's the creator:

```
public Account () {
    // effects: makes a new account
    transactions = new ArrayList ();
    balance = 0;
}
```

Does this constructor construct an `Account` object that satisfies the rep invariant? Clearly, yes: `balance` is greater than or equal to zero; that zero balance is (vacuously) equal to the sum of the amounts in the empty `transactions` list; `transactions` is non-null; and `transactions` (vacuously) contains only `Trans` objects.

How about the observer, `isAllowed`? Like (most) observers, `isAllowed` doesn't modify the representation at all, so it trivially preserves the rep invariant. (Some observers *do* modify the representation, however, without losing the right to be called observers! We'd have to check such an observer more carefully. An example of this phenomenon is coming up at the end of this lecture.)

Finally, we have a mutator:

```
boolean post (Trans t) {
    // requires: t != null
    // effects: adds t to this account and returns true if t is allowed;
    //           otherwise does nothing and returns false
    if (!isAllowed (t)) return false;
    transactions.add (t);
    balance += t.getAmount ();
    return true;
}
```

To check this method, we assume the rep invariant holds on entry. Our task is to show that it also holds on exit. There are two paths through the `post` method: if `isAllowed` rejects the transaction, then `post` returns immediately without modifying the rep, so the rep invariant is preserved in this case.

What if `isAllowed` accepted the transaction? Then it must be the case that `balance + t.amount` will not fall below zero, so that part of the rep invariant is satisfied. Furthermore, we add `t.amount` to both `balance` and to the sum of `transactions`, so the second term of the invariant is still satisfied. We haven't reassigned `transactions`, so it still refers to a non-null object. Finally, the object we add to `transactions` is guaranteed to be an instance of `Trans`. (Actually, the code is a little unsafe here — what if `t` is null? Could that screw up our rep invariant? Luckily, `isAllowed` calls `t.getAmount()`, which would throw a `NullPointerException` without modifying the rep. But it would be safer for `post` to compare `t` with null and throw the `NullPointerException` itself.)

Let's say that we can prove that the rep invariant holds for each constructed object. Further assume that we can prove that the rep invariant holds for every object obtained as a result of a mutator or producer, provided the invariant holds for all objects given to the mutator or producer. Is there any reason to check the invariant at runtime, using the `checkRep` method we wrote above? Yes, because rep exposure may result in ill-formed objects, when code outside of the type's methods, modifies the object.

In this case, we can satisfy ourselves that there's no rep exposure in `Account`, because its fields are marked `private`, and the only mutable object in the representation (`transactions`) is never passed into or leaked out of the class. But some future maintainer of this code may introduce a rep exposure. So even if you are completely confident in your inductive reasoning, it still makes sense to check the invariant explicitly in every method.

To summarize, here's the rule we're using to prove invariants:

Structural induction: If an invariant of an abstract data type is (1) established by creators; (2) preserved by producers, mutators, and observers; and (3) no rep exposure occurs, then the invariant is true of all instances of the abstract data type.

3 More About Rep Exposure

The notion of rep invariants that we have espoused offers a systematic method for checking the correctness of abstract data type implementations. (We haven't yet considered how to ensure that a creator, mutator or producer generates the *right* instance of a type, only that it produces a well-formed instance. When we study abstraction functions later in this lecture, we'll look at that issue.)

Our method says that we can consider the operations one by one, and then appeal to induction to show that every instance will be well-formed. A crucial aspect of this method is local reasoning: we can examine the operations individually, and certainly don't need to look at client code.

But this method is not always sound. It has a proviso: that the representation must not be exposed. Representation exposure is a nasty problem, because it can arise unexpectedly, and have disastrous effects that are hard to pin down.

The simplest form of rep exposure involves allowing client code to manipulate the representation directly. We saw that this is easy to rule out by making all fields `private`, so we don't usually even regard it as a kind

of exposure.

Instead, the rep exposure we'll be concerned with arises because a mutable object inside the representation is accessible from the outside, through a different path. Two common ways in which this happens are:

- when a reference to a mutable object in the rep is returned as the result of an operation;
- when a reference to a mutable object is passed in and made part of the rep, despite being accessible as an existing reference from the outside.

We saw both these cases last lecture in the `Trans` class. Here's another example, which is actually quite common: a method that returns a collection. When the representation already contains a collection object of the appropriate type, it is quite tempting to return it directly. For example, `List` has a method `toArray` that returns an array of elements corresponding to the elements of the list. If we had implemented the list itself as an array, we might just return the array itself. But if the rep invariant requires some other part of the rep to be related to the array (e.g., a `size` field that indicates how many of the entries in the array are actually in use), then a modification to this array may break the invariant:

```
a = p.toArray (); // exposes the rep
a[i] = null; // ouch! breaks invariant
p.get(i); // now behaves unpredictably
```

Once the invariant is violated, all hell breaks loose: subsequent operations may behave in arbitrary ways.

A more subtle variant of this problem arises with iterators. Many Java classes have a method that returns an iterator. In fact, it's often a better design decision to provide a method that returns an iterator over a collection, rather than the collection itself, to save the cost of making defensive copies. But building an iterator is a lot of work for the implementor, so we might be tempted to use one that's already provided by the Java library. Suppose our representation includes an `ArrayList` field `list` that holds a collection of elements, and we want to implement a method

```
public Iterator elements()
```

that returns an iterator over those elements. Noticing that the `ArrayList` interface provides its own method that returns an iterator, we implement our method like this:

```
public Iterator elements() {
    return list.iterator();
}
```

Unfortunately, this too is a rep exposure! The `Iterator` interface in Java includes an optional `remove` operation that allows the client to remove elements from the underlying collection. `ArrayList` implements this operation, so the result of this method is an iterator object that can actually affect the `list` collection, outside the abstract type.

Fortunately there's a solution this problem:

```

public Iterator elements() {
    return Collections.unmodifiableList(list).iterator();
}

```

The `unmodifiableList` method in `Collections` creates a *wrapper* around the list you give it, which forbids all mutations through the wrapper — including mutations by `Iterator.remove`.

4 Element Equality and Rep Exposure

Rep exposure is actually a very subtle notion, because it is not always clear what belongs to the rep. Is it a rep exposure for list operations to return elements of the list? Let's see why it might be.

Suppose our list representation has the invariant that there are no duplicates (e.g., because the list actually represents a set). Furthermore, let's say that our notion of equality is based on the contents of the elements. For example, if the elements are themselves lists, we'll regard two elements to be equal if they contain the same sequence of elements. Now we have a rep exposure: we can create duplicates in the set simply by modifying the elements of the list from outside, making two equal when they were not equal previously.

The root of the problem here is not the passing in or out of the element objects: that can't be avoided. Rather, it's the notion of *equality*. If our set determined equality of the elements using reference equality, so that two elements are equal only when they are the same object, then the rep exposure would not arise. That would itself lead to problems though, since it would result in two strings that represent the same sequence of characters being regarded as distinct. The best approach, therefore, is to have the set call the `equals` method of the element type, and for `equals` of every mutable type to be reference equality.

Unfortunately, the collection classes in Java are not designed in this way. Two `LinkedList` objects, for example, are regarded as equal if they contain equal elements, even if they are distinct list objects. If we insert such lists into a `HashSet`, a subsequent modification to a list breaks the set's invariant. This can lead to weirdly inconsistent behavior, such as the set's iterator returning an element that contains claims isn't even in the set:

```

LinkedList k = new LinkedList ();
HashSet s = new HashSet ();

s.add (k); // put the list in the set
s.contains (s.iterator().next()); // returns TRUE (as it should)

k.add ("deeb"); // mutate the list; breaks the rep invariant of s
s.contains (s.iterator().next()); // returns FALSE!

```

To work around this problem, you should either wrap these kinds of objects in a class that uses reference equality for `equals` before you insert them into a hash set or hash map, or you should make sure you never mutate them. We'll see more about equality problems in the next lecture.

5 Two Spaces: Rep and Abstract

We now take a deeper look at the theory underlying abstract data types. This theory is not only elegant and interesting in its own right; it also has immediate practical application to the design and implementation of abstract types. If you understand the theory deeply, you'll be able to build better abstract types, and will be less likely to fall into subtle traps.

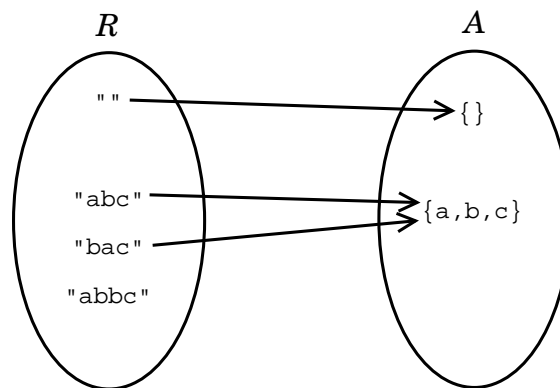
In thinking about an abstract type, it helps to consider the relationship between two spaces of values.

The space of *rep* or *representation* values consists of the values of the actual implementation entities. In simple cases, an abstract type will be implemented as a single object, but more commonly a small network of objects is needed, so this value is actually often something rather complicated. For now, though, it will suffice to view it simply as a mathematical value.

The space of *abstract* values consists of the values that the type is designed to support. These are a figment of our imagination. They're platonic entities that don't exist as described, but they are the way we want to view the elements of the abstract type, as clients of the type. For example, an abstract type for unbounded integers might have the mathematical integers as its abstract value space; the fact that it might be implemented as an array of primitive (bounded) integers, say, is not relevant to the user of the type.

Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor's job to achieve the illusion of the abstract value space using the rep value space.

Suppose, for example, that we choose to use a string to represent a set of characters. Then these form our two value spaces. We can show the two value spaces graphically, with an arc from a rep value to the abstract value it represents:



There are several things to note about this graph:

- Every abstract value is mapped to. The purpose of implementing the abstract type is to support operations on abstract values. Presumably, then, we will need to be able to create and manipulate all possible abstract values, and they must therefore be representable.
- Some abstract values are mapped to by more than one rep value. This happens because the representation isn't a tight encoding. There's more than one way to represent an unordered set of characters as a string.
- Not all rep values are mapped. In this case, the string ""abbc"" is not mapped. If the type of the

rep is nontrivial, it will not make sense to give an interpretation for all rep values. A doubly-linked list representation, for example, can be twisted into all kinds of pretzel configurations that won't correspond to simple sequences, and for which we won't want to write special cases in the code. Or sometimes we will want to impose certain properties on the rep to make the code of the operations more efficient or easier to write. In this case, we have decided that the array should not contain duplicates. This will allow us to terminate the `remove` method when we hit the first instance of a particular character, since we know there can be at most one.

In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite. So we describe it by giving two things:

- An *abstraction function* that maps rep values to the abstract values they represent:

$$AF : \mathcal{R} \rightarrow \mathcal{A}$$

The arcs in the diagram show the abstraction function. In the terminology of functions, the properties we discussed above can be expressed by saying that the function is onto, not necessarily one-to-one, and often partial.

- A *rep invariant* that maps rep values to boolean:

$$RI : \mathcal{R} \rightarrow \text{boolean}$$

For a rep value r , $RI(r)$ is true if and only if r is mapped by AF . In other words, RI tells us whether a given rep value is well-formed. Alternatively, you can think of RI as a set: it's the subset of rep values on which AF is defined.

A common confusion students have about abstraction functions and rep invariants is that they imagine that they are determined by the choice of rep and abstract value spaces, or even by the abstract value space alone. If this were the case, they would be of little use, since they would be saying something redundant that's already available elsewhere.

It's easy to see why the abstract value space alone doesn't determine AF or RI : there can be several representations for the same abstract type. A set of characters could equally be represented as a string, as above, or as a bit vector, with one bit for each possible character. Clearly we need to separate functions to map these two different rep value spaces.

It's less obvious why the choice of both spaces doesn't determine AF and RI . The key point is that defining a type for the rep, and thus choosing the values for the space of rep values, does not determine which of the rep values will be deemed to be legal, and of those that are legal, how they will be interpreted. Rather than deciding, as we did above, that the strings have no duplicates, we could instead allow duplicates, but at the same time require that the characters be sorted, appearing in nondecreasing order. This would allow us to perform a binary search on the string and thus check membership in logarithmic rather than linear time. Same rep value space — different rep invariant.

Even with the same type for the rep value space and the same rep invariant RI , we might still have different interpretations AF . Suppose RI admits any string of characters. Then we could define AF , as above, to interpret the array's elements as the elements of the set. But there's no *a priori* reason to let the rep decide

the interpretation. Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string "acgg" represents the set $\{a, b, c, g\}$.

The essential point is that designing an abstract type means not only choosing the two spaces — the abstract value space for the specification and the rep value space for the implementation — but also deciding what rep values to use and how to interpret them.

6 A Nifty Abstraction Function

You might get the impression that abstraction functions state the obvious: that just looking at the rep, you could guess how it should be interpreted. Much of the time, this is actually true, and for this reason abstraction functions are less important than rep invariants.

Sometimes, however, a clever representation may have an interpretation that is far from obvious. In this case, an abstraction function is a very useful bit of documentation.

Suppose you want to build a Queue datatype using immutable lists. It's not obvious how to implement this efficiently. We know how to implement an immutable list: the cons operation simply creates a new list whose tail is the old list, and the car and cdr operations do the opposite, breaking the list into the first element and the tail. The snag with a queue is that the elements go on one end and come off the other.

A very clever solution is to employ a pair of immutable lists. (This idea is well-known in the functional programming community.) The field declarations in Java may look something like this:

```
class Queue {
    Cons front;
    Cons back;
    ...
}
```

The queue is broken in two. Elements at the front of the queue appear in the list front in natural order. Elements at the back of the queue appear in the list back, in reversed order. We've just described the abstraction function. A bit more precisely,

$$AF(q) = q.front \ ^ \ reverse(q.back)$$

We are assuming these lists represent mathematical sequences, and that reverse is a function that reverses a sequence, and ^ is the concatenation operator. We are also assuming that, in our abstract model of a queue, we think of the elements as ordered so that the first element to be removed will be at the beginning of the list.

Here's how the rep is manipulated. To enqueue an element, we simply cons it to the back list. To dequeue an element, we take it off the front list using car and cdr if the list is non-empty. If it's empty, we reverse the back list, and make it the front list, and replace the back list by the empty list. Here's an example:

Assume the queue is initially empty with the back list and the front list being empty. If A, B and C are enqueued, the back list becomes CBA and the front list is still empty. If we wish to dequeue, then the back

list is reversed to create an ABC front list, the back list is set to empty, and A is dequeued by taking the car of the front list. At this point enqueueing D and E results in a back list of ED and a front list of BC. The abstraction function AF will interpret this concrete configuration as the abstract queue BCDE.

Reversing the list takes time proportional to its length. So, if a lot of enqueue operations have been performed without an intervening dequeue, a single dequeue operation may take some time. But note that each element can only participate in one reversal. The total cost of the reversals over the life of the queue is proportional to the number of elements dequeued. So the cost of each operation is, averaged over all the operations, constant time. This kind of analysis is called an *amortized analysis*, because the cost of a single operation is amortized (spread out) over all operations.

7 Benevolent Side-Effects

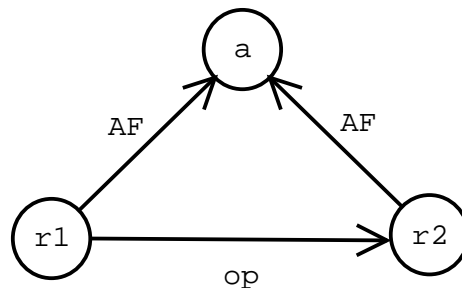
Now suppose we want a `get` operation for the queue — an observer that gives us the object at the head of the queue but doesn't remove it. If the front list is empty, we'll have to reverse the back list to find the object we want. But we only want to do this reversal once, not every time `get` is called. So we really want to write this:

```
Object get () {
  if (front == null) {
    front = reverse (back);
    back = null;
  }
  return front.car();
}
```

But wait — `get` is supposed to be an observer, and here it is assigning fields in the queue. Can we do that?

What is an observer operation? In the last lecture, we defined it as an operation that does not mutate the object. With the theoretical machinery we've developed in this lecture, we can now give a more liberal definition.

An observer operation may mutate an object of the type so that the fields of the representation change, as long as the abstract value it denotes doesn't change. We can illustrate this phenomenon in general with a diagram:



The execution of the operation op mutates the representation of an object from $r1$ to $r2$. But $r1$ and $r2$ are mapped by the abstraction function AF to the same abstract value a , so the client of the datatype cannot

observe that any change has occurred. This kind of mutation is called a *benevolent side-effect*. Because a benevolent side-effect changes only the representation, and not the abstract value visible to a client, we don't need to mention it in the specification for the operation. So `no modifies` this clause is needed for `get`.

Why would you want to make such a mutation? Usually the reason is to improve performance. That's why we did it in the queue, so that `get` could be called repeatedly without having to reverse the back list each time. As another example, suppose clients of a table datatype often look up the same key repeatedly. It might then make sense to cache the key and its value as a hint when a `get` is performed. On a subsequent `get` the first thing you do is to check to see if the key is the one that was just looked up; if so, you can return the value without doing a full lookup. The point is that the client of the datatype cannot see that the value just obtained has been cached (except perhaps by noticing an improvement in performance). All the client cares about is that `get` is an observer in the sense that one `get` can't affect the value obtained by a later `get`.

In general, then, we can allow observers to mutate the rep, so long as the abstract value is preserved. We will need to ensure that the rep invariant is not broken, and if we have coded the invariant as a method `checkRep`, we should insert it at the start and end of observers.

8 Specification Fields

The abstract values of many abstract data types have a tuple structure at the top level. For example, a line is a pair of points; a mailing address is a number, a street, a city and a zipcode; a URL is a protocol, a host name, and a resource name.

In these cases, one can specify a single function that maps representation objects to tuples. This is the approach followed by our textbook. But it's convenient, and perhaps more natural, to break the function into several separate functions, each viewed as defining a *specification field*.

For example, we might represent the `Card` data type, used in a poker game program, by a single integer in a field `index`. The rep invariant requires that `index` be in the range `0...51`. We might have two specification fields defined as follows:

```
c.suit = S(c.index div 13)
c.val  = V(c.index mod 13)
```

where

```
S(0) = Hearts
S(1) = Spades
S(2) = Clubs
S(3) = Diamonds
```

```
V(1) = Ace
V(2) = 2
...
V(11) = Jack
```

```
V(12) = Queen
V(0) = King
```

so that a `Card` object with `index` field of 3, for example, would correspond to the Three of Hearts, and 14 would correspond to the Ace of Spades. This abstraction function maps each representation object `c` to a pair $(c.suit, c.val)$, but rather than writing it as a single function, we've specified it as two separate ones, one for each specification field.

This scheme is so convenient that we'll use it even when there is only one specification field. For example, when we refer to the i th element of a list `v` as `v.elts[i]`, we are using a specification field `elts` whose value is a mathematical sequence. It allows us to talk about the elements of a list without mentioning the representation. Without the specification field, we would have to write $AF(v)$ to denote the list's element sequence, to distinguish it from the value of `v` itself – a Java object reference.

Our abstraction function for our queue now becomes

```
q.elts = q.front.elts ^ reverse (q.back.elts)
```

Note that there are two different `elts` fields here: the one on the left corresponding to the abstraction function of queues, and the two occurrences on the right corresponding to the abstraction function of lists. For annotating code, specification fields are especially convenient. Using the convention that we can omit explicit mention of `this`, we might write, for example

```
/*
  abstraction function
    elts: sequence of elements in queue,
          in order from first to last out
    elts = front.elts ^ reverse(back.elts)
*/
```

as a comment in the `Queue` class.

9 Summary

The abstraction function specifies how the representation of an abstract data type is interpreted as an abstract value. Together with the representation invariant, it allows us to reason in a modular fashion about the correctness of an operation of the type.

In practice, abstraction functions are harder to write than representation invariants. Writing down a rep invariant is always worthwhile, and you should always do it. Writing down an abstraction function is often useful, even if only done informally. But sometimes the abstract domain is hard to characterize, and the extra work of writing an elaborate abstraction function is not rewarded. You need to use your judgement.