Lecture 8: Abstraction Functions

This is the second of a pair of lectures on the theory underlying abstract data types, which provides two key tools for reasoning in a modular way about a data abstraction. Last time we looked at *rep invariants*; this time we look at *abstraction functions*. It's crucial that you understand the concept of the abstraction function, since it lies at the heart of what data abstraction is all about. But it's less important that are proficient in its use. Writing down the abstraction function is a big help when faced with an intricate rep, but for simpler reps can often be dispensed with. In contrast, it's always worth writing down the rep invariant.

8.1 Judging Correctness

Recall from last lecture the mutator method add, which takes an element and adds it to the end of the list:

```
8.1.1   public void add (Object o) {
8.1.2     Entry e = new Entry (o, header. prev, header);
8.1.3     e.prev.next = e;
8.1.4     e.next.prev = e;
8.1.5     }
```

We checked that this operation preserved the rep invariant. It produces a *good* list. But does it produce the *right* list? It correctly splices a new entry into the list, but does it splice it into the right position? Is the new element inserted into the start or the end of the list? It looks as if it's at the end, but that assumes that the order of entries corresponds to the order of elements. It would be quite possible (although perhaps a bit perverse) to store the elements in the reverse order: that is, for a list l with elements o1, o2, o3, to have

```
8.1.6   p.header.next.element = o3
8.1.7   p.header.next.next.element = o2
8.1.8   p.header.next.next.element = o1
```

To resolve this question, we need to know how the representation is *interpreted*: that is, how to view an instance of LinkedList as an abstract sequence of elements. This is what the *abstraction function* provides. Recall (from 7.1) that the abstraction function maps values of the rep space to values of the abstract space: in this case, values of LinkedList to abstract lists. To say that we expect elements to appear in the LinkedList structure in their natural order, we might write

```
8.1.9   A(l) =
8.1.10    if l.header.next = l.header then
8.1.11      the empty sequence
```

```
8.1.12        else
8.1.13            the sequence
8.1.14            <p.header.next.element, p.header.next.next.element, ...
8.1.15            ending with the first entry e such that e.next = l.header
```

We've given the abstraction function informally. It's possible to write it more precisely in a mathematical notation, but for 6170 informal descriptions will suffice.

Note that, in addition to the abstraction function, we also need the *specification* of the method. If it had said that the element should go at the start rather than at the end, the code would be wrong. The abstraction function and the spec go together, since they link the code to the abstract view of the type seen by the client.

The rep invariant, in contrast, can be used without any reference to the spec. This is a merit of sorts: it means that just writing down the rep invariant, and nothing else, is very useful documentation of a type.

8.2 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. (This assumes, by the way, that a human being is interpreting the rep. If you want to build a tool that analyzes code automatically for compliance with its spec – even just that modifications are within the scope permitted by a modifies clauses – you will need to provide the tool with an abstraction function.)

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 whose objects are immutable. 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; I learnt it from Bob Harper of Carnegie Mellon.) The field declarations in Java may look something like this:

```
8.2.1     class ImmutableQueue {
8.2.2         ImmutableList back;
8.2.3         ImmutableList front;
8.2.4         ...
8.2.5         }
```

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

```
8.2.6  A(q) =
8.2.7     A(q.back)) ^ rev(A(q.front))
```

Note that the definition uses the abstraction function on lists. I'm assuming these lists represent mathematical sequences, and that rev is a function that reverses a sequence, and ^ is the concatenation operator. I'm 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 *end* 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.

Reversing the list takes time proportional to its length. So, if a lot of queues 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' over all operations.

8.3 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 a Card datatype, used in card game program, by a single integer in a field index. The rep invariant requires index be in the range 0.. 51. We might have two specification fields defined as follows:

```
8.3.1  c.suit = S(c.index div 13)
8.3.2  c.val = V (c.index mod 13)
8.3.3  where
8.3.4     S(0) = Hearts, S(1) = Spades, S(2) = Clubs, S(3) = Diamonds
```

```
8.3.5        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; 14 corresponds 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. We've actually seen this many times before. When we referred to the ith element of a vector v as v.elts[i], this used a specification field elts whose value is a mathematical sequence. It allowed us to talk about the elements of the vector without mentioning the representation. Without the specification field, we would have to write A(v) to denote the vector's element sequence, to distinguish it from the value of v itself – a Java object reference.

Our abstraction function for our immutable queue now becomes

```
8.3.6    q.elts = q.back.elts ^ rev (q.front.elts)
```

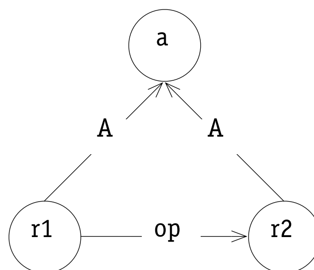
Note that there are two different elts fields here: the one on the left corresponding to the abstraction function of queues, and the two occurrences on the right corresponding to the abstraction function of lists. For annotating code, specification fields are especially convenient. Using the convention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

```
8.3.7    abstraction function
8.3.8       elts: sequence of elements in queue, in order from last to first out
8.3.9       elts = back.elts ^ rev(front.elts)
```

as a comment in the Queue class.

8.4 Benevolent Side Effects

What is an observer operation? In our introductory lecture on representation independence and data abstraction, we defined it as an operation that does not mutate the object. We can now give a more liberal definition.



An operation may mutate an object of the type so that the fields of the representation change, while maintaining the abstract value it denotes. 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 `A` to the same abstract value `a`, so the client of the datatype cannot observe that any change has occurred.

Why would you want to make such a mutation? Usually the reason is to improve performance. 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 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 can't see that the value just obtained has been cached; all it cares about is that `get` is an observer in the sense that one `get` can't effect the value returned 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.

8.5 Idioms for Expressing Abstraction Functions

Writing abstraction functions is difficult, because we don't have a language for talking about abstract values. Such a language, called a *formal specification language*, would also allow us to specify our operations more precisely. But in practice, formal specification languages are not easy to use, so they are usually reserved for critical projects.

Nevertheless, with a little practice, it's possible to write fairly clear and simple abstraction functions. Here are some tips.

First, figure out what the abstract values looks like. If an abstract value is a tuple of several values, examine each of these values and define a separate specification field for it. Can the abstract value be described with standard mathematical sets, sequences and relations? If so, you're in luck: you can now use whatever notation you're comfortable with for constructing and manipulating those mathe-matical values. If not, you might decide to specify the abstraction function by example: that's what I did above with the auxiliary functions S and V in the CardSet example, lacking a standard way to model values such as 'Hearts'.

Second, pick a strategy for mapping concrete objects to the abstract values you've chosen. Here are some common idioms:

- *Comprehension*. The set comprehension expression {x | P(x)} denotes the set of all elements x that satisfy the property P. Suppose we have a set of integers in the range 0..255 represented as an array of booleans (ie, a bit string) stored in a field bitarr. Then we might write the abstraction function like this:

```
8.5.1    s.elems = {i | s.bitarr[i]}
```

- *Recursion*. Recursive representations are usually best handled with recursive abstraction functions. Suppose we have a set represented as a binary tree, with fields val (the value of the tree node, null if the tree is empty), left (the left tree, null for a leaf) and right (the right tree, null for a leaf). Then we might write the abstraction function like this:

```
8.5.2    s.elems = if (t.val = null) then {} else {t.val}  + t.left.elems + t.right.elems
```

where + is set union.

- *Projection*. Sometimes it's easiest to write a formula that relates a part, or a 'projection', of the concrete object to a part of the abstract value. Suppose we have a sequence datatype represented as an array stored in a field eltarray, with a field max giving the index of the highest array element that corresponds to an element of the abstract sequence. Then, using a specification field elts that gives the abstract sequence of elements, we might write the abstraction function as two constraints:

```
8.5.3    size (s.elts) = s.max + 1
8.5.4    for i: 0..size (s.elts) | s.elts [i] = s.eltarray [i]
```

- *By Example*. Finally, you can always fall back on simply illustrating the abstraction function on an example, and hoping that the reader infers the generalization correctly. This is what I did when showing the abstraction for LinkedList:

```
8.5.5    l.elts =
8.5.6       if l.header.next = l.header then
8.5.7          the empty sequence
8.5.8       else
8.5.9          the sequence
8.5.10         <p.header.next.element, p.header.next.next.element, ...
8.5.11         ending with the first entry e such that e.next = l.header
```

8.6 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 judgment.