Abstraction Functions

6.170 Lecture 9

Fall 2005

8.1 Introduction

In this lecture, we turn to our second tool for understanding abstract data types: the abstraction function. The rep invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it. 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.

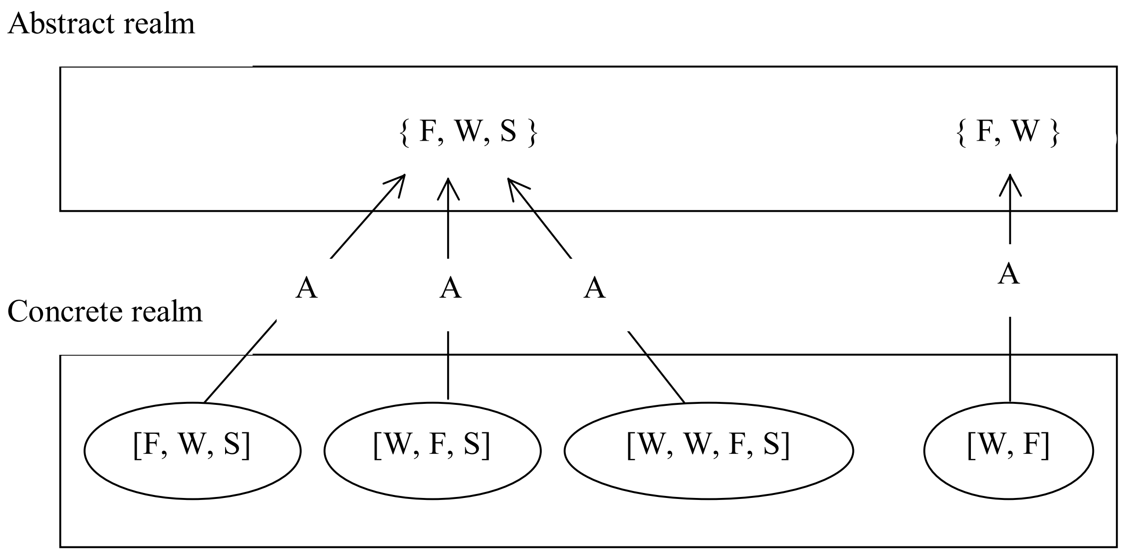
8.2 Abstract and Concrete Objects

In thinking about an abstract type, it helps to imagine objects in two distinct realms. In the concrete realm, we have the actual objects of the implementation. In the abstract realm, we have mathematical objects that correspond to the way the specification of the abstract type describes its values.

Suppose we're building a program for handling registration of courses at a university. For a given course, we need to indicate which of the four terms `Fall`, `Winter`, `Spring` and `Summer` the course is offered in. In good MIT style, we'll call these `F`, `W`, `S` and `U`. What we need is a type `SeasonSet` whose values are sets of seasons; we'll assume we already have a type `Season`. This will allow us to write code like this:

```
  if (course.seasons.contains (Season.S)) ...
```

There are many ways to represent our type. We could be lazy and use `java.util.ArrayList`; this will allow us to write most of our methods as simple wrappers. The abstract and concrete realms might look like this:



The oval below labelled `[F,W,S]` denotes a *concrete* object containing the array list whose first element is `F`, second is `W`, and third is `S`. The oval above labelled `{F,W,S}` denotes an *abstract* set containing three elements `F`, `W` and `S`. Note that there may be multiple representations of the same abstract set: `{F, W, S}`, for example, can also be represented by `[W,F,S]`, the order being immaterial, or by `[W,W,F,S]` if the rep invariant allows duplicates. (Of course there are many abstract sets and concrete objects that we have not shown; the diagram just gives a sample.)

The relationship between the two realms is a function, since each concrete object is interpreted as at most one abstract value. The function may be partial, since some concrete objects -- namely those that violate the rep invariant -- have no interpretation. This function is the *abstraction function*, and is denoted by the arrows marked *A* in the diagram.

Suppose our `SeasonSet` class has a field `eltlist` holding the `ArrayList`. Then we can write the abstraction function like this:

```
  A(s) = {s.eltlist.elts [i] | 0 <= i <= size(s.eltlist)}
```

That is, the set consists of all the elements of the list.

Different representations have different abstraction functions. Another way to represent our `SeasonSet` is using an array of 4 booleans. Here the abstraction function may, for example, map

```
  [true, false, true, false]
```

to `{F,S}`, assuming the order `F`, `W`, `S`, `U` for the elements of the array. This order is the information conveyed by the abstraction function, which might be written, assuming the array is stored in a field `boolarr` as

```
  A(s) =
     (if s.boolarr[0] then {F} else {}) U
     (if s.boolarr[1] then {W} else {}) U
     (if s.boolarr[2] then {S} else {}) U
     (if s.boolarr[3] then {U} else {})
```

We could equally well have chosen a different abstraction function, that orders the seasons differently:

```
A(s) =
    (if s.boolarr[0] then {S} else {}) U
    (if s.boolarr[1] then {U} else {}) U
    (if s.boolarr[2] then {F} else {}) U
    (if s.boolarr[3] then {W} else {})
```

An important lesson from this last example is that 'choosing a representation' means more than naming some fields and selecting their types. The very same array of booleans can be interpreted in different ways; the abstraction function tells us which. Likewise, in our linked list example in Lecture 7, the abstraction function tells us how the order of entries corresponds to the order of elements. It is a common error of novices to imagine that the abstraction function is obvious, since you can always guess what it is from the declarations in the code. Unfortunately, this is often not true: it takes careful reading of the linked list code to discover that the first entry is a dummy entry, for example.

8.3 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. Note that this is essentially implementing a stack.

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 ImmutableQueue {
      ImmutableList back;
      ImmutableList front;
          ...
}
```

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,

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

Note that the definition uses the abstraction function on lists. We are assuming these lists represent mathematical sequences, and that `rev` 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 enqeueing D and E results in a `back` list of ED and a front list of BC. A(q) 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' over all operations.

8.4 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:

```
      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; `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. For example, when we refer to the i^{th} 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

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

as a comment in the `Queue` class.

8.5 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, will maintaining the abstract value it denotes. We can illustrate this phenomenon in general with a diagram:

$$\begin{tikzcd}
 & a & \\
r1 \arrow[ur, "A"] \arrow[rr, "op"] & & r2 \arrow[ul, "A"']
\end{tikzcd}$$

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 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 by noticing an improvement in performance); all it 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.

8.6 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.