Lecture 6: Representation Invariants and Abstraction Functions

6.1 Introduction

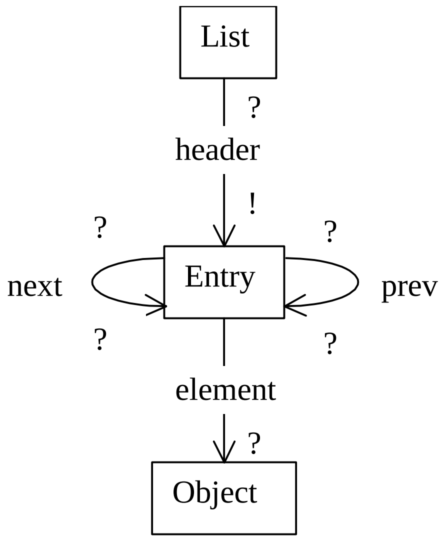
In this lecture, we describe two tools for understanding abstract data types: the representation invariant and the abstraction function. The representation invariant describes whether an instance of a type is well formed; the abstraction function tells us how to interpret it. Representation invariants 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.

6.2 What is a Rep Invariant?

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:

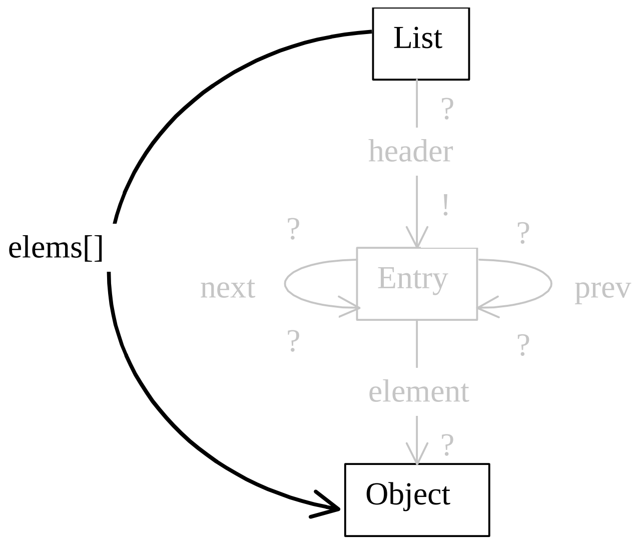
RI : Object -> Boolean

Consider the linked list implementation that we discussed last time. Here was its object model:



The *LinkedList* class has a field, *header*, that holds a reference to an object of the class *Entry.* This object has three fields: *element*, which holds a reference to an element of the list; *prev*, which points to the previous entry in the list; and *next*, which points to the next element.

This object model shows the *representation* of the data type. As we have mentioned before, object models can be drawn at various levels of abstraction. From the point of view of the user of the list, one might elide the box *Entry*, and just show a specification field from *List* to *Object*. This diagram shows that object model in black, with the representation in gold (*Entry* and its incoming and outgoing arcs) hidden:



The representation invariant is a constraint that holds for every instance of the type. Our object model already gives us some of its properties:

- It shows, for example, that the *header* field holds a reference to an object of class Entry. This property is important but not very interesting, since the field is declared to have that type; this kind of property is more interesting for the contents of polymorphic containers such as vectors, whose element type cannot be expressed in the source code.
- The multiplicity marking ! on the target end of the *header* arrow says that the *header* field cannot be null. (The ! symbol denotes exactly one.)
- The multiplicities ? on the target end of the next and prev arrows say that each of the next and prev arrows point to at most one entry. (The ? symbol denotes zero or one.)
- The multiplicities ? on the source end of the *next* and *prev* arrows say that each entry is pointed to by at most one other entry's *next* field, and by at most one other entry's *prev* field. (The ? symbol denotes zero or one.)
- The multiplicity ? on the target end of the element field says that each Entry points to at most one Object.

Some properties of the object model are not part of the representation invariant. For example, the fact that entries are not shared between lists (which is indicated by the multiplicity on the source end of the *header* arrow) is not a property of any single list.

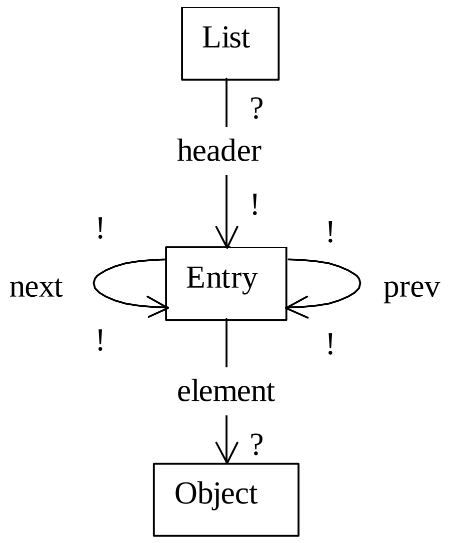
There are properties of the representation invariant which are not shown in the graphical object model:

- When there are two *e1* and *e2* entries in the list, if *e1.next* = *e2*, then *e2.prev* = *e1*.

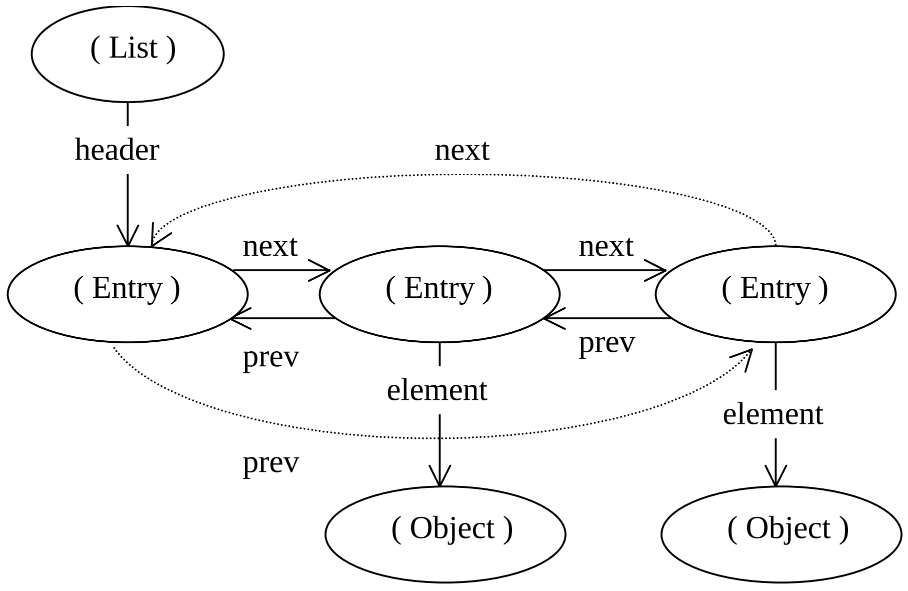
· The dummy entry at the front of the list has a null *element* field.

There are also properties that do not appear because the object model only shows objects and not primitive values. The representation of LinkedList has a field *size* that holds the size of the list. A property of the rep invariant is that *size* is equal to the number of entries in the list representation, minus one (since the first entry is a dummy).

In fact, in the Java implementation java.util.LinkedList, the object model has an additional constraint, reflected in the rep invariant. Every entry has a non-null *next* and *prev*:



Note the stronger multiplicities on the next and prev arrows. Here is a sample list of two elements (and therefore three entries, including the dummy):



When examining a representation invariant, it is important to notice not only what constraints are present, but also which are missing. In this case, there is no requirement that the *element* field be non-null, nor that elements not be shared. This is what we'd expect: it allows a list to contain null references, and to contain the same object in multiple positions.

Let's summarize our rep invariant informally:

for every instance of the class LinkedList
the header field is non-null
the header field has a null element field
there are (size + 1) entries
the entries form a cycle starting and ending with the header entry
for any entry, taking prev and then next returns you to the entry

We can also write this a bit more formally:

all p: LinkedList |
p.header != null
&& p.header.element = null
*&& p.size + 1 = | p.header.*next |*
&& p.header = $p.header.next^{p.size + 1}$
*&& all e in p.header.*next | e.prev.next = e*

To understand this formula, you need to know that

- for any expression *e* denoting some set of objects, and any field *f*, *e.f* denotes the set of objects you get if you follow *f* from each of the objects in *e*;
- *e.*f* means that you collect the set of objects obtained by following *f* any number of times from each of the objects in *e*;
- | e | is the number of objects in the set denoted by e.

So *p.header.*next* for example denotes the set of all entries in the list, because you get it by taking the list *p*, following the *header* field, and then following the *next* field any number of times.

One thing that this formula makes very clear is that the representation invariant is about a single linked list *p.* Another fine way to write the invariant is this:

R(p) =
p.header != null
&& p.header.element = null
*&& p.size + 1 = | p.header.*next|*
&& p.header = $p.header.next^{p.size + 1}$
*&& all e in p.header.*next | e.prev.next = e*

in which we view the invariant as a boolean function. This is the point of view we'll take when we convert the invariant to code as a runtime assertion.

The choice of invariant can have a major effect both on how easy it is to code the implementation of the abstract type, and how well it performs. Suppose we strengthen our invariant by requiring that the *element* field of all entries other than the header is non-null. This would allow us to detect the *header* entry by comparing its element to null; with the current invariant, operations that require traversal of the list must count entries instead or compare to the header field. Suppose, conversely, that we weaken the invariant on the *next* and *prev* pointers and allow *prev* at the start and *next* at the end to have any values. This will result in a need for special treatment for the entries at the start and end, resulting in less uniform code. Requiring *prev* at the start and *next* at the end both to be null doesn't help much.

6.3 Inductive 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 constructor creates an object that satisfies the invariant, and that every mutator and producer *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 some sample operations of our *LinkedList* class. The representation is declared in Java like this:

```
public class LinkedList {
    Entry header;
    int size;
    class Entry {
        Object element;
        Entry prev;
        Entry next;
        Entry (Object e, Entry p, Entry n) {element = e; prev = p; next = n;}
        }
    ...
```

Here's our constructor:

```
public LinkedList () {
    size = 0;
    header = new Entry (null, null, null);
    header.prev = header.next = header;
    }
```

Notice that it *establishes* the invariant: it creates the dummy element, forms the cycle, and sets the size appropriately.

The mutator *add* takes an element and adds it to the end of the list:

```
public void add (Object o) {
    Entry e = new Entry (o, header.prev, header);
    e.prev.next = e;
    e.next.prev = e;
    size++;
    }
```

To check this method, we can assume that the invariant holds on entry. Our task is to show that it also holds on exit. The effect of the code is to splice in a new entry just before the *header* entry, i.e., this new entry becomes the last entry in the *next* chain, so we can see that the constraint that the entries form a cycle is preserved. Note that one consequence of being able to assume the invariant on entry is that we don't need to do null reference checks: we can assume that *e.prev* and *e.next* are non-null, for example, because they are entries that existed in the list on entry to the method, and the rep invariant tells us that all entries have non-null *prev* and *next* fields.

Finally, let's look at an observer. The operation *getLast* returns the last element of the list or throws an exception if the list is empty:

```
public Object getLast () {
   if (size == 0) throw new NoSuchElementException ();
   return header.prev.element;
   }
```

Again, we assume the invariant on entry. This allows us to dereference *header.prev*, which the rep invariant tells us cannot be null. Checking that the invariant is preserved is trivial in this case, since there are no modifications.

6.4 Interpreting the Representation

Consider the mutator *add* again, which takes an element and adds it to the end of the list:

```
public void add (Object o) {
    Entry e = new Entry (o, header.prev, header);
    e.prev.next = e;
    e.next.prev = e;
    size++;
    }
```

We checked that this operation preserved the rep invariant, by correctly splicing a new entry into the list. What we didn't check, however, was that it was spliced 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) for a list *p* with elements *o1, o2, o3* to have

```
p.header.next.element = o3;
p.header.next.next.element = o2;
p.header.next.next.element = o1;
```

To resolve this problem, 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. The abstraction function for our implementation is:

```
A(p) =
   if p.size = 0 then
      <>  (the empty list)
   else
```

<p.header.next.element, p.header.next.next.element, ...>
(the sequence of elements with indices 0.. p.size-1 whose ith element is p.next^{i+1}.element)

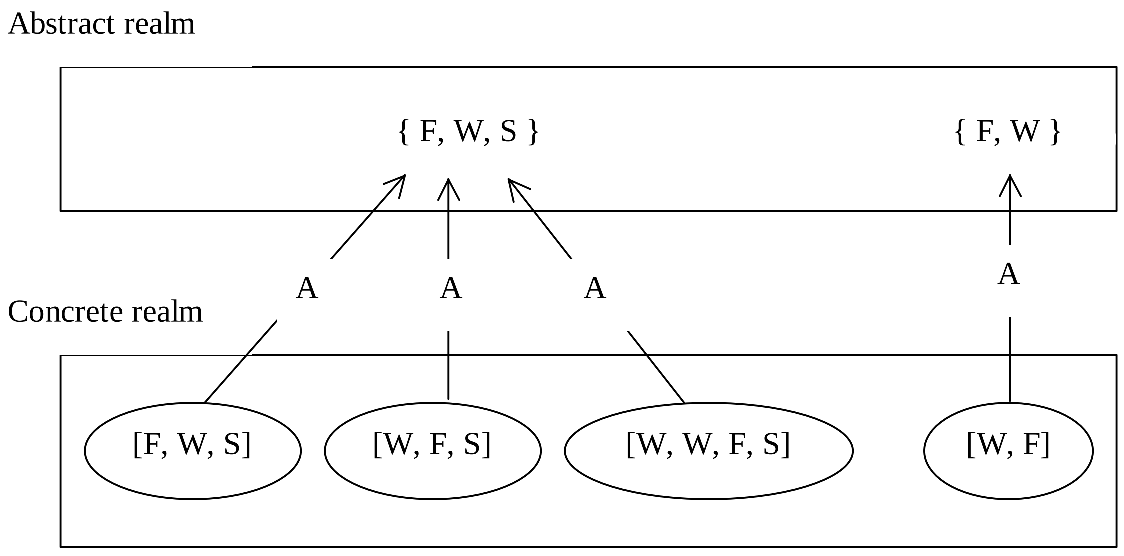
6.5 Abstract and Concrete Objects

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, 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.

6.6 Example: Boolean Formulas in CNF

Let's look at an example of a simple representation with a tricky abstraction function. A boolean formula is a mathematical formula constructed from *propositions* (symbols that can be assigned the values true and false) and logical *operators*. For example, the formula

CourseSix => sixOneSeventy

uses two propositions, *courseSix* and *sixOneSeventy*, and the logical implication operator. It says that if *courseSix* is true, *sixOneSeventy* is true also. A boolean formula is *satisfiable* if there is some assignment of boolean values to the propositions that makes the formula true. This formula is satisfiable, since we can set *courseSix* to false, or we can set both propositions to true.

An algorithm that determines whether a formula is satisfiable, and if so returns satisfying values for the propositions is called a *SAT solver*. SAT solvers have many applications, and their technology has advanced dramatically in the last decade. They are used in design tools for checking design constraints, in planners for finding plans, in testing tools for finding tests that expose particular classes of error, and so on. A SAT solver can also be used to check a proof. Suppose we assert that it follows from

CourseSix => sixOneSeventy

and

sixOneSeventy =>lateNights

that!

courseSix => lateNights

This is elementary reasoning using modus ponens, of course, but let's see how to check it with a SAT solver. We simply conjoin the premises to the negation of the conclusion:

(courseSix => sixOneSeventy) (sixOneSeventy => lateNights) (! (courseSix => lateNights))

and present this formula to the solver. The solver will find it not satisfiable, and will have demonstrated that it is impossible to have the premises be true and not the conclusion: in other words, the proof is valid.

Most SAT solvers use a representation of boolean formulas known as *conjunctive normal form*, or *CNF* for short. A formula in CNF is a set of clauses; each clause is a set of literals; a literal is a proposition or its negation. The formula is interpreted as a conjunction of its clauses and each clauses is interpreted as a disjunction of its literals. A more helpful name for CNF is *product of sums*, which makes it clear that the outermost operator is product (ie., conjunction).

For example, the CNF formula

{{a}{ !b,c}}

is equivalent to the conventional formula

a Λ (!b V c)

Our formula above would be represented in CNF as

{ {! courseSix,sixOneSeventy}, {! sixOneSeventy, lateNights}, {courseSix}, {! lateNights} }

Let's consider now how we might build an abstract data type that holds formulas in CNF. Suppose we already have a class *Literal* for representing literals.

Here is one reasonable representation that uses the Java library *ArrayList* class:

```
public class Formula {
    private ArrayList clauses;
    ...
    }
```

The *clauses* field is an *ArrayList* whose elements are themselves *ArrayLists* of literals.

Our representation invariant might then be

```
R(f) =
    f.clauses != null &&
    all c: f.clauses.elts |
        c instanceof ArrayList && c != null &&
            all l: c.elts | c instanceof Literal && c != null
```

I've used the specification field *elts* here to denote the elements of an ArrayList. The rep invariant says that the elements of the *ArrayList* clauses are non-null *ArrayLists*, each containing elements that are non-null *Literals*.

Here, finally, is the abstraction function:

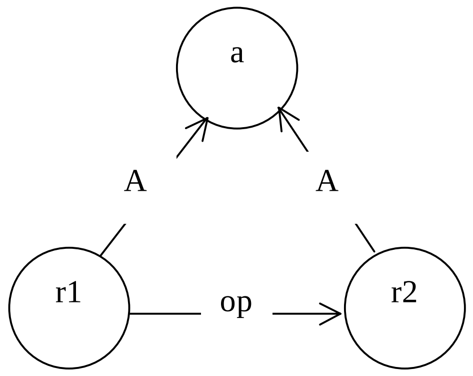
$$A(f) = true \Lambda C(f.clauses.elts[0]) \Lambda \ldots \Lambda C(f.clauses.elts[(size(f.clauses)-1])$$
$$\quad where\ C(c) = false \vee c.elts[0] \vee \ldots \vee c.elts[0]$$

Note how I've introduced an auxiliary function *C* that abstracts clauses into formulas. Looking at this definition, we can resolve the meaning of the boundary cases. Suppose *f.clauses* is an empty *ArrayList*. Then *A(f)* will be just true, since the conjuncts on the right-hand side of the first line disappear. Suppose *f.clauses* contains a single clause *c*, which itself is an empty ArrayList. Then *C(c)* will be false, and *A(f)* will be false too. These are our two basic boolean values: true is represented by the empty set of clauses, and false by the set containing the empty clause.

6.7 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:



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.

For example, the *get* method of *LinkedList* may cache the last element extracted, so that repeated calls to *get* for the same *index* will be speeded up. This writing to the cache (in this case just the two fields) certainly changes the rep, but it has no effect on the value of the object as it may be observed by calling operations of the type. The client cannot tell whether a lookup has been cached (except by noticing the improvement in performance).

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.

6.8 Summary

Why use rep invariants? Recording the invariant can actually save work:

· It makes modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what's going on before you can confidently add a new method.

· It helps catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.

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.