L11: Recursive Data Types

Today

- Immutable lists
- Datatypes & functions over datatypes
- Boolean expressions

Immutable Lists

Immutability is powerful not just because of its safety, but also because of the potential for sharing. Sharing actually produces performance benefits – less memory consumed, less time spent copying. Today we’re going to look at how to represent list data structures a different way.

Let’s define a data type for an immutable list, ImList<E>. The data type has four fundamental operations:

empty: void → ImList
   // returns an empty list

cons: E × ImList → ImList
   // returns a new list formed by adding an element to the front of another list

first: ImList → E
   // returns the first element of a list. requires the list to be nonempty.

rest: ImList → ImList
   // returns the list of all elements of this list except for the first. requires the list to be nonempty.

These four operations have a long and distinguished pedigree. They are fundamental to the list-processing languages Lisp and Scheme (where for historical reasons they are called nil, cons, car, and cdr, respectively). They are widely used in functional programming, where you can often find them called head and tail instead of first and rest.

Before we design Java classes to implement this datatype, let’s get used to the operations a bit, using lists of integers. For convenience, we'll write lists with square brackets, like [1,2,3], and we'll write the operations as if they are mathematical functions. Once we get to Java, the syntax will look different, but the operations will have the same meaning.

empty() = [ ]
cons(0, empty()) = [0]
cons(0, cons(1, cons(2, empty()) ) ) = [0, 1, 2]

x ≡ cons(0, cons(1, cons(2, empty()))) = [0, 1, 2]
first(x) = 0
rest(x) = [1, 2]
first(rest(x)) = 1
rest(rest(x)) = [2]
first(rest(rest(x))) = 2
rest(rest(rest(x))) = []

Fundamental relationship between first, rest, and cons:
first(cons(e, l)) = e
rest(cons(e, l)) = l

What cons puts together, first and rest peel back apart.

Immutable Lists in Java
To implement this datatype, we’ll use an interface:

```java
public interface ImList<E> {
    public ImList<E> cons(E e);
    public E first();
    public ImList<E> rest();
}
```

and two classes that implement it. Empty represents the result of the empty operation (an empty list), and Cons represents the result of a cons operation (an element glued together with another list):

```java
public class Empty<E> implements ImList<E> {
    public Empty() {}
    public ImList<E> cons(E e) { return new Cons<E>(e, this); }
    public E first() { throw new UnsupportedOperationException(); }
    public ImList<E> rest() { throw new UnsupportedOperationException(); }
}
```

```java
public class Cons<E> implements ImList<E> {
    private E e;
    private ImList<E> rest;

    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }
    public ImList<E> cons(E e) {
        return new Cons<E>(e, this);
    }
    public E first() {
        return e;
    }
    public ImList<E> rest() {
        return rest;
    }
}
```

So we’ve got methods for cons, first, and rest, but where is the fourth operation of our datatype, empty? Unfortunately Java makes it hard. The right way to do it is as a static method that takes no arguments and produces an instance of Empty. We’d like to cons this to the ImList interface along with the other operations, but Java doesn’t allow any method bodies in an interface, not even static
methods. So we’ll sacrifice some representation independence (more on this later) and use the Empty class constructor directly to represent the *empty* operation. So to do *empty*, you write new Empty().

Here’s some actual Java code that parallels the abstract examples we wrote earlier:

<table>
<thead>
<tr>
<th>Java syntax</th>
<th>Functional syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java ImList&lt;Integer&gt; nil = new Empty&lt;Integer&gt;();</code></td>
<td><code>nil = empty()</code></td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>nil.cons(0)</td>
<td><code>cons(0, nil)</code></td>
</tr>
<tr>
<td></td>
<td>nil.cons(2).cons(1).cons(0)</td>
<td><code>cons(0, cons(1, cons(2, nil)))</code></td>
</tr>
<tr>
<td><code>ImList&lt;Integer&gt; x = nil.cons(2).cons(1).cons(0);</code></td>
<td><code>x = cons(0, cons(1, cons(2, nil)))</code></td>
<td>[0, 1, 2]</td>
</tr>
<tr>
<td></td>
<td>x.first()</td>
<td><code>first(x)</code></td>
</tr>
<tr>
<td></td>
<td>x.rest()</td>
<td><code>rest(x)</code></td>
</tr>
<tr>
<td></td>
<td>x.rest().first()</td>
<td><code>first(rest(x))</code></td>
</tr>
<tr>
<td></td>
<td>x.rest().rest()</td>
<td><code>rest(rest(x))</code></td>
</tr>
<tr>
<td></td>
<td>x.rest().rest().first()</td>
<td><code>first(rest(rest(x)))</code></td>
</tr>
<tr>
<td></td>
<td>x.rest().rest().rest()</td>
<td><code>rest(rest(rest(x)))</code></td>
</tr>
<tr>
<td><code>ImList&lt;Integer&gt; y = x.rest().cons(4);</code></td>
<td><code>y = cons(4, rest(x))</code></td>
<td>[4, 1, 2]</td>
</tr>
</tbody>
</table>

Let’s look at a snapshot diagram of what x and y would look like:

![Diagram](image)

The key thing to note here is the sharing of structure that the immutable list provides.

**Two Classes, One Interface**

Note that this design is different from what we saw with List, ArrayList, and LinkedList. List is an abstract data type and ArrayList and LinkedList are two *alternative* concrete representations for that datatype.

For ImList, the two implementations Empty and Cons *cooperate* in order to implement the datatype – you need them both.
Recursive Datatype Definitions

The abstract data type ImList, and its two concrete classes Empty and Cons, form a recursive data type. Cons is an implementation of ImList, but it also uses ImList inside its own rep (for the rest field), so it recursively requires an implementation of ImList in order to successfully implement its contract.

To make this fact clearly visible, we'll write a datatype definition:

\[
\text{ImList}<E> = \text{Empty} + \text{Cons}(\text{first}: E, \text{rest}: \text{ImList})
\]

This is a recursive definition of ImList as a set of values. Read it like this: the set ImList consists of values formed in two ways: either by the Empty constructor, or by applying the Cons constructor to an element and an ImList. The recursive nature of the datatype becomes far more visible when written this way.

We can also write ImList values as terms or expressions using this definition, e.g.:

\[
\text{Cons}(0, \text{Cons}(1, \text{Cons}(2, \text{Empty})))
\]

Formally, a datatype definition has:

- an abstract datatype on the left, defined by its representation (or concrete datatype) on the right
- the representation consists of variants of the datatype separated by +
- each variant is a constructor with zero or more named (and typed) arguments

Another example is a binary tree:

\[
\text{Tree}<E> = \text{Empty} + \text{Node}(e:E, \text{left}: \text{Tree}, \text{right}: \text{Tree})
\]

We’ll see more examples later.

Functions over Immutable Datatypes

This way of thinking about datatypes – as a recursive definition of an abstract datatype with concrete variants – is appealing not only because it can handle recursive and unbounded structures like lists and trees, but also because it provides a convenient way to describe operations over the datatype, as functions with one case per variant.

For example, consider the size of the list, which is certainly an operation we'll want in ImList. We can define it like this:

\[
\text{size} : \text{ImList} \rightarrow \text{int}
\]

// returns the size of the list

and then fully specify its meaning by defining it for each variant of ImList:

\[
\text{size}(\text{Empty}) = 0
\]
\[
\text{size}(\text{Cons}(\text{first}: E, \text{rest}: \text{ImList})) = 1 + \text{size}(\text{rest})
\]

We can think about the execution of size on a particular list as a series of reduction steps:

\[
\text{size}(\text{Cons}(0, \text{Cons}(1, \text{Empty})))
\]
\[
= 1 + \text{size}(\text{Cons}(1, \text{Empty}))
\]
\[
= 1 + (1 + \text{size}(\text{Empty}))
\]
\[
= 1 + (1 + 0)
\]
Root = 2

And these cases can be translated directly into Java as methods in ImList, Empty, and Cons:

```java
public interface ImList<E> {
   ...
   public int size();
}

public class Empty<E> implements ImList<E> {
   ...
   public int size() { return 0; }
}

public class Cons<E> implements ImList<E> {
   ...
   public int size() { return 1 + rest.size(); }
}
```

Let’s try a few more examples:

isEmpty : ImList → boolean

- isEmpty(Empty) = true
- isEmpty(Cons(first: E, rest: ImList)) = false

contains : ImList x E → boolean

- contains(Empty, e: E) = false
- contains(Cons(first: E, rest: ImList), e: E) = (first = e) \lor contains(rest, e)

get: ImList x int → E

- get(Empty, e: E) = undefined
- get(Cons(first: E, rest: ImList), n) = if n=0 then first else get(rest, n-1)

append: ImList x ImList → ImList

- append(Empty, list2: ImList) = list2
- append(Cons(first: E, rest: ImList), list2: ImList) = cons(first, append(rest, list2))

reverse: ImList → ImList

- reverse(Empty) = empty()
- reverse(Cons(first: E, rest: ImList)) = append(reverse(rest), cons(first, empty()))

For reverse, it turns out that the recursive definition produces a pretty bad implementation in Java, with performance that’s quadratic in the length of the list you’re reversing. We can rewrite that better using an iterative approach.

### Tuning the Rep

Getting the size of a list is a common operation. Right now our implementation of size() takes O(n) time, where n is the length of the list. We can make it better with a simple change to the rep of the list that caches the size the first time we compute it, so that subsequently it costs only O(1) time to get:
```java
public class Cons<E> implements ImList<E> {
    private final E e;
    private final ImList<E> rest;
    private int size = 0;
    // rep invariant:
    // rest != null
    // size > 0 implies size == 1+rest.size()

    public int size() {
        if (size == 0) size = 1 + rest.size();
        return size;
    }
}
```

Note that we’re using the special value 0 (which can never be the size of a Cons) to indicate that we haven’t computed the size yet. Note also that this change introduces a new clause to the rep invariant, relating the size field to the rest field.

There’s something interesting happening here: this is an immutable datatype, and yet it has a mutable rep. It’s modifying its own size field, in this case to cache the result of an expensive operation. This is an example of a **beneficient mutation**, a state change that doesn’t change the abstract value represented by the object, so the type is still immutable.

**Rep Independence and Rep Exposure Revisited**

Does ImList still have rep independence? Yes and no. We really want to hide the classes Empty and Cons from the client. For simplicity, we exposed the constructor of Empty. We could still hide Cons, though, by making it package-private (i.e. declaring it with neither public nor private), so that classes outside ImList’s package can’t see it or use it.

But we do still have a lot of freedom. The internal rep of Cons could add a size field. We could even have an extra array in there to make get() run fast! This would get pretty expensive in space, though.

Is there rep exposure because Cons.rest() returns a reference to its internal list? Could a clumsy client add elements to the rest of the list? If so, this would threaten two of Cons’s invariants: that it’s immutable, and that the cached size is always correct. But there’s no risk of rep exposure, because the internal list is immutable. Nobody can threaten the rep invariant of Cons.

**Null vs. Empty**

It might be tempting to get rid of the Empty class and just use null instead. Resist that temptation.

Using an object, rather than a null reference, to signal the base case or endpoint of a data structure is an example of a design pattern called *[sentinel objects](https://en.wikipedia.org/wiki/Sentinel_%28data_structure%29)*. The enormous advantage that a sentinel object provides is that it acts like an object in the datatype, so you can call methods on it. So we can call the size() method even on an empty list. If empty lists were represented by null, then we wouldn’t be able to do that, and as a consequence our code would be full of tests like:

```
if (lst != null) n = lst.size();
```

which clutter the code, obscure its meaning, and are easy to forget. Better the much simpler

```
n = lst.size();
```

which will always work if an empty lst refers to an Empty object.

Keep nulls out of your data structures, and your life will be happier.
Declared Type vs. Actual Type

Now that we’re using interfaces and classes, it’s worth a moment to reinforce an important point about how Java’s type-checking works. In fact every statically-checked object-oriented language works this way.

There are two worlds in type checking: **compile time** before the program runs, and **run time** when the program is executing.

At compile time, every variable has a **declared type**, stated in its declaration. The compiler uses the declared types of variables (and method return values) to deduce declared types for every expression in the program.

At run time, every object has an **actual type**, imbued in it by the constructor that created the object. For example, new String() makes an object whose actual type is String. new Empty() makes an object whose actual type is Empty. new ImList() is forbidden by Java, because ImList is an interface – it has no object values of its own, and no constructors.

Another Example: Boolean Formulas

Another useful sort of recursive datatype in computer science is for Boolean formulas. For instance, here’s a formula of propositional logic:

\[(P \lor Q) \land (\neg P \lor R)\]

It means “either P or Q is true, and either P is false or R is true.”

We can give a datatype definition suitable for representing all formulas of propositional logic.

```java
Formula = Variable(name:String) 
+ Not(formula: Formula) 
+ Or(left: Formula, right: Formula) 
+ And(left: Formula, right: Formula)
```

\[(P \lor Q) \land (\neg P \lor R)\] would be

```java
And( Or(Variable(“P”), Variable(“Q”)), 
     Or(Not(Variable(“P”)), Variable(“R”)) )
```

A key operation on Boolean formulas is testing whether they are **satisfiable**, that is, whether some assignment of true/false values to the variables leads the formula to evaluate to true. There is a simple but slow algorithm for checking satisfiability:

1. extract set of variables from formula
2. try all assignments of true/false values to those variables
   a. we'll represent an assignment with an *environment* Environment, which is just a list of variables and their values
3. evaluate the formula for each environment
4. return the first environment in which the formula evaluates to true
A few functions will be useful to implement the above pseudocode:

- `satisfiable : Formula → Boolean`
- `variables : Formula → Set<Variable>`
- `evaluate : Formula × Environment → Boolean`

And here are definitions of the datatypes we mention above:

- `Set<T> = ImList<T>`
- `Environment = ImList<Variable × Boolean>`
- `Boolean = True + False`

Let's end with some nice recursive definitions of the functions.

```plaintext
// Helper function: duplicate-free union of duplicate-free sets
union : Set<E> * Set<E> → Set<E>
union(Empty, s) = s
union(Cons(x, s1), s2) = if contains(s2, x) then union(s1, s2)
else union(s1, Cons(x, s2))

variables(Variable(x)) = Cons(x, Empty)
variables(Not(f)) = variables(f)
variables(And(f1, f2)) = union(variables(f1), variables(f2))
variables(Or(f1, f2)) = union(variables(f1), variables(f2))

// Helper function: get a variable's value from an Environment
lookup : Environment × String → Boolean
lookup(Empty, x) = undefined
lookup(Cons((y, v), rest), x) = if y = x then v else lookup(rest, x)

evaluate(Variable(x), environment) = lookup(environment, x)
evaluate(Not(f), environment) = ! evaluate(f, environment)
evaluate(And(f1, f2), environment) = evaluate(f1, environment) && evaluate(f2, environment)
evaluate(Or(f1, f2), environment) = evaluate(f1, environment) || evaluate(f2, environment)

// Helper function: consider all possible assignments to a set of variables
evalVars : Set<Variable> * Environment * Formula -> Boolean
evalVars(Empty, environment, f) = evaluate(f, environment)
evalVars(Cons(x, xs), environment, f) = evaluate(f, Cons((x, false), environment))
```
satisfiable(f) = evalVars(variables(f), Empty, f)

Backtracking Search with Immutability

A final comment about what we’ve seen in this lecture. We started out with immutable lists, which are a representation that permits a lot of sharing between different list instances. Sharing of a particular kind, though: only the ends of lists can actually be shared. If two lists are identical at the beginning but then diverge from each other, they have to be stored separately. (Why?)

It turns out that backtracking search is a great application for these lists, and here’s why. A search through a space (like the space of assignments to a set of Boolean variables) generally proceeds by making one choice after another, and when a choice leads to a deadend, you backtrack.

Mutable data structures are typically not a good approach for backtracking. If you use a mutable Map, say, to keep track of the current variable bindings you’re trying, then you have to undo those bindings every time you backtrack. That’s error-prone and painful compared to what you do with immutable maps – when you backtrack, you just throw the map away!

But immutable data structures with no sharing aren’t a great idea either, because the space you need to keep track of where you are (in the case of the satisfiability problem, the environment) will grow quadratically if you have to make a complete copy every time you take a new step. You need to hold on to all the previous environments on your path, in case you need to back up.

Immutable lists have the nice property that each step taken on the path can share all the information from the previous steps, just by adding to the front of the list. When you have to backtrack, you stop using the current step’s state – but you still have references to the previous step’s state.

Perhaps best of all, a search that uses immutable data structures is immediately ready to be parallelized. You can delegate multiple processors to search multiple paths at once, without having to deal with the problem that they’ll step on each other in a shared mutable data structure. We’ll talk about this more when we get to concurrency.

Summary

big ideas

• datatype definitions: a powerful way to think about abstract data types, particularly recursive ones
• backtracking search: easy with immutable types