L7: Recursive Data Types

Today

- Immutable lists
- Datatypes & functions over datatypes
- Abstract syntax trees
- SAT

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:

```
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):

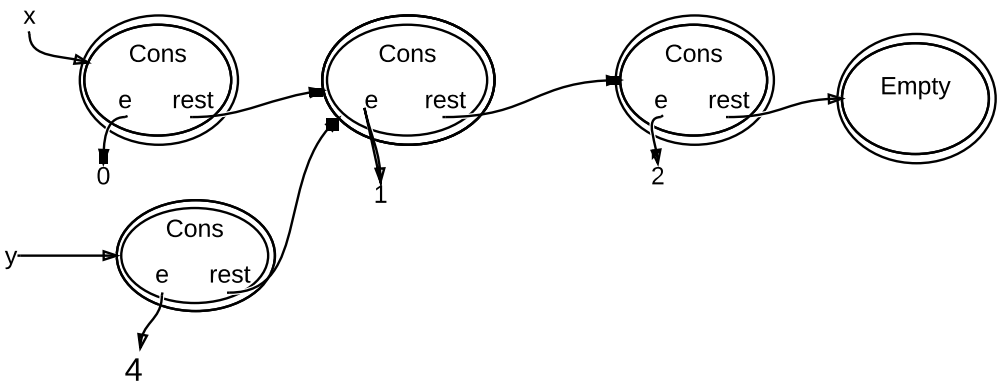
```
public class Empty<E> implements ImList<E> {
    public Empty() { }
    public ImList<E> cons(E e) { return new Cons<E>(e, this); }
    public E first() { throw new UnsupportedOperationException(); }
    public ImList<E> rest() { throw new UnsupportedOperationException(); }
}

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:

Java syntax	*Functional syntax*	Result
`ImList<Integer> nil = new Empty<Integer>();`	*nil ≡ empty()*	[ ]
`nil.cons(0)`	*cons(0, nil)*	[0]
`nil.cons(0).cons(1).cons(2)`	*cons(0, cons(1, cons(2, nil)))*	[0, 1, 2]
`ImList<Integer> x = nil.cons(0).cons(1).cons(2);`	*x ≡ cons(0, cons(1, cons(2, nil)))*	[0, 1, 2]
`x.first()`	*first(x)*	0
`x.rest()`	*rest(x)*	[1, 2]
`x.rest().first()`	*first(rest(x))*	1
`x.rest().rest()`	*rest(rest(x))*	[2]
`x.rest().rest().first()`	*first(rest(rest(x))*	2
`x.rest().rest().rest()`	*rest(rest(rest(x)))*	[ ]
`ImList<Integer> y = x.rest().cons(4);`	*y ≡ cons(4, rest(x))*	[4, 1, 2]

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



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**:

ImList = Empty + Cons(first:E , rest:ImList)

This is a recursive definition of ImList 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.:

Cons(0, Cons(1, Cons(2, Empty)))

Formally, a datatype definition has:

- the datatype on the left
- **variants** of the datatype separated by + on the right
- each variant is a constructor with zero or more named (and typed) arguments

Another example is a binary tree:

Tree = Empty + Node(e:E, left:Tree, right: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:

size : ImList → int

// returns the size of the list

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

size(Empty) = 0

size(Cons(first: E, rest: ImList)) = 1 + size(rest)

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

size(Cons (0, Cons (1, Empty)))

= 1 + size(Cons (1, Empty))

= 1 + (1 + size(Empty))

= 1 + (1 + 0)

= 2

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

```
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) ∨ 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(rest, reverse(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:

```
    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, 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*. 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.

Datatype Definitions vs. Grammars

You can think of datatype definitions are being like little grammars -- just like the kind you've seen for specifying the form of programs, or commands to a shell, etc. But unlike grammars, which show the concrete form the objects take (ie, their physical appearance – e.g. including parentheses and curly braces), datatype definitions represent only the conceptual shape. Put another way, datatype definitions don't help you parse, but are good for representing the results of parsing.

A parser for a textual language usually produces a tree-shaped datatype. This datatype is called an *abstract syntax tree,* or AST. It's called abstract to distinguish it from a concrete syntax tree that directly follows the syntax. An abstract syntax tree may omit details (like extra pairs of parentheses or curly braces) or expand syntactic sugar.

The datatype definition for an AST, despite being abstract, can take its shape directly from the grammar. Recall our grammar for HTML markup:

> Html ::= (Normal | Italic) *
>
> Italic ::= Html
>
> Normal ::= Text
>
> Text ::= [^ _]*

Let's use this grammar to define a recursive datatype for Html. Here's one possibility, which uses an ImList to represent the repetition in the Html production, and then needs a new datatype to represent Normal | Italic expression:

> Html = ImList<Node>
>
> Node = Normal(content: String) + Italic(content: Html)

Here's the code:

```
public class Markup {
    public class Html {
        private final ImList<Node> nodes;
        public Html(ImList<Node> nodes) { this.nodes = nodes; }
    }

    public interface Node {
    }

    public class Normal {
        private final String content;
        public Normal(String content) { this.content = content; }
    }

    public class Italic {
        private final Html content;
        public Italic(Html content) { this.content = content; }
    }
}
```

Now let's implement some functions over this datatype.

length : Html → int

// returns number of characters of text

length(Cons(first: Node, rest: ImList) = true

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

scramble : Html → String

// returns text in the tree concatenated together,
// with each italic region randomly scrambled

Boolean Formulas and Satisfiability

the SAT problem

- given a formula made of boolean variables and operators $(P \vee Q) \wedge (\neg P \vee R)$
- find an assignment to the variables that makes it true
- possible assignments, with solutions in green, are:

{P = false, Q = false, R = false}

{P = false, Q = false, R = true}

{P = false, Q = true, R = false}

{P = false, Q = true, R = true}

{P = true, Q = false, R = false}

{P = true, Q = false, R = true}

{P = true, Q = true, R = false}

{P = true, Q = true, R = true}

SAT solver

- program that takes a boolean formula in CNF
- returns an assignment, or says none exists

how to build a SAT solver, version one

- just enumerate assignments, and check formula for each
- for k variables, 2^k assignments: surely can do better?

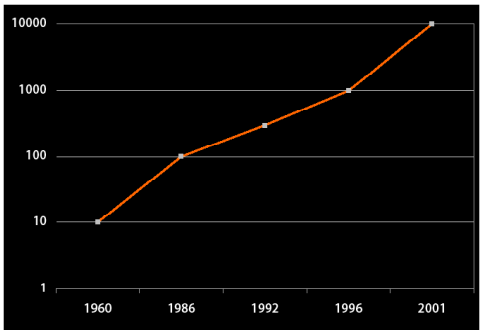
SAT is hard

- in the worst case, no: you can't do better
- Cook (1973): 3-SAT (3 literals/clause) is "NP-complete"
- the quintessential "hard problem" ever since

how to be a pessimist

- suppose you have a problem P (that is, a class of problems)
- show SAT reducible to P (ie, can translate any SAT-problem to a P-problem)
- then if P weren't hard, SAT wouldn't be either; so P is hard too

remarkable discovery



Courtesy of Sharad Malik. Used with permission.

#boolean vars SAT solver can handle (from Sharad Malik)

- most SAT problems are easy
- can solve in much less than exponential time

how to be an optimist

- suppose you have a problem P
- reduce it to SAT, and solve with SAT solver

In the 1980s, researchers were publishing papers on how to find hard SAT problems! It turned out that even though in the worst case SAT is really hard, in practice almost all the cases you get if you generate them randomly are easy. And the ones that arise in real problems are often easy too. The story's actually more complicated than this though; it turns out that there's what's called a 'phase transition', a point at which problems get really hard, and this phase transition is roughly at the midpoint between the two extremes of the formula being so constrained it's easy to solve because you can easily determine values for variables early on, and the formula being so underconstrained that it's easy to solve just by guessing.

applications of SAT

planning

- solve (initial state $\wedge$ goal $\wedge$ rules) to obtain plan
- eg, ZYpp package manager for Linux

verification

- solve (code $\wedge$ $\neg$ spec) to obtain counterexample
- industrial application to hardware; software applications coming
- eg, Cadence Incisive hardware verifier

design

- solve (design rules $\wedge$ constraints $\wedge$ requirements) to obtain design

for more info

- see http://www.satlive.org

why are we teaching you this?

SAT is cool

- good for (geeky) cocktail parties
- many useful applications
- compilation-to-SAT idea is powerful
- builds on your 6.042 knowledge

fundamental techniques

- you'll learn about datatypes and functions
- same ideas will work for any compiler or interpreter

A Naive SAT Solver

one way to represent boolean formulas

Formula = Var(name:String)
+ Not(formula: Formula)
+ Or(left: Formula,right: Formula)
+ And(left: Formula,right: Formula)

(P∨ Q)∧ (¬P∨ R) would be

```
And( Or(Var("P"), Var("Q")),
     Or(Not(Var("P")), Var("R")) )
```

Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal) would be:

```
And( Or(Not(Var("Socrates")), Var("Human")),
     And ( Or(Not(Var("Human")), Var("Mortal")),
           Not( Or(Not(Var("Socrates")),Var("Mortal"))))))
```

Note that a client of this datatype should NOT see the internal classes, Not, Or, and And. They should use abstract operations to build the formula. In this case they would be operators like *and*, *or*, and *not*. We have to make an exception with the Var class, and expose it, or else use a constructor method. So here's what a client might write in Java:

P∨ Q new Var("P") .or (new Var("Q"))

¬P∨ R (new Var("P").not()) .or (new Var("R"))

a naive SAT solver

generate and test strategy

- steps
    1. extract set of variables from formula
    2. try all assignments of true/false values to those vars
        a. we'll represent an assignment with an *environment* Env, 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
- functions we'll need

    vars: Formula → Set<Var>

    solve: Formula → Env?

    eval: Formula, Env → Boolean

new datatypes we'll need

Set<T> = ImList<T>

Env = ImList<Var × Boolean>

Boolean = True + False + Undefined

what's wrong with our solver?

consider formula

Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal)

suppose order of trying the variables is Socrates, Human, Mortal

and suppose we set Socrates to true

then clearly must set Human to true

and then must set Mortal to true...

…but our solver ignores all this!

A better SAT Solver

Conjunctive Normal Form (CNF)

conjunctive normal form (CNF) or "product of sums"

(P∨ Q)∧ (¬P∨ R) is in conjunctive normal form.

- set of clauses, each containing a set of literals {{P, Q}, {¬P, R}}
- literal is just a variable, maybe negated

Note that CNF is just a format for a boolean formula -- but one that turns out to be very helpful, making it easier to write solvers. The notion of a literal is important, since it means you can only negate variables, and not clauses.

Datatype definition:

Formula = ImList<Clause> // a list of clauses ANDed together

Clause = ImList<Literal> // a list of literals ORed together

Literal = Positive(v: Var) + Negative(v:Var) // either a variable P or its negation ¬P

Var = String

Note that as long as the concrete classes we were using in the old Formula (And, Or, Not) were hidden from the client, and the client was limited to using abstract operations to combine formulas (and, or, not), then we can freely make this change to the rep without changing any client code. e .g.,

¬P∨ R (new Var("P").not()) .or (new Var("R"))

now constructs a data structure that looks like:

[[Negative(Var("P")), Positive(Var("R"))]]

That is, a list containing a single clause, which in turn contains two literals, one negative and one positive.

basic backtracking algorithm using CNF

- CNF is a product of sums: we need every clause true, and at least one literal in each clause

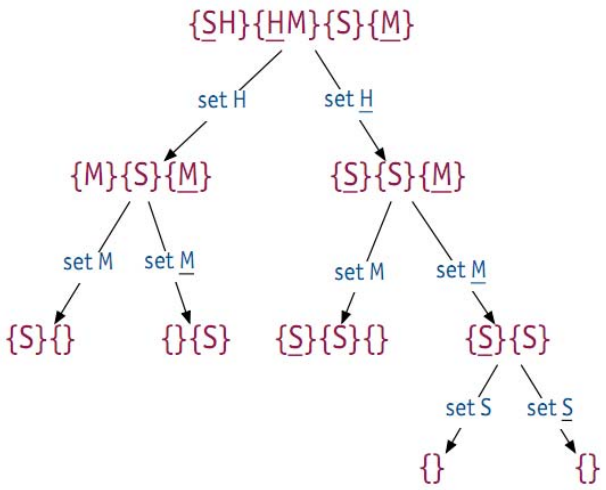
elements of the algorithm

- backtracking search: pick a literal, try false then true
- if clause set is empty, success
- if clause set contains empty clause, failure

example

- want to prove Socrates⇒Mortal from Socrates⇒Human ∧ Human⇒Mortal
- so give solver: Socrates⇒Human ∧ Human⇒Mortal ∧ ¬ (Socrates⇒Mortal)
- in CNF: (¬Socrates ∨ Human) ∧ (¬Human ∨ Mortal) ∧ Socrates ∧ ¬Mortal
- in clausal form: {{¬Socrates,Human},{¬Human,Mortal},{Socrates},{¬Mortal}}
- in shorthand: {S̲H} {H̲M} {S} {M̲} (underlines mean negation)

backtracking execution



- stop when node contains {} (failure) or is empty (success)
- in this case, all paths fail, so theorem is valid
- in worst case, number of leaves is 2^#literals

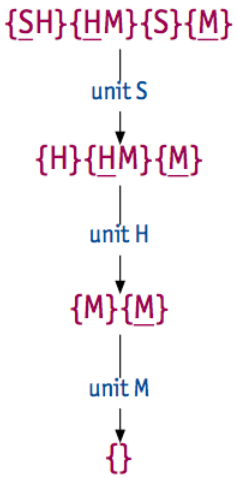
DPLL: the classic SAT algorithm

- Davis-Putnam-Logemann-Loveland, 1962

key idea: **unit propagation** on top of backtracking search

- if a clause contains one literal, set that literal to true

example



- in this case, no splitting needed
- propagate S, then H, then M

- performance is often much better, but worst case still exponential

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 SAT, 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
- SAT: an important problem, theoretically & practically

6.005 Elements of Software Construction

Fall 2011