Recursive Data Types

Spring 2013
Today’s Topics

Review: Abstraction Functions & Representation
Invariants
Immutable lists
Datatype definitions
Abstract syntax trees (ASTs)
Abstraction Function

**In the graph**

- Every abstract value is mapped to, since we need to manipulate/create all possible abstract values
- Some abstract values are mapped to from more than one representation value
- Not all representation values are mapped

**Abstraction function AF: R → A**

- Maps representation values to the abstract values they represent
- Not necessarily one-to-one, but is an onto function
- For all values in A, there is a representation in R such that AF maps that representation to the value in A
Representation Invariant

In the graph

- Every abstract value is mapped to, since we need to manipulate/create all possible abstract values
- Some abstract values are mapped to from more than one representation value
- Not all representation values are mapped

Representation invariant \( \text{RI}: R \rightarrow \text{boolean} \)

- For a representation value \( r \), \( \text{RI}(r) \) is true iff \( r \) is mapped by \( \text{AF} \)
- Representation invariant tells us whether a given representation value corresponds to an abstract value
- \( \text{RI} \) forms a set: the subset of the representation values on which \( \text{AF} \) is defined
We could have the same ADT with a looser Representation Invariant

- E.g. remove the requirement that numer/denom needs to be in reduced form
Proving Correctness of ADT Implementations

Combination of Abstraction Function & Representation Invariant define an invariant for the class

- Use Structural Induction!

Additional correctness constraint: representation-independent specification of each method

- Interpret the spec using the abstraction function & verify the method follows the specification
Why Do We Talk So Much About Mutability?

Last week, we learned to think about mutable data types as state machines

- Easy to model behavior, even if model is not exact
- Model can then be used to reason about correctness, implementations, etc.

For immutable data types, the corresponding model is a function

- Methods & programs using immutable data types modeled as function application & composition
- Powerful programming model used in functional languages like Haskell, ML, Scheme
- Java is not a perfect fit, but supports some kinds of functional programming
- Many applications have portions or data types best viewed functionally
Immutable Lists

A fundamental data structure in many languages
- Scheme, Lisp, etc
- Can be shared safely
- Performance benefits: less time copying, less memory consumed

Four fundamental operations
- empty: void → ImList
  - Constructor
- cons: E × ImList → ImList
  - Returns a new list formed by adding a new element to the front of an existing list
- first: ImList → E
  - Returns the first element of a list. The list must be nonempty.
- rest: ImList → ImList
  - Returns the list of all elements of the list except the first. The list must be nonempty
Example Operations

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

x = cons(1, cons(2, cons(3, empty()))) = [1, 2, 3]
first(x) = 1
rest(x) = [2, 3]

first(rest(x)) = 2
rest(rest(x)) = [3]
rest(rest(rest(x))) = []

// fundamentally, for element e and list l,
first(cons(e, l)) = e
Rest(cons(e, l)) = l
Implementing ImList

/** Immutable List interface */
public interface ImList<E> {
    /**
     * Cons adds a new item to the front of the list
     * @param e the element to add to the front
     * @return a new list consisting of e followed by this list */
    public ImList<E> cons (E e);

    /**
     * Returns the first element of the list. Requires the list be nonempty.
     * @return the first element of the list */
    public E first();

    /**
     * Returns the list except the first element. Requires the list to be nonempty.
     * @return the list except for the first element.
     */
    public ImList<E> rest();
Implementing ImList: Empty

/**
 * Implements the result of the empty() operation on immutable
 * lists.
 */

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();
    }

}
Implementing ImList: Cons

/**
 * Implements the result of a cons operation.
 */

public class Cons<E> implements ImList<E> {

    // the element
    private E e;

    // the rest of the list
    private ImList<E> rest;

    // constructor
    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; }

}
What About the empty() Operation?

Would really like to have a static method that creates an empty list, but Java doesn’t let us do this through ImList interface.

- Instead, empty() is (new Empty())
- Sacrificing some representation independence

Example Operations

```java
ImList<Integer> e = new Empty<Integer>();
e.cons(0)
e.cons(2).cons(1)
x.rest().first()
```
ImList & Sharing

ImList<Integer> x = empty().cons(2).cons(1).cons(0)
ImList<Integer> y = x.rest().cons(4)

Is this sharing safe?
ImList vs ArrayList/LinkedList

ArrayList and LinkedList are two implementations of the List ADT

But here, Cons and Empty are two classes that cooperatively implement the ADT

➢ We need both classes
Data Type Definitions

MyInt = int

- MyInt is a synonym for the built-in type int

Suit = Club + Diamond + Heart + Spade

- Example of an enum-like type

IntOrFloat = Int(num:int) + Float(num:float)

- Union-like type

Data type definition has

- Data type on the left
- Variants of the data type separated by “+” on the right
- Each variant = constructor with 0 or more named & typed arguments
Recursive Data Types

Data type definition for ImList

- \( \text{ImList} = \text{Empty} \) + \( \text{Cons(first:E, rest:ImList)} \)

ImList appears on both the left and right sides

- \( \text{ImList} \) is a recursive data type

Another example: binary tree

- \( \text{Tree} = \text{Empty} \) + \( \text{Node(element:E, left:Tree, right:Tree)} \)
Functions on Recursive Data Types

Using data type definitions makes defining operations easier

- Just think of operations in terms of one case per variant

**Example: size()**

- size: ImList → Int
- Data type definition: ImList = Empty + Cons(first:E, rest:ImList)
- So, two cases
  - size(Empty) = 0
  - size(Cons(first:E, rest:ImList)) = 1 + size(rest)
- This recursive definition leads naturally to simple, understandable, recursive code
Functions on Recursive Data Types

**isEmpty: ImList \rightarrow boolean**
- isEmpty(Empty) = true
- isEmpty(Cons(first:E, rest:ImList)) = false

**contains: ImList \times E \rightarrow boolean**
- contains(Empty, e:E) = false
- contains(Cons(first:E, rest:ImList), e:E) = first==e || contains(rest, e)

**append: ImList \times ImList \rightarrow ImList**
- append(Empty, list2:ImList) = list2
- append(Cons(first:E, rest:ImList), list2:ImList) = cons(first, append(rest, list2))
Null vs Empty

One might think the Empty class is unnecessary--- why not use null instead?

Empty is an example of a Sentinel Object
- Like the special last list element in the previous lecture

Sentinel objects behave like objects of the same data type
- E.g. can call size() on Empty
- Cleaner code, since instead of checking if a reference is null, we just call the method on the object
- Good practice in designing ADTs: will prevent bugs & save programmer effort
Declared Type vs Actual Type

In Java, two regimes for type checking: compile time and run time

Compile-time type checking
- Every variable has a *declared* type (the types in the declarations)
- Compiler enforces restrictions on declared types
  - E.g. only calling methods of declared type through its reference
  - E.g. not having statements like `int foo = "hello";`

Run-time type checking
- Every object has an *actual type* dictated by the constructor
- E.g. `ImList<Integer> a = new Empty();`
  - Actual type of `a` is `Empty`, not `ImList`
- Some classes of type errors are caught at run time
Grammars, Again

Data type definitions probably remind you of our grammar syntax

- But different! Data type definitions express conceptual idea of a type
- Grammars express concrete form of a language
- Key idea: data type definitions are good way of expressing results of parsing

Textual languages usually parsed into Abstract Syntax Tree

- A particular kind of tree-like recursive data type
- *Abstract* because it may omit some details, unlike *concrete syntax*, which is tree that directly expresses the syntax
AST Example: HTML Subset

Html ::= ( Normal | Italic ) *
Italic ::= <i> Html </i>
Normal ::= Text
Text ::= [^ < ]*

Our data type definition corresponds closely to grammar

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

Another possibility

- Node = Html(content:ImList<Node>) + Normal(content:String) + Italic(content:Html)
AST Example: HTML Subset
We can define functions over this recursive data type just like any other one

\[
\text{numTextRegions: Node} \rightarrow \text{int}
\]

\[
\text{numTextRegions(Html(content:ImList))} = \sum \text{numTextRegions(content.get(i)) for } i=0..\text{content.size}
\]

\[
\text{numTextRegions(Italic(content:Html))} = \text{numTextRegions(content)}
\]

\[
\text{numTextRegions(Normal(content:String))} = 1
\]
Boolean Satisfiability (SAT)

Given a formula made up of boolean variables and operators \( \land \) (and), \( \lor \) (or), \( \neg \) (not), find an assignment of variables that makes the formula true

Example: \((P \lor Q) \land (\neg P \lor R)\)
- Not true if \(P=\text{false}, Q=\text{false}, R=\text{false}\)
- Satisfied when \(P=\text{false}, Q=\text{true}, R=\text{true}\)
- Other solutions as well (E.g. \(P=\text{true}, Q=\text{false}, R=\text{true}\))

Conjunctive Normal Form (CNF)
- Write a boolean formula as a “product of sums”
- i.e. Each clause is a sum (just \(\lor\)) and the clauses are combined with \(\land\)
- Standard way to write boolean formulas
SAT Solving

Why do we care about boolean satisfiability?
- Theoretically speaking, SAT is a canonical difficult problem
- 3-SAT (3 literals per clause) is known to be NP-complete [Cook, 1973]
- Basis for theory of difficulty in computability

Why is it hard?
- $k$ variables have $2^k$ possible assignments
- Can we do better than to check each possible assignment?
- For many SAT problems, no. But for others, yes.
Recursive Data Type for SAT

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

Example: (P ∨ Q) ∧ (¬P ∨ R)
�� And(Or(Var(“P”), Var(“Q”)), Or(Not(Var(“P”)), Var(“R”)))

This is an abstract syntax tree for SAT
We can write a simple grammar for 2-SAT and parse that into our recursive data type Formula

Grammar for 2-SAT

- SatFormula ::= Clause | Clause andsymbol SatFormula
- Clause ::= leftparen Var orsymbol Var rightparen
- Var ::= [A-Z]+

Use the recipe for recursive descent parsing

- Each grammar rule has its own method
- Each grammar rule produces a Formula object
- E.g. the Clause rule above results in a method that produces an Or object
Parsing a SAT Problem

\[(P \lor Q)\]

**Formula** `parseSATFormula(…)`

```java
return parseClause(…)
```

**Formula** `parseClause(…)`

```java
return new Or(parseVar(…), parseVar(…))
```

**Formula** `parseVar(…)`

```java
return new Var(…)
```

**Formula** `parseVar(…)`

```java
return new Var(…)
```
Implementing Satisfiability

We’ll use the simplest possible strategy: enumerate all variables, then try every boolean value for each variable

1. Extract the set of variables from a formula
2. Try all assignments of true/false for each variable
   - Represent the assignment with a data type Environment, which just maps each variable to its current assignment
3. Evaluate the boolean expression for each environment
4. If an environment results in the expression being true, return the assignment

Create functions functions over our Formula data type

- satisfiable: Formula → boolean
- getAllVariables: Formula → Set<Var>
- eval: Formula, Environment → boolean
Implementing Satisfiability

Easy to see how eval works (assume Environment is just a Java Map)

- eval(Var(name:String), Environment) = Environment.get(name)
- eval(Not(formula:Formula), Environment) = !eval(formula, Environment)
- eval(And(left:Formula, right:Formula), Environment) = eval(left, Environment) && eval(right, Environment)
- eval(Or(left:Formula, right:Formula), Environment) = eval(left, Environment) || eval(right, Environment)
Summary

Recursive data types are ADTs that may reference instances of themselves

Abstract syntax tree: an important recursive data type used to represent parsing languages & performing operations over the parsed tree

Next time: Patterns in Software Engineering & patterns over ASTs