how to design an abstract type

Daniel Jackson
March 17, 2008
plan for today

topics

‣ the idea of abstract types
‣ value of immutability
‣ defining the contract
‣ classic types: List, Set, Map
‣ a domain-specific type: Clause
‣ representation independence
‣ patterns: iterators and factories

reminders

‣ lecture note handout
‣ Thursday: P3a due
abstract types
what types do you need?

a square root procedure needs
- floating point numbers

a SAT solver needs
- booleans, literals, clauses, environments

characteristic of complex programs
- computations defined over set of datatypes
- most of the datatypes are not built-in, but user-defined
- datatypes are abstract, with representation hidden from clients

schedule
- how to design abstract types (this lecture)
- how to implement them (next lecture)
what makes an abstract type?

defined by operations

- an abstract type is used through its operations
- eg, +,-,\,* for built-in integers; store/get for arrays

representation is hidden or “encapsulated”

- client can't see how the type is represented in memory
- eg, is integer twos-complement? big or little endian?

language support for data abstraction

- packaging operations with representations
- hiding representation from clients
encapsulation

two reasons for encapsulation of representations

rep independence
· if client can’t see choice of rep, implementor can change it
· eg: integers: your program can run on a different platform
· eg: sets: programmer can switch rep from list to array

rep invariants
· not all values of the rep make legal abstract values
· prevent client from accessing rep so code of ADT can preserve invariants
· eg: sets: make sure element does not appear twice
mutable vs. immutable

two kinds of abstract types

• mutable: value of an object can change over time
• immutable: value of object is fixed on creation; object is a value

why is immutability useful?

• easier reasoning: $f(x) = f(x)$ is true
• safe concurrency: sharing does not cause races
• network objects: can send objects over the network
• performance: can exploit sharing

but mutability is useful too (hence 6.005, part 3)

• local modification can avoid copying
• cyclic structures are possible
• sometimes natural (bank account that never changes?)
defining the contract
why a contract?

code module needs

- contract between client and implementor
- if client uses appropriately, then implementor delivers expected service

in worst case can just read the code

- but this won’t work if rep is to be hidden from client
- so need to define contract independently of the code

two approaches to specifying abstract types

- axiomatic or algebraic: define by axioms on operations
- model-based: define in terms of a set of abstract values
algebraic approach

first, give the operations the client can use

• constructors: for making values of the type
  
  eg. new Set ()

• producers: for making new values from existing values
  
  eg. Set s3 = s1.addAll (s2)

• mutators: for changing the values of objects (mutable types only)
  
  eg. s1.addAll (s2)

can now define contract with axioms

• for simple types, often a good approach
  
  eg. array axiom: get(store(a,i,e), j) = (i==j) ? e : get(a,j)

  writing store(a,i,e) for a[i]=e, and get(a,i) for a[i]
model-based approach

but for more complex types

\cdot axiomatic approach is usually awkward
\cdot easier to give a “model” of the abstract type
\cdot model = set of abstract values + operations over them

example

\cdot values of type Set modelled as mathematical sets

\begin{verbatim}
Set<E> addAll (Set<E> s)
// returns (this U s)
\end{verbatim}

\cdot arrays modelled as functions from integer to element

\begin{verbatim}
store (a, i, e)
// effects a’(j) = (i==j) ? e : a(j)
using a’ for the name of the function after, and a for the function before
\end{verbatim}
interpreting types

how to check the code
  • contract talks about abstract values
  • but code uses concrete values
  • how to bridge the gap?

abstraction function
  • interprets concrete values in terms of abstract values

representation invariant
  • defines which concrete values have interpretations

more on this in the next lecture
rep invariant $R$

- defines set of legal representation values

abstraction function $A$

- interprets legal representation values as abstract values
classic abstract types
classic types

**domain specific and generic types**

- some types are specific to a domain (clause, literal)
- some have wide application (list, set)
- widely applicable types are usually polymorphic
- these are the “classic ADTs”

**in Java**

- found in the standard package `java.util`
- often called “Java collection framework”
a zoo of types

<table>
<thead>
<tr>
<th>type</th>
<th>overview</th>
<th>producers</th>
<th>observers</th>
<th>common reps</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>sequence for concatenation and front-append</td>
<td>add, append</td>
<td>first, rest, ith</td>
<td>array, linked list</td>
</tr>
<tr>
<td>queue</td>
<td>FIFO: first in, first out</td>
<td>enq, deq</td>
<td>first</td>
<td>array, list, circular buffer</td>
</tr>
<tr>
<td>stack</td>
<td>LIFO: last in, first out</td>
<td>push, pop</td>
<td>top</td>
<td>array, list</td>
</tr>
<tr>
<td>map</td>
<td>associates keys and values</td>
<td>put</td>
<td>get</td>
<td>association list, hash table, tree</td>
</tr>
<tr>
<td>set</td>
<td>unordered collection</td>
<td>insert, remove</td>
<td>contains</td>
<td>map, list, array, bitvector, tree</td>
</tr>
<tr>
<td>bag</td>
<td>like set, but element can appear more than once</td>
<td>insert, remove</td>
<td>count</td>
<td>map, array, association list</td>
</tr>
</tbody>
</table>

note

- producers and observers: just examples
- common reps: some (eg, hash table, bitvector) just for mutable versions
examples: types for SAT
literal type

literal and its operations

```java
public class Literal {
    Literal (String name) {...}
    public String getName () {...}
    public Literal getNegation () {...}
    public boolean negates (Literal literal) {...}
}
```

notes

- actually extends bool.Literal which itself extends Formula
- recall discussion of factory method: constructor not public
- methods such as eval, clausify, etc aren’t operations of the ADT

can you classify operations into producers & observers?
clause type

what’s a clause?

\[\text{clause is disjunction of set of literals; empty means } \text{FALSE, no rep of } \text{TRUE}\]

```java
public class Clause {
    public Clause() {...}
    public Clause(Literal literal) {...}
    public Clause add(Literal l) {...}
    public Clause merge (Clause c) {...}
    public Clause reduce(Literal literal) {...}
    public Literal chooseLiteral() {...}
    public boolean isUnit() {...}
    public boolean isEmpty() {...}
    public int size() {...}
}
```

notes

\[\text{order not exposed in observers: chooseLiteral is non-deterministic}\]

\[\text{isUnit, isEmpty, merge are for convenience of clients, not strictly necessary}\]

\[\text{add, reduce are the key producers:}\]

- add \(l\): return clause obtained by adding \(l\) as a disjunct
- reduce \(l\): return clause obtained by setting \(l\) to \text{TRUE}\]
the null controversy

issue

\* what should \texttt{add}, \texttt{reduce} return when result is \texttt{TRUE}?
\* could have special value for \texttt{TRUE}
\* instead, decided to return \texttt{null} on producer for \texttt{TRUE}

using null is bad

\* \texttt{null} is a special reference, not a special object
\* \texttt{e.m()} will fail, throwing an exception, if \texttt{e} is \texttt{null}
\* using \texttt{null} as a value of a type is therefore disastrous

but in this case

\* the \texttt{TRUE} clause is never used as a value: doesn’t appear in clause sets
\* so \texttt{null} is used only as a return value for \texttt{add} and \texttt{reduce}
\* other options: throw exception (too slow in this case), create special option class for \texttt{Clause} or \texttt{TRUE} (not convenient in Java)
bool type

introduced my own boolean ADT

• has three boolean values: TRUE, FALSE and UNDEFINED

• why did I do this?

```java
public enum Bool {
   TRUE, FALSE, UNDEFINED;

   public Bool and (Bool b) {
      if (this==FALSE || b==FALSE) return FALSE;
      if (this==TRUE && b==TRUE) return TRUE;
      return UNDEFINED;
   }

   public Bool or (Bool b) {
      if (this==FALSE && b==FALSE) return FALSE;
      if (this==TRUE || b==TRUE) return TRUE;
      return UNDEFINED;
   }

   public Bool not () {
      if (this==FALSE) return TRUE;
      if (this==TRUE) return FALSE;
      return UNDEFINED;
   }
}
```
should `Environment` be an ADT at all?

- just a mapping from literals to booleans
- decided yes, in case I wanted to add functionality later
- sure enough, I did: return `Bool.UNDEFINED` if no mapping

```java
public class Environment {
    private Map<Literal, Bool> bindings;
    ...
    public Bool get(Literal l){
        Bool b = bindings.get(l);
        if (b==null) return Bool.UNDEFINED;
        else return b;
    }
}
```
representation independence
choice of rep

an abstract type can be implemented with different reps

' example: two versions of Environment

```java
public class Environment {
    private Map <Literal, Bool> bindings;
    ...
    public Bool get(Literal l){
        Bool b = bindings.get(l);
        if (b==null) return Bool.UNDEFINED;
        else return b;
    }
}
```

```java
public class Environment {
    private Set <Literal> trues, falses;
    ...
    public Bool get(Literal l){
        if (trues.contains (l)) return Bool.TRUE;
        if (falses.contains (l)) return Bool.FALSE;
        return Bool.UNDEFINED;
    }
}
```
achieving rep independence

rep independence
  - want to be able to change rep without changing client

what does this require?
  - if client can access fields directly
    rep is fully “exposed”: heavy modification of client code required
  - if client calls methods that return fields directly
    can fix by modifying ADT methods, but will be ugly
  - if client can’t access fields even indirectly (as in previous slide)
    ADT is easily modified locally

so independence is achieved by
  - combination of language support and programmer discipline
iterator pattern
define a class to represent a formula in CNF

• has methods to support clausifying

• or method forms all pairwise merges of clauses

```java
public class Problem {
    List<Clause> clauses;
    ...

    public Problem or (Problem p) {
        List<Clause> result = new Empty<Clause>();
        for (Clause c1 : p.clauses)
            for (Clause c2 : clauses) {
                Clause c = c1.merge(c2);
                result = result.add(c);
            }
        return new Problem(result);
    }
}
```

how does this work?

• hidden iterator at play
the iterator pattern

a Java shorthand

• the statement
  
  for (E e: c) {...}

• is short for
  
  Iterator i = c.iterator();
  while (i.hasNext()) {
    E e = i.next();
    ...
  }

iterator interface

public interface Iterator<E> {
  boolean hasNext ();
  E next ();
  void remove ();
}

list iterator

public class ListIterator<E> implements Iterator<E> {
  List<E> remaining;
  public ListIterator (List<E> list) {
    remaining = list;
  }
  public boolean hasNext () {
    return !remaining.isEmpty();
  }
  public E next () {
    E first = remaining.first ();
    remaining = remaining.rest();
    return first;
  }
}

iterator method

public abstract class List<E> implements Iterable<E> {
  public Iterator<E> iterator () {
    return new ListIterator<E>(this);
  }
}
iterator state machine

why a stateful object in a side-effect free program?

‣ the only convenient way to do iteration in Java
‣ so long as iterator used only in for loop as shown, no mutability issues arise
alternatives to iterators

instead of iterator complexity, why not

• define observer `get(i)` to get ith element of collection?
  violates abstraction: type might not be ordered!

• define observer that returns an array?
  clumsy, and requires unnecessary copying
  also, won’t work for infinite iterator
factory pattern
last week, saw use of factory method for interning literals

• instead of constructor, client calls a static method

```java
public static T make () { return new T(); }
```

• factory method can call constructor, but can also recycle objects

```java
public abstract class Literal {
    protected Literal negation;
    protected String name;
    public Literal (String name) {this.name = name;}
}

public class Pos extends Literal {
    protected static Map<String,Pos> alloc = new ListMap<String,Pos>();
    private Pos (String name) {super(name);}
    public static Pos make (String name) {
        Pos l = alloc.get(name);
        if (l==null) {
            l = new Pos(name);
            Neg n = new Neg(name);
            l.negation = n; n.negation = l;
            alloc = alloc.put(name, l);
        }
        return l;
    }
}
```
factory object pattern

note that factory method can hide choice of class

\[\text{instead of constructor, client calls}\]
\[
\begin{align*}
\text{public static } & \text{T make ()} \{ \text{return new } \text{T1(); } \}
\end{align*}
\]
\[\text{decision to use subtype } \text{T1 is encapsulated in factory method}\]

factory pattern

\[\text{define a class whose sole purpose is generating objects}\]
\[
\begin{align*}
\text{public class } & \text{T1Factory extends TFactory} \{ \\
& \text{public T make ()} \{ \text{return new } \text{T1(); } \}
\end{align*}
\]
\[
\begin{align*}
\text{public } & \text{X client (TFactory f) } \{ \\
& \text{T t = f.make(); } \\
& \text{...} \\
& \}
\end{align*}
\]
example

scenario

\* suppose we want to support both reps for Environment
\* then we could do this:

```java
public class Environment1 implements Environment {...}
pubic class Environment2 implements Environment {...}
public class Environment1Factory implements EnvironmentFactory {
    public Environment1 make () {...}
}
pubic class Environment2Factory implements EnvironmentFactory {
    public Environment2 make () {...}
}

public class SATProblem {
    public Environment solve (EnvironmentFactory ef) {
        Environment e = ef.make (); ...}
}
public class Main {
    ...
    Environment e = p.solve (new Environment1Factory());
}
```
exercise

for the scenario on the previous slide

› draw a module dependency diagram

› how does this differ from Generator plugins in lecture 3?
hints on ADT design
two strategies

bottom-up strategy

- when you recognize the need for an ADT, define an empty one
- add observers and producers as the client code evolves
- refactor as you go, by consolidating operations and simplifying

top-down strategy

- consider the set of abstract values and operations likely to be needed
- carefully design a core set of operations

in practice, some of both needed

- can’t fully predict usage patterns, so bottom-up needed
- if bottom-up alone, often end in a mess
inside or outside?

a big consideration
• should an operation be part of the ADT?
• eg: should Environment include eval?

key issue is not whether inside module or not
• but rather whether operation accesses rep directly

a good strategy
• keep to a minimum the number of ops that access the rep
• make them simple and powerful
summary
summary

abstract data type

- characterized by operations and abstract values they take and return
- representation hidden from client, and can be changed independently of it

classic types

- study the Java collection framework
- use them freely, but as reps, not in place of your own types

important patterns

- factory (for creating objects without knowledge of rep)
- iterator (for traversing objects in collection without seeing internal structure)
lecture exercise

for Wednesday

some easy performance experiments:

‣ set your Java heap size to half the size of your memory (ask Google how)
‣ see how big a Latin square your machine can solve (see bool.Main.main)
‣ variant #1: use the visitor in the List.size method instead of the size field
‣ variant #2: make NonEmptyList.add return a copy of the list
‣ what can you learn from these experiments?

note

‣ the code’s already there: just comment/uncomment
further study

Lampson’s “Hints for Computer System Design”
  › many comments relevant to datatype design
  http://research.microsoft.com/~lampson/33-Hints/WebPage.html

Liskov & Guttag’s “Program Development in Java”
  › see especially Chapter 5

“Gang of Four” Design Pattern Book
  › Design Patterns: Elements of Reusable Object-Oriented Software, Gamma et al
  › see wikipedia on “design patterns” too

Bloch’s “Effective Java”
  › items 12-18, 23-28, and optionally 39-47

Redesigning ATC: a case study showing the value of ADTs

chapter from my 6170 lecture notes