Lists

Rob Miller
Fall 2007
Abstract data type = type (set of values) + operations

An ADT expresses a contract between implementer and clients

Client uses operations on abstract type:
- constructors
- observers
- producers

Operations have preconditions (satisfied by client) and postconditions (satisfied by implementer)

representation type $R = \text{int}$

abstract type $A = \text{subset of } \{0,\ldots,9\}$

rep invariant $RI$ describes legal reps

abstraction function $AF$ maps legal reps to abstract values

Abstraction barrier preserves representation independence
Option Types

When BigInt has multiple possible representations, it becomes an option (or union) type

BigInt = IntRep(int) \cup ArrayRep(int^N)

- An option type represents a union of disjoint sets of values
  - Unique type tags ensure disjointness, even if the internal reps are the same
  - An equivalent way to write it:
    BigInt = IntRep \cup ArrayRep
    IntRep = int
    ArrayRep = int^N
  - For example, we might implement positive & negative integers:
    SuperBigInt = Positive(BigInt) \cup Negative(BigInt)
Other Option Types

Enumerations are option types

- \( \text{RPS} = \text{Rock()} \cup \text{Paper()} \cup \text{Scissors()} \)
  - A type tag with an empty tuple is actually a singleton set: i.e., \( \text{Rock()} \) means a one-element set \( \{\text{Rock}\} \)

Option types are very useful for message-passing

- Recall our example using threads to collect sender email addresses from mail folders and send them back to the main thread
- Our solution then was to represent messages as \( \text{Set<String>} \), where:
  - the set contains a folder name as well as email addresses, e.g. \( \{\text{"Inbox"}, \text{"rcm@mit.edu"}, \text{"dnj@mit.edu"}\} \)
  - the all-done event is represented by the empty set
- With option types, we can have a much cleaner design
  - \( \text{Message} = \text{Senders(String \times Set<String>)} \cup \text{Done()} \)

```java
interface Message { ... }
class Senders implements Message { ... }
class Done implements Message { ... }
```

© Robert Miller 2007
Recursive Types

Let’s represent big integers as a list of digits, rather than a fixed-length array

\[
\text{ListRep} = \text{Digit(int x ListRep)} \cup \text{Zero()}
\]

interface ListRep extends BigInt { }

class Digit implements ListRep {
    private int d;
    private ListRep rest;
    public Digit(int d, ListRep rest) { ... }
    ...
}

class Zero implements ListRep {
    public Zero() { }
    ...
}
Dangers of Null

Why not use null to terminate the list of digits?

```java
class ListRep {
    private int d;
    private ListRep rest; // null if end of list
    ...
}
```

Null is not a useful value for an abstract data type

- Why? because you can’t call methods on it!
  - `a.plus(b)` fails if `a == null`

Using null short-circuits static typing

- An expression that is typesafe at compile-time may nevertheless at runtime have values outside the type (i.e. nulls) that cause runtime errors (i.e. NullPointerException)

Avoid null!

- Let’s exclude null from all our abstract data types:
  - so the declaration “ListRep rest” means not only that rest has type ListRep but also that rest != null
Examples of ListRep Values

```plaintext
let
's forbid reps with
new Zero()
new Digit(5, new Zero())
new Digit(5, new Digit(0, new Zero())))
new Digit(1, new Digit(3, new Zero()))

representation type R = ListRep

abstract type A = nonnegative integers
```

let’s forbid reps with leading zeros (i.e., not “05”, just “5”)

© Robert Miller 2007
Functions over option types and recursive types have special form

- Rep invariant and abstraction function are functions over the rep type
- Functions over an option type have **multiple cases**
- Functions over a recursive type are often also **recursively defined**

\[
\text{ListRep} = \text{Digit}(\text{int} \times \text{ListRep}) \cup \text{Zero()}
\]

\[
\begin{align*}
\text{AF(Digit}(d, \text{rest})) &= d + 10 \times \text{AF(rest)} \\
\text{AF(Zero())} &= 0
\end{align*}
\]

\[
\begin{align*}
\text{RI(Digit}(d, \text{rest})) &= d \geq 0 \\
&\quad \text{&& (} d == 0 \Rightarrow \text{AF(rest)} \neq 0) \\
\text{RI(Zero())} &= \text{true}
\end{align*}
\]
class Digit implements ListRep {
    private int d;
    private ListRep rest;

    public boolean isZero() {
        return false;
    }

    public boolean isOne() {
        return d == 1 && rest.isZero();
    }

    public BigInt incr() {
        if (d < 9) {
            return new Digit(d+1, rest);
        } else {
            return new Digit(0, (ListRep) rest.incr());
        }
    }

    ... 
}

class Zero implements ListRep {
    public boolean isZero() {
        return true;
    }

    public boolean isOne() {
        return false;
    }

    public BigInt incr() {
        return new Digit(1, this);
    }

    ... 
}

what would this be if the rep invariant allowed leading zeros?
Sharing in Immutable Lists

BigInt x = new Digit(8, new Digit(9, new Zero()))

BigInt y = x.incr();

BigInt z = y.incr();

Sharing is possible because ListRep is immutable

- Note also that the ArrayRep representation of BigInt couldn’t do this kind of sharing easily, because it stored the digits in an array
Exercise

Implement the following methods of Digit and Zero

➢ Your implementations should be recursive methods

```java
/**
 * @requires k >= 0
 * @return this * 10^k
 */
public BigInt timesPowerOf10(int k)

/**
 * @requires this != 0
 * @return this - 1
 */
public BigInt decr()
```
BigInt as a Sequence of Digits

We now have two “digit sequence” reps for BigInt

ArrayRep = array of digits ListRep = list of digits

Suppose we want the BigInt ADT to give access to the actual sequence of digits

One reason might be to support interoperability between reps, so that we can say x.plus(y) even if x ∈ ArrayRep and y ∈ ListRep

Suppose we represent the sequence as an array:

```java
interface BigInt {
    ...
    public int[] getDigits();
}
```

class ArrayRep impl BigInt {
    private int[] digits;
    ...
    public int[] getDigits() {
        return digits;
    }
}

class Digit impl ListRep {
    public int[] getDigits() {
        ...
    }
}
```

rep exposure! (must make a copy of the digits)
Iterator Pattern

An iterator is a representation-independent object that steps through the elements of a data structure

- In Java, an `Iterator<T>` object produces a sequence of objects of type `T`

```
interface Iterator<
      T> {
    boolean hasNext();
    T next();
  }
```

- An iterator is a state machine, not an immutable object

```
Iterator<T>
vars seq: sequence<T>
op hasNext
  pre true
  post true
  return seq != []

op next
  pre seq != []
  post seq’ = rest where seq = [first] :: rest
  return first where seq = [first] :: rest
```
interface BigInt {
    ...

    /** @return an iterator that yields the digits of the
decimal representation of this integer,
    from least significant to most significant.
e.g., if this == 3115, then the iterator
    produces [5, 1, 1, 3].
    If this == 0, yields no digits. */
    public Iterator<Integer> iterator();
}
Using Iterators

We can use BigInt’s iterator to write rep-independent operations on the digit sequence

```java
static boolean divisibleBy3(BigInt n) {
    // n is divisible by 3 if and only if the sum of its digits is
    // also divisible by 3
    int sum = 0;
    for (Iterator<Integer> g = n.iterator(); g.hasNext(); ) {
        int digit = g.next();
        sum += digit;
    }
    return (sum % 3) == 0;
}
```

> Java has a convenient “syntactic sugar” for iterators:

```java
for (int digit : n) {
    sum += digit;
}
```

> Note that n must have an iterator () method for this to work
Implementing ListRep’s Iterator

Let’s represent the state of the iterator by a reference pointer \( p \)

- We’ll step \( p \) along the list until it reaches the end (Zero)

ListRepIterator implements Iterator<Integer>

- vars \( p : \text{ListRep} \)
- op hasNext
  - pre true
  - post true
  - return \( p \not\in \text{Zero} \)
- op next
  - pre \( p \in \text{Digit} \)
  - post \( p' = p\text{.rest} \)
  - return \( p\text{.digit} \)

© Robert Miller 2007
Implementing ListRep’s Iterator

class ListRepIterator implements Iterator<Integer> {
    private ListRep p;

    public ListRepIterator(Digit p) { this.p = p; }

    public boolean hasNext() {
        return !(p instanceof Zero);
    }

    public Integer next() {
        try {
            Digit dp = (Digit) p;
            p = dp.rest;
            return dp.d;
        } catch (ClassCastException e) {
            throw new NoSuchElementException();
        }
    }
}
Exercise

Write a specification for IntRepIterator

- Recall that IntRep represents a BigInt using a single int field:
  ```java
  private int n;
  ```

IntRepIterator implements Iterator<Integer>

```java
chars n : int

op hasNext
  pre
  post
  return

op next
  pre
  post
  return
```

© Robert Miller 2007
Generic Lists

Last time we saw generic pairs

- Pair<A,B> is a general pattern for representing tuples

Now we have a general pattern for recursive list types

- List<E> = NonEmpty(E x List<E>) \cup Empty()

```java
interface List<E> {
    boolean isEmpty();
    \textit{returns} true iff list is empty

    E first();
    \textit{requires} !isEmpty()
    \textit{returns} first element

    List<E> rest();
    \textit{requires} !isEmpty()
    \textit{returns} this list with
    first elt removed

    List<E> cons(E elt);
    \textit{returns} list whose first element is elt
    and remaining elts are this list
}
```

© Robert Miller 2007
Implementing Generic Lists

class NonEmpty<E> implements List<E> {
  private E first;
  private List<E> rest;
  public NonEmpty(E first, List<E> rest) {
    this.first = first; this.rest = rest;
  }
  public boolean isEmpty() { return false; }
  public E first() { return first; }
  public List<E> rest() { return rest; }
  public List<E> cons(E elt) { return new NonEmpty(elt, this); }
}

class Empty<E> implements List<E> {
  public Empty() {
  }
  public boolean isEmpty() { return true; }
  public E first() { throw new EmptyListException(); }
  public E rest() { throw new EmptyListException(); }
  public List<E> cons(E elt) { return new NonEmpty(elt, this); }
}
Java Already Has a Mutable List

`java.util.List`:

```java
interface List<E> extends Collection<E> {
    Iterator<E> iterator();
    boolean isEmpty();
    int size();

    boolean add(E elt);
    boolean remove(E elt);
    ...
}
```

List’s specification allows these mutators to be “unimplemented” – i.e., throw `UnsupportedOperationException`
Fitting Immutable Lists into Java

Let’s make our immutable lists extend java.util.List

- That way we can use immutable lists in the same way we’d use mutable lists (at least for observer operations – can’t use mutators)

```java
interface ImList<E> extends List<E> {
    Iterator<E> iterator();
    boolean isEmpty();
    int size();

    boolean add(E elt);
    boolean remove(E elt);
    ...

    ImList<E> cons(E elt);
    E first();
    E rest();

    ImList<E> _add(E elt);
    ImList<E> _remove(E elt);
    ...
}
```

- Observers inherited from List<E>
- Mutators inherited from List<E> all throw UnsupportedOpExc
- Key operations of immutable lists
- Familiar java.util.List mutators written as producers instead

Need to change the method names to avoid conflict with existing mutators (can’t overload a method that differs only in return type)
List Processing

Let’s write some simple procedures for ImList

➤ Reversing a list

class NonEmpty ... {
    public ImList<E> reverse () {
        return rest.reverse()._add(first);
    }
}

class Empty ... {
    public ImList<E> reverse () {
        return this;
    }
}

recursive algorithm takes O(n^2) time; iterative algorithm is O(n)

ImList<E> result = new Empty<E>();
ImList<E> list = this;
while (!list.isEmpty()) {
    result = result.cons(list.first());
    list = list.rest();
}
return result;

or even simpler, because ImList has an iterator:

for (E elt : this) {
    result = result.cons(elt);
}
List Processing

Appending two lists

class NonEmpty ... {
    public ImList<E> append(ImList<E> that) {
        return rest.append(that).cons(first);
    }
}

class Empty ... {
    public ImList<E> append(ImList<E> that) {
        return that;
    }
}

x.append(y)
Summary

Option types
- A type with disjoint alternatives is expressed as $\text{Tag}_1(T_1) \cup \text{Tag}_2(T_2)$
- Option types are implemented in Java by subclassing

Recursive types
- A recursive type $T$ is expressed as $T = \text{Tag}_{\text{Recursive}}(...T...) \cup \text{Tag}_{\text{Base}}(...)$
- Recursive types often have naturally recursive method implementations

List types
- List-like recursive data types take the form
  $$\text{List} = \text{NonEmpty}(E \times \text{List}) \cup \text{Empty}()$$

Iterators
- A state machine for stepping through the elements of a data structure