Data Abstraction

Rob Miller
Fall 2007

© Robert Miller 2007
Symbolic Programming

So far we’ve viewed computation as transitions made by state machines

- Object = state machine or state
- Method = operation or state transition
- Constructor = initializer

Now we’ll be taking a symbolic view: computation as functions over data types

- Object = value of a data type (e.g. 3, “hello”, URL(http://www.mit.edu/))
- Method = function over the data type
- Constructor = creator of data type values

Key difference: in symbolic view, objects never change

- An object that always represents the same value is called immutable

© Robert Miller 2007
Data Types

Data type is a set of values

- e.g. Java has primitive data types for integers and floating point numbers

```java
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

Problem: int isn’t big enough – it can represent at most $2^{31} \approx 2$ billion.

But how big is fact(30)?

```java
static double estimateE(int n) {
    double e = 0;
    for (int i = 0; i < n; ++i) {
        e += 1.0 / fact(i);
    }
    return e;
}
```

Problem: double isn’t precise.

It can represent \(\frac{1}{2}\) and \(\frac{1}{4}\) exactly, but not \(\frac{1}{6}\).
Let’s represent an integer as an array of digits

\[ \text{BigInt} = \text{int}^N \text{ for some fixed } N > 0 \]

- If the largest number we need is 100 digits, we’ll set \( N = 100 \)
- For now, BigInt will only represent \textbf{nonnegative} integers -- that’s all we need for \texttt{fact()} and \texttt{estimateE()}.

Use a class \texttt{BigInt} to represent the data type

```java
public class BigInt {
    private int[] digits;
    ...
}
```

- Instances of \texttt{BigInt} are \textbf{values} of the data type
- Methods are \textbf{functions} over the data type

```java
public boolean isZero() { ...} // isZero: BigInt \rightarrow boolean
public BigInt incr() { ...}    // incr: BigInt \rightarrow BigInt
```
Immutability

BigInt is immutable

➢ Once a BigInt instance is created, it represents the same integer forever
➢ Primitive data types in Java are immutable too:

```java
int x = 5;
int y = x;
x++; // no effect on y
BigInt x = new BigInt("5");
BigInt y = x;
x = x.incr(); // no effect on y
```

To increment immutable BigInt, we have to make a new BigInt:

```java
public BigInt incr() {
    int[] newDigits = ...;
    // compute new digits
    return new BigInt(newDigits);
}
```

Contrast with increment as a state transition:

```java
public void incr() {
    // change digits array
}
```

We don’t want this: integers are eternal values, not state machines
Immutability Considered

A data type $T$ is **immutable** if every instance of $T$ represents the same value over its entire lifetime.

**Immutability is a powerful idea**

- Much easier to reason about immutable objects, since they never change
- **Sharing** immutable objs is always safe (e.g. with other modules or threads)
- **Copying** immutable objs is also safe (e.g. across the network)
- Sharing saves unnecessary duplicate storage & unnecessary computation

**But it has some costs**

- New values require new storage; old values must be garbage-collected
- Cyclic data structures are impossible (e.g. doubly-linked lists)
Using BigInt

Client code like fact() can interact with BigInt only through its public methods

class BigInt {
    public BigInt(String s) {...}
    public boolean isZero() {...}
    public boolean isOne() {...}
    public BigInt incr() {...}
    public BigInt decr() {...}
    public BigInt plus(BigInt that) {...}
    public BigInt minus(BigInt that) {...}
    public BigInt times(BigInt that) {...}
    public BigInt divide(BigInt that) {...}
}

static BigInt fact(BigInt n) {
    if (n.isZero()) return new BigInt(“1”);
    else return n.times(fact(n.decr()));
}

BigInt is an abstract data type

© Robert Miller 2007
**Abstract Data Type**

An abstract data type $T$ is a data type defined by its operations

- Recall that a data type is a set; **operations** are functions over that set.
- The values of the data type are **opaque** — you can apply operations to them, but you can’t otherwise inspect them.
- BigInt actually represents the abstract set of (nonnegative) integers.

### Operations

<table>
<thead>
<tr>
<th>Type</th>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>$10^{17}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **constructors** $t,... \rightarrow T$
  - `new BigInt : String \rightarrow BigInt`
  - `new BigInt : int[] \rightarrow BigInt`

- **observers** $T,... \rightarrow t$
  - `isZero : BigInt \rightarrow boolean`
  - `isOne : BigInt \rightarrow boolean`
  - `incr : BigInt \rightarrow BigInt`
  - `decr : BigInt \rightarrow BigInt`

- **producers** $T,... \rightarrow T$
  - `plus : BigInt, BigInt \rightarrow BigInt`
  - `minus : BigInt, BigInt \rightarrow BigInt`
  - `times : BigInt, BigInt \rightarrow BigInt`
  - `divide : BigInt, BigInt \rightarrow BigInt`

© Robert Miller 2007
Contracts

An abstract data type is a contract between client and implementor

- Client (e.g. fact): the values will behave like nonnegative integers as long as fact uses only the given operations and uses them correctly
  
  \[
  \text{decr: } \text{BigInt} \rightarrow \text{BigInt} \\
  \text{requires} \! \text{this.isZero()} \leftarrow \text{precondition} \\
  \text{returns} \text{this - 1} \leftarrow \text{postcondition}
  \]

- Implementor (e.g. BigInt): as long as the values behave like nonnegative integers, BigInt can use any representation it wants

Representation independence

- As long as clients and implementer all obey the contract, BigInt can change its representation without making any changes to the clients
Rep Exposure

Failure to ensure representation independence is called rep exposure

- If you expose details of your representation through your operations, clients may come to depend on those details, and you won’t be able to change them easily.

Oops: one of BigInt’s constructors exposes the rep

```java
public BigInt(int[] digits) {...}
```

- Exposes not only that BigInt contains an int array, but also the order of the array, the number of digits, the fact that it’s decimal base, ...

- Let’s fix this with more abstract constructors:

```java
private
public BigInt(int[] digits) { ... }
public BigInt(int n) { ... }
public BigInt(String s) { ... }
```

© Robert Miller 2007
Implementing BigInt

Fill in the methods below

class BigInt {
    private int[] digits;
    private static final int N = 100;
    public BigInt(String s) {

    }

    public boolean isOne() {

    }
}

➢ What do these methods need to know about the digits array that its type alone (int $N$) doesn’t say?

➢ Can you think of a digits array for which your implementation of isOne() will give the wrong answer?
Representation vs. Abstraction

Abstract data type has a representation
- e.g. BigInt uses an array of integers as its internal representation
- Each representation value maps to some value in the abstract type

Representation type $R$
- $= \text{int}^N$

Abstract type $A$
- $= \text{nonnegative integers}$
Representation Type Isn’t Enough

Knowing the representation type R isn’t enough

- To understand how an abstract data type is implemented, and to implement the operations correctly, you need more than just the types of R (int[]) and A (nonnegative integers)

You also need to know:

- Which values of R are legal representations? (rep invariant)
  - int array should have exactly N elements
  - every element should be nonnegative (otherwise isOne() is hard!)
  - every element should be less than 10

- How does each legal rep map to a value of A? (abstraction function)
  - the array is ordered with the least-significant digit in digits[0]
  - digits represent a number in decimal form, so digits[i]
    contributes digits[i]*10^i

© Robert Miller 2007
**Rep Invariant**

**Rep invariant is the subset of R that are legal reps**

- Often written as a predicate $R_I : R \rightarrow \text{boolean}$, where $R_I(r)$ is true if and only if $r$ is a legal rep value
- Suppose number of digits $N=100$

Then the rep invariant we want is

$$R_I(\text{digits}) = \text{digits.length} == N$$

$$\&\& \text{ for all } 0 \leq i < N . \ 0 \leq \text{digits}[i] < 10$$

© Robert Miller 2007
Abstraction Function

Abstraction function describes how to map a legal representation value to an abstract value.

- Written as a function $AF: \mathbb{R} \rightarrow \mathbb{A}$
  \[
  AF(digits) = \sum_{0 \leq i < N} digits[i] \cdot 10^i
  \]

**Abstraction function is:**

- **usually partial**
  - (doesn’t have to map illegal values of $\mathbb{R}$)
- **ideally onto**
  - (maps to every value of $\mathbb{A}$)
- **maybe not one-to-one**
  - (some values of $\mathbb{A}$ may be mapped to by multiple values of $\mathbb{R}$)
Checking Rep Invariants

Rep invariant isn’t just a neat mathematical idea

- If your implementation asserts the rep invariant at run time, then you can catch bugs early

```java
private void checkRep() {
    assert digits != null;
    assert digits.length == N;
    assert allDigits();
}
```

```java
private boolean allDigits() {
    for (int d:digits) {
        if (d < 0 || d >= 10) return false;
    }
    return true;
}
```

```java
public BigInt(String s) {
    ...
    checkRep();
}
```

checkRep() is a direct translation of the rep invariant into code

assert is a Java statement that throws an unchecked exception if the expression is false

call checkRep() in each constructor (why don’t you need to call it anywhere else?)
Implementing the Abstraction Function

Data types inherit a method from Object:

```java
public String toString() {...}
/** @returns abstract value of this, as a string */
```

`toString()` is a useful place to implement the abstraction function

```java
public String toString() {
    String s = "";
    for (int d: digits) { s = d + s; }
    return s;
}
```

➢ What’s wrong with this implementation?
  • Hint: what would you get from
    ```java
    new BigInt("1").toString()
    ```
➢ How would you fix it?
Summary

Abstract data type (ADT)
- a set of values with operations

Immutability
- Immutable object represents the same abstract value for its entire lifetime

Preconditions and postconditions
- Contract between client of ADT and implementer of ADT

Representation independence vs. rep exposure
- Representation details should not leak through the operations

Rep invariant & abstraction function
- RI describes legal reps, AF describes how to map rep to abstract value
- Implement them with checkRep() and toString()