CSE 331
Software Design & Implementation

Hal Perkins
Spring 2017
Abstraction Functions
Connecting implementations to specs

**Representation Invariant**: maps Object $\rightarrow$ boolean
- Indicates if an instance is *well-formed*
- Defines the set of valid concrete values
- Only values in the valid set make sense as implementations of an abstract value
- **For implementors/debuggers/maintainers of the abstraction:** no object should ever violate the rep invariant
  - Such an object has no useful meaning

**Abstraction Function**: maps Object $\rightarrow$ abstract value
- What the data structure *means* as an abstract value
- How the data structure is to be interpreted
- Only defined on objects meeting the rep invariant
- **For implementors/debuggers/maintainers of the abstraction:**
  Each procedure should meet its spec (abstract values) by “doing the right thing” with the concrete representation
Rep inv. constrains structure, not meaning

An implementation of `insert` that preserves the rep invariant:

```java
public void insert(Character c) {
    Character cc = new Character(encrypt(c));
    if (!elts.contains(cc))
        elts.addElement(cc);
}
```

```java
public boolean member(Character c) {
    return elts.contains(c);
}
```

Program is still wrong

- Clients observe incorrect behavior
- What client code exposes the error?
- Where is the error?
- We must consider the `meaning`
- The `abstraction function` helps us

```
CharSet s = new CharSet();
s.insert('a');
if (s.member('a'))
    ...
```
Abstraction function: \( \text{rep} \rightarrow \text{abstract value} \)

The \textit{abstraction function} maps the concrete representation to the abstract value it represents.

\[ \text{AF: } \text{Object} \rightarrow \text{abstract value} \]

\[ \text{AF(CharSet this)} = \{ c \mid c \text{ is contained in this.elts} \} \]

“set of Characters contained in this.elts”

Not executable because abstract values are “just” conceptual.

The abstraction function lets us reason about what [concrete] methods do in terms of the clients’ [abstract] view.
Abstraction function and **insert**

Goal is to satisfy the specification of **insert**:

```java
// modifies: this
// effects: this_post = this_pre \cup \{c\}
public void insert (Character c) {...}
```

The AF tells us what the rep means, which lets us place the blame

```
AF(CharSet this) = \{ c \mid c \text{ is contained in this.elts} \}
```

Consider a call to **insert**:

- **On entry**, meaning is \(AF(this_{pre}) = elts_{pre}\)
- **On exit**, meaning is \(AF(this_{post}) = AF(this_{pre}) \cup \{\text{encrypt('a')}\}\)

What if we used this abstraction function instead?

```
AF(this) = \{ c \mid \text{encrypt(c) is contained in this.elts} \}
= \{ \text{decrypt(c)} \mid c \text{ is contained in this.elts} \}
```
The abstraction function is a function

Why do we map concrete to abstract and not vice versa?

• It’s not a function in the other direction
  – Example: lists \([a, b]\) and \([b, a]\) might each represent the set \(\{a, b\}\)

• It’s not as useful in the other direction
  – Purpose is to reason about whether our methods are manipulating concrete representations correctly in terms of the abstract specifications
Stack AF example

Abstract stack with array and “top” index implementation

<table>
<thead>
<tr>
<th>new()</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(17)</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>push(-9)</td>
<td>17</td>
<td>-9</td>
<td>0</td>
</tr>
<tr>
<td>pop()</td>
<td>17</td>
<td>-9</td>
<td>0</td>
</tr>
</tbody>
</table>

stack = <>

stack = <17>

stack = <17, -9>

Abstract states are the same
stack = <17> = <17>

Concrete states are different
<[17, 0, 0], top=1> ≠ <[17, -9, 0], top=1>

AF is a function
Inverse of AF is not a function
Benevolent side effects

Different implementation of `member`:

```java
boolean member(Character c1) {
    int i = elts.indexOf(c1);
    if (i == -1)
        return false;
    // move-to-front optimization
    Character c2 = elts.elementAt(0);
    elts.set(0, c1);
    elts.set(i, c2);
    return true;
}
```

- Move-to-front speeds up repeated membership tests
- Mutates rep, but does not change abstract value
  - *AF maps both reps to the same abstract value*
    - Precise reasoning/explanation for “clients can’t tell”
For any correct operation…
Writing an abstraction function

**Domain:** all representations that satisfy the rep invariant

**Range:** can be tricky to denote

- For mathematical entities like sets: easy
- For more complex abstractions: give names to specification
  - AF defines the value of each “specification field”
    - (Course notes have examples of complex AFs with many spec. fields, but it’s possible to be too complex – go for simple, correct, understandable whenever possible)

Overview section of the specification should provide a notation for writing abstract values

- Could implement a method for printing in this notation
  - Useful for debugging
  - Often a good choice for `toString`
Data Abstraction: Summary

Rep invariant
  – Which concrete values represent abstract values

Abstraction function
  – For each concrete value, which abstract value it represents

Together, they modularize the implementation
  – Neither one is part of the ADT’s specification
  – Both are needed to reason that an implementation satisfies the specification

In practice, representation invariants are documented more often and more carefully than abstraction functions
  – A more widely understood and appreciated concept