Understanding an ADT implementation: Abstraction functions

CSE 331
University of Washington

Michael Ernst
Outline of data abstraction lectures

ADT specification

Abstract data type

Today Abstraction function (AF): Relationship between ADT specification and implementation

ADT implementation

Implementation (e.g., Java class)

Representation invariant (RI): Relationship among implementation fields

Abstraction barrier
Review: Connecting specifications and implementations

Representation invariant: Object → boolean
- Indicates whether rep/instance is well-formed
- Defines the set of valid values of the data structure
- Only well-formed representations are meaningful

Abstraction function: Object → abstract value
- What the rep/instance means as an abstract value
- How the rep/instance is to be interpreted

Used by implementors/maintainers of the abstraction
Abstraction function: rep $\rightarrow$ abstract value

The *abstraction function* maps the concrete representation to the abstract value it represents.

AF: Object $\rightarrow$ abstract value

AF(CharSet this) = \{ c | c is contained in this.elts \}

“set of Characters contained in this.elts”

Typically *not* executable (Why?)

The abstraction function lets us reason about concrete method behavior *from the client (abstract) perspective*
Rep invariant constrains structure, not meaning

An implementation of `insert` that preserves the rep invariant (no nulls or duplicates in `elts`):

```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);
}
```

Implementation is still wrong; this client code observes incorrect behavior:

```java
CharSet s = new CharSet();
s.insert('a');
if (s.member('a'))
    ...
```
Abstraction function and insert impl.

Our real goal is to satisfy the specification of insert:

```java
// modifies: this
// effects: this_post = this_pre U {c}
public void insert(Character c);
```

The AF tells us what the rep means (and lets us place the blame)

```java
AF(CharSet this) = { c | c is contained in this.elts }
```

Consider a call `insert('a')`:

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

What if we used this abstraction function instead?

```java
AF(this) = \{ c | encrypt(c) is contained in this.elts \}
= \{ decrypt(c) | c is contained in this.elts \}
```
The abstraction function: concrete $\rightarrow$ abstract

Q: Why don’t we use the inverse of the AF? What function maps abstract to concrete?

1. It’s not a function in the other direction. E.g., lists [a,b] and [b,a] each represent the set {a, b}

2. To go from abstract to concrete, just construct and modify objects via the provided operators.

3. Not helpful in reasoning about impl correctness.

A function maps each argument to at most one value.
Multiple reps for the same abstract value

Stack rep:
```java
int[] elements;
int top; // first unused index
```

```
new Stack()
stack = <>
push(17)
stack = <17>
push(-9)
stack = <17,-9>
pop()
stack = <>
```

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
AF⁻¹ is not a function
**Benevolent side effects**

Different implementation of member:

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

Example: \{ a, c, i, n, o, t, u \} = AF(auction) = AF(caution)
For any correct operation
Writing an abstraction function

The **domain**: all representations that satisfy the rep invariant

The **range**: can be tricky to denote
  - For mathematical entities like sets: easy
  - For more complex abstractions: give them fields
    - AF defines the value of each “specification field”
    - For “derived specification fields”, see the handouts

The overview section of the specification should provide a way of writing abstract values
  - This printed representation is valuable for debugging
    - `toString`
ADTs and Java language features

• Java classes
  – Make operations in the ADT public
  – Make other operations and fields of the class private
  – Clients can only access ADT operations

• Java interfaces
  – Clients only see the ADT, not the implementation
  – Multiple implementations have no code in common
  – Cannot include creators (constructors) or fields

• Both classes and interfaces are sometimes appropriate
  – Write and rely upon careful specifications
  – Prefer interface types instead of specific classes in declarations (e.g., List instead of ArrayList for variables and parameters)
Connecting ADTs to implementations: Summary

Rep invariant
- Which concrete values represent abstract values

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

Neither one is part of the abstraction (the ADT)
Use both to reason that an implementation satisfies the specification
- They modularize the implementation
- Can examine operators one at a time

When you program:
- Always write a rep invariant (standard industry best practice)
- Write an abstraction function when you need it
  - Write an informal one for most non-trivial classes
  - A formal one is harder to write and often less useful
    - Helps with reasoning and debugging
Invariants simplify reasoning

• Why focus so much on invariants (properties of code that do not change)?
• Why focus so much on immutability (a specific kind of invariant)?

• Software is complex – invariants/immutability reduce the intellectual complexity
• If we can assume some property remains unchanged, we don’t have to worry about it
• Reducing what we need to think about can be a huge benefit