Understanding an ADT implementation: Abstraction functions

CSE 331
University of Washington

Michael Ernst

Review: Connecting specifications and implementations

Representation invariant: Object → boolean

- Indicates whether a data structure is well-formed
  - Only well-formed representations are meaningful
- Defines the set of valid values of the data structure

Abstraction function: Object → abstract value

- What the data structure means (as an abstract value)
- How the data structure is to be interpreted
- How do you compute the inverse, abstract value → Object ?

Abstraction function: rep → abstract value

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

AF: Object → abstract value

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

“set of Characters contained in this.elts”

Typically *not* executable

The abstraction function lets us reason about behavior from the client perspective

Abstraction function and insert impl.

Our real goal is to satisfy the specification of insert:

```
// 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)

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

Consider a call to insert:

On entry, the meaning is AF($this_{pre}$) ≈ $elts_{pre}$

On exit, the meaning is AF($this_{post}$) = AF($this_{pre}$) U {encrypt('a')}

What if we used this abstraction function?

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

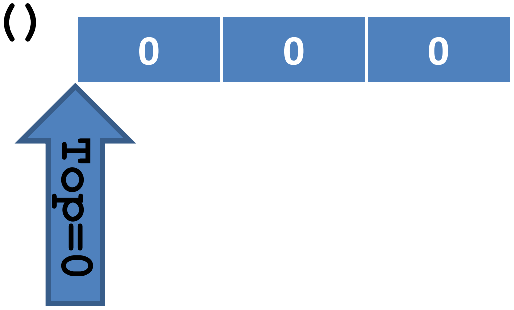
= { decrypt(c) | c is contained in this.elts }

Stack example

Stack rep:

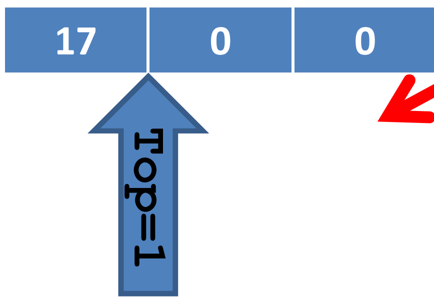
```
int[] elements;
int top; // first unused index
```

`new Stack()`



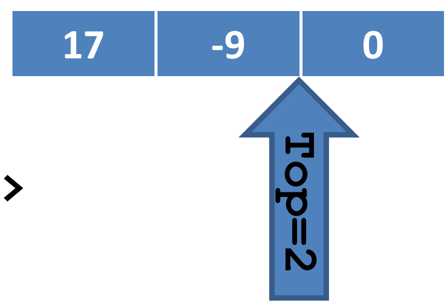
`stack = <>`

`push(17)`



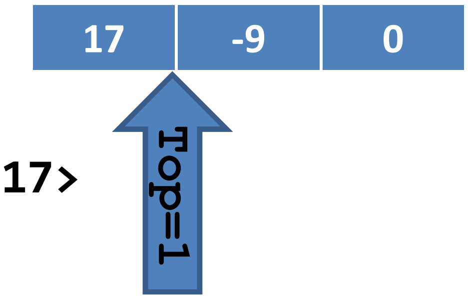
`stack = <17>`

`push(-9)`



`stack = <17,-9>`

`pop()`



`stack = <17>`

Abstract states are the same

`stack = <17> = <17>`

Concrete states are different

`<[17,0,0], top=1>`

≠

`<[17,-9,0], top=1>`

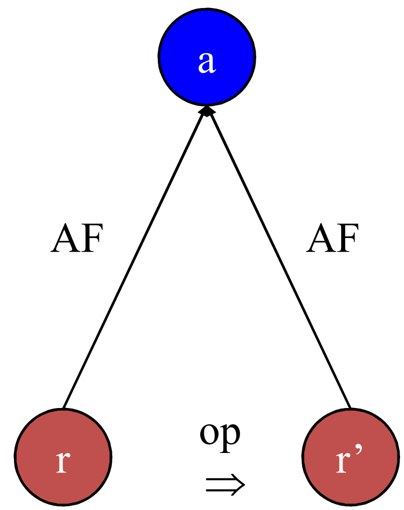
AF is a function

AF^{-1} is not a function

Benevolent side effects

Different implementation of member:

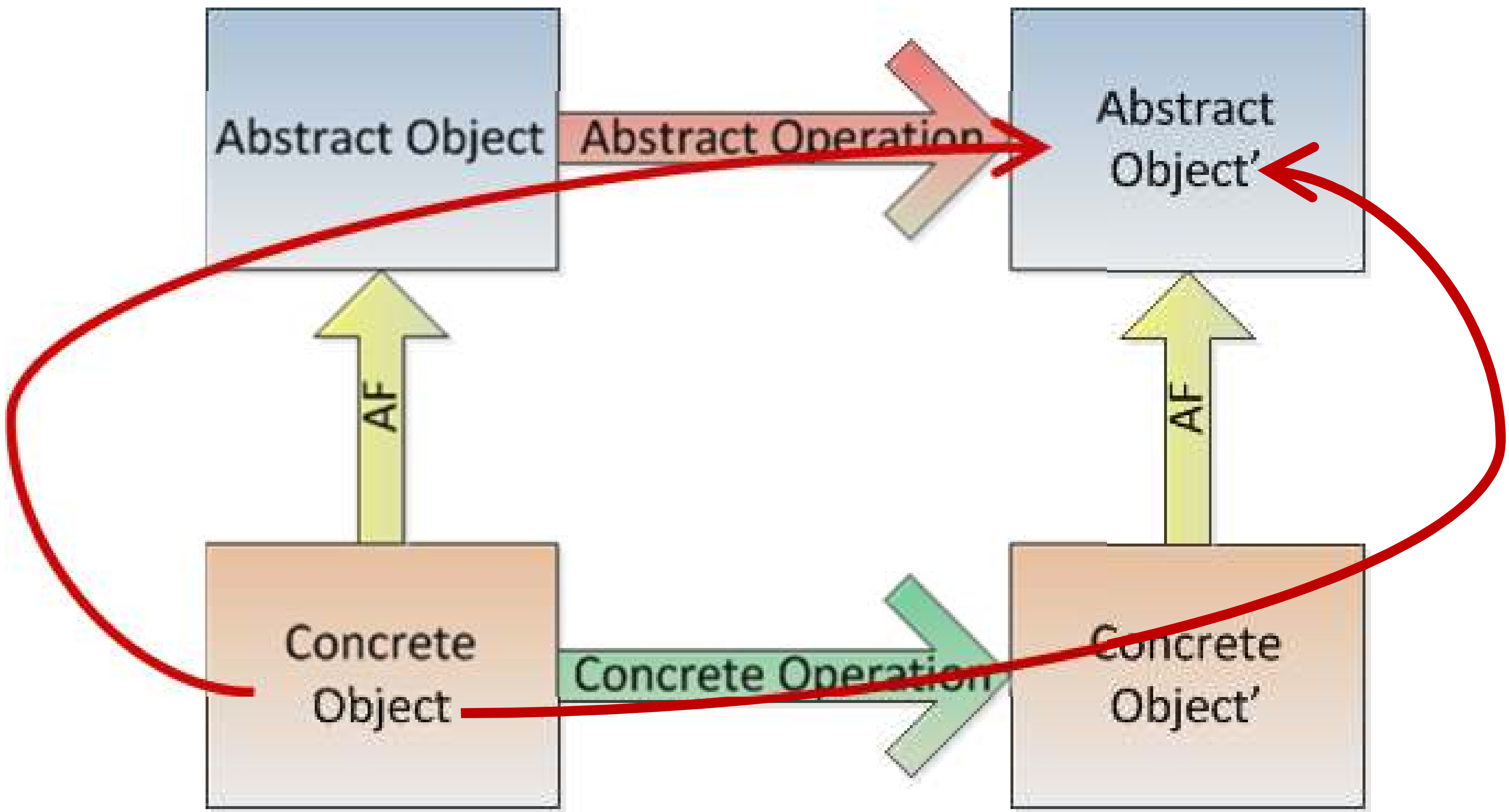
```
boolean member(Character c1) {
  int i = elts.indexOf(c1);
  if (i == -1)
    return false;
  // move-to-front optimization
  Character c2 = elts.elementAt(0);
  elts.set(0, c1);
  elts.set(i, c2);
  return true;
}
```



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
- Example: AF(a u c t i o n) = { a, c, i, n, o, t, u } = AF(c a u t i o n)
- Example: AF(s h r u b) = { b, h, r, s, u } = AF(b r u s h)



Creating the concrete object:

- Establishes the rep invariant
- Establishes the abstraction function

Every operation:

- Maintains the rep invariant
- Maintains the abstraction function

Why is each of these properties important?

The abstraction function: concrete → abstract

Q: Why do we map concrete to abstract rather than vice versa?

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. It's not as useful in the other direction.

   Can construct objects via the provided operators

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

- A printed representation is valuable for debugging

ADTs and Java language features

- Java classes
  - Make operations in the ADT public
  - Make other operationss 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)

Implementing an ADT: 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

- Can examine operators one at a time
- Neither one is part of the abstraction (the ADT)

In practice

- Always write a representation invariant
- 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

A half-step backwards

- Why focus so much on invariants (properties of code that do not – or are not supposed to – 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 can consider other properties instead
- Reducing what we need to think about can be a huge benefit