Specifications, conclusion.
Abstract Data Types (ADTs)

Based on notes by Michael Ernst,
University of Washington

Announcements

- Quiz 1 and 2 graded
  - Grades in LMS
- Labs 1 and 2 graded
  - Feedback in Homework server
- Currently grading HW0 and HW1
  - Feedback in Homework server
  - Grades in Homework server and the LMS

Announcements

- HW2 due on Friday, Sep. 25th
  - Commit to SVN then
  - Submit through Homework Server!
- HW3 out on Friday

So Far

- Specifications
  - Benefits of specifications
  - Specification conventions
  - Javadocs, PoS specifications, JML/Dafny
- Specification style
- Specification strength
- Comparing specifications

Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
- Specifying an ADT
  - Immutable
  - Mutable
  - The ADT design methodology
- Next: reasoning about ADT implementations

Specification Strength

- "A is stronger than B" (or A => B) means
  - For every implementation I
    - "I satisfies A" implies "I satisfies B"
    - The opposite is not necessarily true!
    - Larger world of Is satisfies the weaker spec
  - For every client C
    - "C works with B" implies "C works with A"
    - The opposite is not necessarily true!
    - Larger world of clients works with stronger spec
- A stronger spec is harder to implement
- A stronger spec is easier to use
Why Care About Specification Strength?

- Because of substitutability!

- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - i.e., an implementation that satisfies a stronger specification, can be used in a client that expects a weaker specification

Substitutability

- Substitutability ensures correct hierarchies
- Client code: \( X \ x = \ldots \ . \ x.\ foo(\text{index}) \);
  - Client is “polymorphic”: written against \( X \), but is expected to work with any subclass of \( X \)
  - A subclass of \( X \), say \( Y \), may have its own implementation of \( \text{foo} \), \( Y.\ foo(\text{int}) \). Client must work correctly with \( Y.\ foo(\text{int}) \) too!
  - If the spec of \( Y.\ foo(\text{int}) \) is stronger than the spec of \( X.\ foo(\text{int}) \) then we can safely substitute \( Y.\ foo(\text{int}) \) for \( X.\ foo(\text{int}) \)!

Comparison by Logical Formulas

- The following is a sufficient condition:
  - If \( P_B \Rightarrow P_A \) and \( Q_A \Rightarrow Q_B \) then \( A \) is stronger than \( B \)
  - \( P_B \Rightarrow P_A \) and \( Q_A \Rightarrow Q_B \)
  - A is stronger than \( B \)
  - Too strong a requirement!

Simple Example

Spec A: requires: 0 <= arg
  returns: 1 <= result <= 10
Spec B: requires: -1 <= arg
  returns: 2 <= result <= 5
Spec C: requires: true // the weakest condition
  returns: 2 <= result <= 5
Spec D: requires: 1 <= arg <= 10
  returns: 1 <= result <= 20

Example: int find(int[] a, int val)

- Specification B:
  - requires: a is non-null and \( \text{val occurs in a} \ [P_B] \)
  - returns: i such that \( a[i] = \text{val} \ [Q_B] \)
- Specification A:
  - requires: a is non-null \([P_A]\)
  - returns: i such that \( a[i] = \text{val if value occurs in a and -1 if value is not in a} \ [Q_A] \)

Clearly, \( P_B \Rightarrow P_A \)
But \( Q_A \), which states
"\text{val occurs in a} \Rightarrow \text{returns i such that a[i] = val AND val does not occur in a} \Rightarrow \text{returns -1}" does not imply \( Q_B \)!
Comparing by Logical Formulas

- (I satisfies specification A) is a logical formula: \( P_A \Rightarrow Q_A \)
  \((P_A \text{ is the precondition of } A, Q_A \text{ is the postcondition of } A)\)

- Spec A is stronger than spec B if and only if for each implementation I, \((I \text{ satisfies } A) \Rightarrow (I \text{ satisfies } B)\)
  which is equivalent to \( A \Rightarrow B \)

- A is stronger than B iff \((P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B)\)

Recall from FoCS and/or Intro to Logic: \( p \Rightarrow q \equiv \neg p \lor q \)

Example: \textbf{int find(int[] a, int val)}

- Specification B:
  - requires: \( a \text{ is non-null and } \text{val} \text{ occurs in } a \ [P_B] \)
  - returns: \( i \text{ such that } a[i] = \text{val} \ [Q_B] \)

- Specification A:
  - requires: \( a \text{ is non-null [P_A]} \)
  - returns: \( i \text{ such that } a[i] = \text{val} \text{ if val occurs in a and } -1 \text{ if val does not occur in } a \ [Q_A] \)

\( P_B \Rightarrow P_A \) (P_B includes P_A and one more condition)
Now, let’s show \( P_B \land Q_A \Rightarrow Q_B \)
\( P_B \) implies “\text{val} \text{ occurs in } a”. \( Q_B \) states “\text{val} \text{ occurs in } a \Rightarrow \text{return } i \text{ s.t. } a[i] = \text{val} \).
\( P_B \land Q_A \Rightarrow \text{returns } i \text{s.t. } a[i] = \text{val}, \text{ precisely } Q_B \)

Comparing by Logical Formulas

- \((P_A \Rightarrow Q_A) \Rightarrow (P_B \Rightarrow Q_B) = \)
  \( \neg(P_A \Rightarrow Q_A) \lor (P_B \Rightarrow Q_B) = \) [ due to law \( p \Rightarrow q \equiv \neg p \lor q \)]
  \( (P_A \lor \neg Q_A) \lor (P_B \Rightarrow Q_B) = \) [ due to \( p \Rightarrow q \equiv \neg p \lor q \)]
  \( (P_A \land \neg Q_A) \lor (P_B \lor Q_B) = \) [ due to \( p \lor q \equiv \neg p \land q \)]
  \( (P_B \lor Q_B) \lor (P_A \land Q_A) \land \) [ distributivity ]
  \[ \neg Q_B \Rightarrow (Q_B \Rightarrow P_A) \land \) [ due to \( p \Rightarrow q \equiv \neg p \lor q \)]

\( A \text{ is stronger than } B \) if and only if \( P_B \Rightarrow Q_B \) is true trivially or \( P_B \) implies \( P_A \) AND \( Q_B \) together with \( P_B \) imply \( Q_A \) (i.e., for the inputs permitted by \( P_B \), \( Q_B \) holds)

Example: \textbf{int find(int[] a, int val)}

- Specification B:
  - requires: \( a \text{ is non-null and } \text{val} \text{ occurs in } a \ [P_B] \)
  - returns: \( i \text{ such that } a[i] = \text{val} \ [Q_B] \)

- Specification A:
  - requires: \( a \text{ is non-null [P_A]} \)
  - returns: \( i \text{ such that } a[i] = \text{val} \text{ if val occurs in a and } -1 \text{ if val does not occur in } a \ [Q_A] \)

Intuition: \( Q_A \), by itself, does not imply \( Q_B \) because \( A \) may return \(-1\). But \( P_B \) and \( Q_B \) imply \( Q_A \). Thus, it’s still ok to substitute \( A \) for \( B \). The client, written against \( B \), meets the precondition \( P_B \). \( P_A \) ensures the “right branch” of \( Q_A \) which subsumes \( Q_B \).

Converting PoS Specs into Logical Formulas

- PoS specification has 5 clauses
  - Precondition:
    - requires: …
  - Postcondition:
    - modifies: …, effects: …, returns: …, throws: …
- To reason about PoS specs, we must first convert specs into \( P \Rightarrow Q \) logical formulas
  - Step 1: absorbs \text{returns} and \text{throws} into effects
  - Step 2: converts into logical formula

Converting PoS Specs into Logical Formulas

- PoS specification
  - requires: \( R \)
  - modifies: \( M \)
  - effects: \( E \)
- is equivalent to this logical formula
  \( R \Rightarrow (E \land \text{(nothing but } M \text{ is modified)}) \)
- throws and returns are absorbed into effects \( E \)
Convert Spec to Formula, step 1:
absorb throws and returns into effects

- PoS specification convention
  
  requires: (unchanged)
  modifies: (unchanged)
  effects: absorbed into effects
  returns: throws:

\[ \text{set from java.util.ArrayList<T>}
\]

\[ T \text{ set(int index, T element)} \]

\[ \text{requires: true} \]

\[ \text{modifies: this[index]} \]

\[ \text{effects: this[index]} = \text{element} \]

\[ \text{throws: IndexOutOfBoundsException if index < 0 || index \geq \text{size}} \]

\[ \text{returns: this[index]} \]

Absorb effects, returns and throws into new effects:
if index < 0 || index \geq \text{size} then throw IndexOutOfBoundsException
else \this\[\text{index}\] = \text{element} and returns \this\[\text{index}\]

Exercise

- Convert spec into logical formula

\[ \text{set from java.util.ArrayList<T>}
\]

\[ T \text{ set(int index, T element)} \]

\[ \text{requires: true} \]

\[ \text{modifies: this[index]} \]

\[ \text{effects: if index < 0 || index \geq \text{size} then throws IndexOutOfBoundsException}
\]

\[ \text{else } \]

\[ \this\[\text{index}\] = \text{element} \]

\[ \text{and returns this[\text{index}]} \]

Denote effects expression by \( E \). Resulting formula is:
\[ \text{true} \implies (E \land (\text{foreach } i \neq \text{index}, \this\[i\] = \this\[\text{index}\})) \]

Exercise

- Convert PoS spec into logical formula

\[ \text{public static int binarySearch(int[]} \text{a}, \text{int key)} \]

\[ \text{requires: a is sorted in ascending order} \]

\[ \text{modifies: none} \]

\[ \text{effects: none} \]

\[ \text{returns: i such that a[i] = key if such an i exists; -1 otherwise} \]

Exercise

- Convert spec into logical formula

\[ \text{static void listAdd2(List,Integer} \text{lst1, List,Integer} \text{lst2)} \]

\[ \text{requires: lst1, lst2 are non-null. lst1 and lst2 are same size.} \]

\[ \text{modifies: lst1} \]

\[ \text{effects: i-th element of lst1 is replaced with the sum of i-th elements of lst1 and lst2} \]

\[ \text{returns: none} \]

Exercise

- Convert spec into logical formula

\[ \text{private static void swap(int[] a, int i, int j)} \]

\[ \text{requires: a non-null, 0 \leq i, j < a.length} \]

\[ \text{modifies: a[i] and a[j]} \]

\[ \text{effects: a_{post}[i]=a_{pre}[j] and a_{post}[j]=a_{pre}[i]} \]

\[ \text{returns: none} \]

\[ \text{static void swap(int[] a, int i, int j)} \{
\]

\[ \text{int tmp = a[j];}
\]

\[ a[j] = a[i];
\]

\[ a[i] = tmp;
\}

\]
Specifications, Review

- Benefits
- Conventions
  - Javadocs, PoS specifications, JML
- Style
- Strength
- Comparing specifications
  - By hand, by logical formulas
- Converting PoS specs into logical formulas

Outline

- Comparing specifications with logical formulas
  - What is an abstract data type (ADT)?
  - Specifying an ADT
    - Immutable
    - Mutable
  - The ADT design methodology
  - Next: reasoning about ADT implementations

Abstraction

- Abstraction: hiding unnecessary low-level details
- Control abstraction (procedural abstraction)
  - A procedure (method) abstracts the details of an algorithm
  - One part of abstraction: signature, provides name, parameter types, return type. Not enough!
  - Another part: specification, provides detail about behavior and effects
    - E.g., \texttt{int binarySearch(int[] a, int key)}
    - Reasoning about code connects implementation to specification

Abstract Data Types are Important

- ADTs are about organizing and manipulating data
- Organizing and manipulating data is pervasive. Inventing and describing algorithms is not
- Start your design by designing data structures. Write code to access and manipulate data
- Chose data structures carefully!

ADT is a way of thinking about programs and design

- From domain concept
  - E.g., the math concept of the polynomial, the integer set, the concept of a library item, etc.
- through ADT
  - Describes domain concept in terms specification fields and abstract operations
- to implementation
  - Implements ADT with representation fields and concrete operations
Example: Polynomial with Int Coefficients, Domain Concept & ADT

ADT:
Overview description:
A Poly is an immutable polynomial with int coefficients. A Poly is:
\[ c_0 + c_1 x + c_2 x^2 + \ldots \]

Set of abstract operations:
add, mul, eval, etc. with PoS style specs referencing abstract specification fields.

Another Example: A Meeting, Domain Concept & ADT

ADT:
Overview description:
An appointment for a meeting.
\[ \text{date} : \text{Date} \] // the time
\[ \text{room} : \text{Integer} \] // room number
\[ \text{with} : \text{Set}\langle \text{Person} \rangle \] // appt with

Set of abstract operations:
e.g., addAttendee, etc. with PoS style specs referencing abstract specification fields.

Why ADTs?
- Bridges gap between domain concept and implementation
- Formalizes domain concept, provides basis for reasoning about correctness of the implementation
- Shields client from implementation. Implementation can vary without affecting client!

An ADT Is a Set of Operations
- Operations operate on data
- ADT abstracts from organization to meaning of data
- ADT abstracts from structure to use
- Data representation (implementation) doesn’t matter!

Are These Types Same or Different?
- They are different!
- They are the same! Both implement the concept of the 2-d point. Goal of ADT methodology is to express sameness.
  - Clients depend only on the set of operations: x(), y(), r(), theta(), etc.
  - Data representation can be changed: to change algorithms, to delay decisions, to fix bugs

Specification fields (Abstract fields).

class Poly {
    // rep. invariant: d = coeffs.length-1
    private int d; // degree of the polynomial
    private int[] coeffs; // coefficients
    // concrete operations add, sub, mul, eval, in terms of rep. fields coeffs, d.
}

class Poly {
    // rep. invariant: ...
    private List<Term> terms; // terms of poly
    // operations add, sub, mul, eval, etc. in terms of rep. field terms.
}

class RightTriangle {
    float base, altitude;
}

class RightTriangle {
    float base, hypot, angle;
}

class Point {
    float x;
    float y;
}

class Point {
    float r;
    float theta;
}
Are These Types Same or Different?

Clients depend only on the set of operations: `add(Poly)`, `mul(Poly)`, etc.

```
class Poly {
    private int d;
    private int[] coeffs;
}
class Poly {
    private List<Term> terms;
}
```

Abstraction Barrier

Clients access the ADT through its operations. They never access the data representation.

```
Abstraction Barrier: 
  x()  
  y()  
  theta() 
  Point() etc.
```

2-d Point as an ADT

```
// A 2-d point in the plane
public float x();
public float y();
public float r();
public float theta();

// can be created
public Point(); (0,0)
public Point(float x, float y);
public Point centroid(Set<Point> points);
```

Specifying an ADT

```
immutable    mutable
class TypeName class TypeName
1. overview    1. overview
2. specification fields 2. specification fields
3. creators    3. creators
4. observers    4. observers
5. producers    5. producers (rare!)
6. mutators     6. mutators
```

Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
- Specifying an ADT
  - immutable
  - mutable
- The ADT design methodology
- Next: reasoning about ADT implementations
Poly, an immutable datatype:

**overview**

```java
/**
 * A Poly is an immutable polynomial with integer coefficients. A Poly is:
 *     c_0 + c_1 x + c_2 x^2 + ...
 */
class Poly {
    // Abstract state (specification fields).
    // More on this later.

    // Overview: Always state whether mutable or immutable
    Define abstract model for use in specification of operations. In ADTs state is abstract, not concrete (i.e., this are NOT actual, implementation fields of Poly, just what we call specification fields.)

    // creators
    public Poly() {
        // modifies: none
        // effect: makes a new Poly = 0
    }

    public Poly(int c, int n) {
        // modifies: none
        // effect: makes a new Poly = cx^n
        // throws: NegExponentException if n < 0
    }

    // creators: This is example of overloading, two Poly constructors with different signatures.
    // New object is part of effects not preexisting state. Hence, modifies is none.

    // observers
    public int degree() {
        // returns: degree of this polynomial
    }

    public int coeff(int d) {
        // returns: the coefficient of the term of this polynomial, whose power is d
    }

    // observers: Used to obtain information about this polynomial. Return values of other types. Never modify the abstract state!
    this: the current Poly object. Also called the receiver
    Poly x = new Poly(...)
    c = x.coeff(3);

    // producers
    public Poly add(Poly q) {
        // modifies: none
        // returns: a new Poly with value this + q
    }

    public Poly mul(Poly q) {
        // modifies: none
        // returns: a new Poly with value this*q
    }

    // producers: Operations on a type that create other objects of the same type. Common in immutable types. No side effects, i.e., cannot change abstract values of any existing object
    Poly x = new Poly(...)
    c = x.add(3); 
```

Fall 15 CSCI 2600, A Milanova (slide due to Michael Ernst)

---

IntSet, a mutable datatype:

**overview, creators and observers**

```java
/**
 * Overview: An IntSet is a mutable, unbounded set of integers. E.g.,
 *     { x_1, x_2, ... x_n }
 */
class IntSet {

    // effects: makes a new empty IntSet
    public IntSet() {
        // returns: true if x in this IntSet,
        // else false
        public boolean contains(int x) {
            // modifies: this
            // effects: this_post = this_pre U { x }
            public void add(int x) {
                // modifies: this
                // effects: this_post = this_pre - { x }
                public void remove(int x) {
                    Mutators: operations that modify receiver this. Rarely modify anything other than this. Must list this in modifies clause. Typically, mutators have no return value.
```
Exercise: String, an immutable datatype

- Overview?
- Creators?
  - String(), String(char[] value), String(String original), ...
- Observers?
  - charAt, compareTo, contains, endsWith, ...
- Producers?
  - concat, format, substring, ...
- Mutators?
  - None!

Exercise: The Stack datatype

```java
public Stack()

public boolean empty()

public E peek()

public int search(Object o)

public E push(E item)

public E pop()
```

ADTs and Java Language Features

- Both classes and interfaces are appropriate
- Java classes
  - Operations in the ADT are public
  - Other operations are private
  - Clients can only access ADT operations
- Java interfaces
  - Clients only see the ADT operations
  - Cannot include creators or fields

Outline

- Comparing specifications with logical formulas
- What is an abstract data type (ADT)?
  - Specifying an ADT
    - immutable
    - mutable
- The ADT design methodology
- Next: reasoning about ADT implementations