Condition Strength

“P is stronger than Q” means:

-P implies Q”

-P guarantees more than Q

-Fewer values satisfy P than Q

Stronger means more specific

Weaker means more general

Examples:

x>1 is stronger than x>0  
x > 0 is stronger than x>-1

y is even is stronger than y Ξ 2 (mod 4)

No relationship: 0 ≤ x ≤ 10 or 5 ≤ x ≤ 11

y = x + 1 && x is even stronger than y is odd && x is even

{ |x| > w } vs { x > w } (no relationship)

Let P => Q => R, S => T => U and { Q } code { T } be true

{ Q } code { T }

{ Q } code { U }

{ P } code { T }

{ P } code { U }

Are true

Loops:

Partial Correctness:

1. Guess loop invariant

2. Prove loop invariant holds before loop (base case)

3. Prove loop invariant holds after k+1th iteration (induction)

4. Prove exit condition and invariant imply postcondition

Termination:

1. Guess decrementing function

2. Prove it’s bounded

3. Prove it approaches bound

4. Prove lower bound and loop invariant imply exit condition

Specifications:

Contract between a method and its caller

Obligations of the method (implementation of specification):

agrees to provide postcondition if precondition held!

Obligations of the caller (user of specification):

agrees to meet the precondition and not expect more than postcondition promised

Specs are good cuz:

Documentation, Abstraction, Modularity, Enables reasoning about correctness, Enables reasoning about substitutability

PoS Specification Format:

requires: clause spells out constraints on client

modifies: lists objects (typically parameters) that may be modified by the method.

throws: lists possible exceptions

effects: describes final state of modified objects

returns: describes return value

Specification Style:

Bad to have side effects AND return

Point is to be helpful – balance formalness

Specification Strength:

“A is stronger than B” means:

For every implementation I

“I satisfies A” implies “I satisfies B”

The opposite is not necessarily true

For every client C

“C works with B” implies “C works with A”

The opposite is not necessarily true

A larger world of implementations satisfy the weaker spec B than the stronger spec A

(FALSE) If specification A is stronger than specification B, then any implementation that satisfies B satisfies A as well.

(TRUE) If PA => PB and QB => QA then spec B is stronger than spec A. (PA denotes the precondition of A and QA denotes the postcondition of A.)

To Strengthen a spec:

Require less of client – fewer conditions. Promise more – effects, modifies returns

To weaken a spec:

Require more – add conditions

Promise less – effects, modifies, returns

Principle of substitutability:

A stronger spec can always be substituted for a weaker one

If spec of Y.foo(int) is stronger than that of X.foo(int) then

we can safely substitute Y.foo(int) for X.foo(int)!

ADTs Overview:

-Higher level data abstraction

-A specification mechanism

-A way of thinking about programs and design

-A set of operations:

ADT abstracts from organization to meaning of data

ADT abstracts from structure to use

Mutability:

Overview, Abstract Fields, Creators,

Observers, Producers, Mutators

Immutable ADTs have no mutators

Rare for immutables to have producers

Representation Stuff

Representation Invariant: Function that ensures

Representation is valid

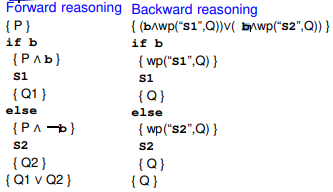
a) (FALSE) If every method in a Java class returns only immutable objects then the class is immutable.

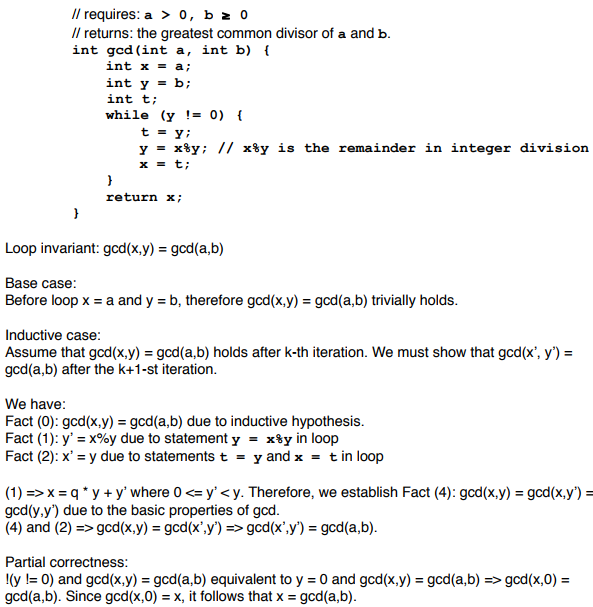
b) (FALSE) The representation invariant is part of the ADT specification.

c) (FALSE) The abstraction function is part of the ADT specification.

d) (FALSE) The rep invariant must hold before and after every statement in every method.

e) (TRUE) The abstraction function maps valid objects to abstract values



Spec A: requires: x ≥ 0 returns: y such that |y\*y – x| < 0.001

Spec B: requires: x ≥ 0 returns: y such that |y\*y – x| < 0.0001

Spec C: returns: y such that |y\*y – x| < 0.001 if x ≥ 0, and 0.0 if x < 0

Spec D: returns: y such that |y\*y – x| < 0.001 if x ≥ 0 throws: IllegalArgumentException if x < 0

a) A **B** Neither

b) A **C** Neither

c) A **D** Neither

d) B C **Neither**

e) B D **Neither**

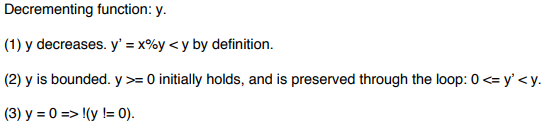
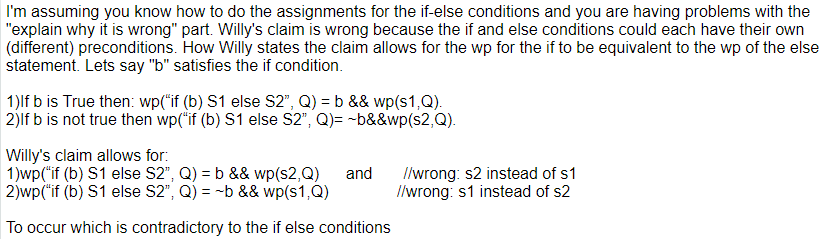
f) C D **Neither**

requires: none

modifies: this[index]

effects: thispost[index] = element returns: thispre[index]

throws: IndexOutOfBoundsException if index < 0 || index >= this.size()

true => ( if index < 0 || index ≥ this.size then throws IndexOutOfBoundsException else thispost[index] = element and returns thispre[index] ) AND (foreach i ≠ index, thispost[i] = thispre[i])

Overloading :(same function name but different signature)

1. Two or more methods having the same name with different arugment in same class is known as Overloading. 2. Overloading is used when you want to extend the functionlity. 3. Overloading is known as compile time polymorphism

Overriding :(same function name but same signature)

1. Two or more methods having the same method name and same arugment in parent class and child class in known as overriding. 2. Overriding is used when you want to reuse the existing functionlity. 3. Overriding is known as run time polymorphism