4 Assertions

4.1 Introduction

At each point in a program, there are usually constraints on the computational state that must hold for the program to be correct. For example, if a certain variable is supposed to record a count of how many changes have been made to a file, this variable should never be negative. It helps human readers to know about these constraints. Furthermore, if a program checks these constraints as it executes, it may find errors almost as soon as they occur. For both these reasons, it is advisable to record constraints about program state in assertions.

Assertion: A statement that must be true at a designated point in a program.

4.2 Types of Assertions

There are three sorts of assertions that are particularly useful:

Preconditions—A **precondition** is an assertion that must be true at the initiation of an operation. For example, a square root operation cannot accept a negative argument, so a precondition of this operation is that its argument be non-negative. Preconditions most often specify restrictions on parameters, but they may also specify that other conditions have been established, such as a file having been created or a device having been initialized. Often an operation has no preconditions, meaning that it can be executed under any circumstances.

Post conditions—A **post condition** is an assertion that must be true at the completion of an operation. For example, a post condition of the square root operation is that its result, when squared, is within a small amount of its argument. Post conditions usually specify relationships between the arguments and the results, or restrictions on the results. Sometimes they specify that the arguments do not change, or that they change in certain ways. Finally, a post condition may specify what happens when a precondition is violated (for example, that an exception will be thrown).

Class invariants—A **class invariant** is an assertion that must be true of any class instance before and after calls of its exported operations. Usually class invariants specify properties of attributes and relationships between the attributes in a class. For example, suppose a `Bin` class models containers of discrete items, like apples or nails. The `Bin` class might have `currentSize`, `spaceLeft`, and `capacity` attributes. One of its class invariants is that `currentSize` and `spaceLeft` must always be between zero and `capacity`; another is that `currentSize` + `spaceLeft` = `capacity`.

A class invariant may not be true during execution of a public operation, but it must be true between executions of public operations. For example, an operation to add something to a container must increase the size and decrease the space left attributes, and for a moment during execution of this operation their sum might not be correct, but when the operation is done, their sum must be the capacity of the container.

Other sorts of assertions may be used in various circumstances. An **unreachable code assertion** is an assertion that is placed at a point in a program that should not be executed under any circumstances. For example, the cases in a switch statement often exhaust the possible values of the switch expression, so execution should never reach the default case. An unreachable code assertion can be placed at the default case; if it is every executed, then the program is in an erroneous state. A **loop invariant** is an assertion that must be true at the start of a loop on each of its iterations. Loop invariants are used to prove program correctness. They can also help programmers write loops correctly, and understand loops that someone else has written.

4.3 Assertions and Abstract Data Types

Although we have defined assertions in terms of programs, the notion can be extended to abstract data types (which are mathematical entities). An **ADT assertion** is a statement that must always be true of the carrier set values or the operations in the ADT. ADT assertions can describe many things about an ADT, but usually they help describe the operations of the ADT. Especially helpful in this regard are operation *preconditions*, which usually constrain the parameters of operations, operation *post conditions*, which define the results of the operations, and *axioms*, which make statements about the properties of operations, often showing how operations are related to one another. For examples, consider the Natural ADT whose carrier set is the set of non-negative integers and whose operations are the usual arithmetic operations. A precondition of the mod (%) operation is that the modulus not be zero; if it is zero, the result of the operation is undefined. A post condition of the mod operation is that its result is between zero and the modulus less one. An axiom of this ADT is that for all natural numbers a, b, and $m > 0$,

$$(a+b) \% m = ((a \% m) + b) \% m$$

This axiom shows how the addition and mod operations are related.

We will often use ADT assertions, and especially preconditions, in specifying ADTs. Usually, ADT assertions translate into assertions about the data types that implement the ADTs, which helps insure that our ADT implementations are correct.

4.4 Using Assertions

When writing code, programmer should state pre- and subtle post conditions for public operations, state class invariants, and insert unreachable code assertions and loop invariants wherever appropriate.

Some languages have facilities to directly support assertions and some do not. If a language does not directly support assertions, then the programmer can mimic their effect. For example, the first statements in a class method can test the preconditions of the method and throw an exception if they are violated. Post conditions can be checked in a similar manner. Class invariants are more awkward to check because code for them must be inserted at the start and end of every exported method. For this reasons, it is often not practical to do this. Unreachable code assertions occur relatively infrequently and they are easy to insert, so they should always be used. Loop invariants are mainly for documenting and proving code, so they can be stated in comments at the tops of loops.

Often efficiency issues arise. For example, the precondition of a binary search is that the array searched is sorted, but checking this precondition is so expensive that one would be better of using a sequential search. Similar problems often occur with post conditions. Hence many assertions are stated in comments and are not checked in code, or are checked during development and then removed or disabled when the code is compiled for release.

Languages that support assertions often provide different levels of support. For example, Java has an `assert` statement that takes a `boolean` argument; it throws an exception if the argument is not true. Assertion checking can be turned off with a compiler switch. Programmers can use the `assert` statement to write checks for pre- and post conditions, class invariants, and unreachable code, but this is all up to the programmer.

The languages Eiffel and D provide constructs in the language for invariants and pre- and post conditions that are compiled into the code and are propagated down the inheritance hierarchy. Thus Eiffel and D make it easy to incorporate these checks into programs. A compiler switch can disable assertion checking for released programs.

4.5 Assertions in Ruby

Ruby provides no support for assertions whatever. Furthermore, because it is weakly typed, Ruby does not even enforce rudimentary type checking on operation parameters and return values, which amount to failing to check type pre- and post conditions. This puts the burden of assertion checking firmly on the Ruby programmer.

Because it is so burdensome, it is not reasonable to expect programmers to perform type checking on operation parameters and return values. Programmers can easily document other pre- and post conditions and class invariants, however, and insert code to check most value preconditions, and some post conditions and class invariants. Checks can also be inserted easily to check that supposedly unreachable code is never executed. Assertion checks can raise appropriate exceptions when they fail, thus halting erroneous programs.

For example, suppose that the function `f(x)` must have a non-zero argument and return a positive value. We can document these pre- and post conditions in comments and incorporate a check of the precondition in this function's definition, as shown below.

```
# Explanation of what f(x) computes
# @pre: x != 0
# @post: @result > 0
def f(x)
   raise ArgumentError if x == 0
   …
end
```

Figure 1: Checking a Precondition in Ruby

Ruby has many predefined exceptions classes (such as `ArgumentError`) and new ones can be created easily by sub-classing `StandardError`, so it is easy to raise appropriate exceptions.

We will use assertions frequently when discussing ADTs and the data structures and algorithms that implement them, and we will put checks in Ruby code to do assertion checking. We will also state pre- and post conditions and class invariants in comments using the following symbols.

Use	Symbol
Mark a precondition	`@pre:`
Mark a post condition	`@post:`
Mark a class invariant	`@inv:`
Specify a return value	`@result`
Specify the value of `@attr` after an operation executes	`@attr`
Specify the value of `@attr` before an operation executes	`old.@attr`

4.6 Review Questions

1. Name and define in your own words three kinds of assertions.
2. What is an axiom?
3. How can programmers check preconditions of an operation in a language that does not support assertions?
4. Should a program attempt to catch assertion exceptions?
5. Can assertion checking be turned off easily in Ruby programs as it can in Eiffel or D programs?

4.7 Exercises

1. Consider the Integer ADT with the set of operations { +, -, *, /, %, == }. Write preconditions for those operations that need them, post conditions for all operations, and at least four axioms.
2. Consider the Real ADT with the set of operations { +, -, *, /, $\sqrt[n]{x}$, x^n }, where x is a real number and n is an integer. Write preconditions for those operations that need them, post conditions for all operations, and at least four axioms.

   Consider the following fragment of a class declaration in Ruby.

```
NUM_LOCKERS = 138

class Storage

  # Set up the locker room data structures
  def initialize
    @lockerIsRented = new Array(NUM_LOCKERS, false)
    @numLockersAvailable = NUM_LOCKERS
  end

  # Find an empty locker, mark it rented, return its number
  def rentLocker
    …
  end

  # Mark a locker as no longer rented
  def releaseLocker(lockerNumber)
    …
  end

  # Say whether a locker is for rent
  def isFree(lockerNumber)
    …
  end

  # Say whether any lockers are left to rent
  def isFull?
    …
  end

end
```

This class keeps track of the lockers in a storage facility at an airport. Lockers have numbers that range from 0 to 137. The Boolean array keeps track of whether a locker is rented. Use this class for the following exercises.

3. Write a class invariant comment for the `Storage` class.
4. Write precondition comments and Ruby code for all the operations that need them in the `Storage` class. The precondition code may use other operations in the class.
5. Write post condition comments for all operations that need them in the `Storage` class. The post condition comments may use other operations in the class.
6. Implement the operations in the `Storage` class in Ruby.

4.8 Review Question Answers

1. A precondition is an assertion that must be true when an operation begins to execute. A post condition is an assertion that must be true when an operation completes execution. A class invariant in an assertion that must be true between executions of the operations that a class makes available to clients. An unreachable code assertion is an assertion stating that execution should never reach the place where it occurs. A loop invariant is an assertion true whenever execution reaches the top of the loop where it occurs.
2. An axiom is a statement about the operations of an abstract data type that must always be true. For example, in the Integer ADT, it must be true that for all Integers n, $n * 1 = n$, and $n + 0 = n$, in other words, that 1 is the multiplicative identity and 0 is the additive identity in this ADT.
3. Preconditions can be checked in a language that does not support assertions by using conditionals to check the preconditions, and then throwing an exception, returning an error code, calling a special error or exception operation to deal with the precondition violation, or halting execution of the program when preconditions are not met.
4. Programs should not attempt to catch assertion exceptions because they indicate errors in the design and implementation of the program, so it is better that the program fail than that it continue to execute and produce possibly incorrect results.
5. Assertion checking cannot be turned off easily in Ruby programs because it is completely under the control of the programmer. Assertion checking can be turned of easily in Eiffel and D because assertions are part of the programming language, and so the compiler knows what they are. In Ruby, assertions are not supported, so all checking is (as far as the compiler is concerned) merely code.