Reading

1. Sections 9.4.1 and 9.4.3

Problems

1. Template expansion versus erasure

The Java and C# language designs have related forms of generics that can be compiled separately and can be type-checked at compile time. However, Java and C# implement generics in two different ways. Java generics are implemented using an “erasure” technique discussed in class that inserts casts and uses one run-time copy for all instances of the generic. The C# implementation of generics may produce separate copies if several instances of a generic are used in a program.

(a) Java chose the erasure and cast implementation of generics for several reasons. One was the large number of users running old Java VM's and a huge amount of legacy code. Why does the type erasure and casts implementation support both legacy code and old VM's better than an implementation that uses expansion.

(b) At the Big Game, a Cal student says, “I think C# made the wrong choice when adding generics to their language: they've invalidated all their old code and VM's. Java does it right.” The Stanford student coughs and says, “But what if your old code starts adding Strings onto your Vector<int>. This is wrong, but the error is not detected until you use the inserted String. It should be detected when the old code inserts a String into a Vector<int>.” The Cal student replies “I have a solution to this problem. I just use this special MyVector written below. It will catch errors at run time right when a program tries to add a String into a Vector<int>, instead of waiting until the String is used as an Integer.”

```java
class MyVector<T> extends Vector {
    void add (Object a) {
        try {
            T tmp = (T)a;
            super.add(tmp);
        } catch (ClassCastException c) {
            System.out.println("Error Detected");
        }
    }
}
```
The Stanford student rolls his eyes and tells the Cal student that his pathetic scheme would never detect those insertion errors when they happen, even at runtime. Who is right? Explain why. (Hint: Include an explanation of what happens to T for the add function used in a MyVector<Integer> after Java erasure/casting.)

(c) Translate the following code into the code that could run on an older VM without generics (i.e. Java 1.4 or earlier). Remember that ints are value types that are not subtypes of Object.

```java
MyVector<Integer> a = new MyVector<int>();
a.add(5);
int j = a.lastElement();
```

Fill in the following code:

```java
MyVector a = new MyVector();
a.add ___________________________ 5 ___________
int j = __________________________ a.lastElement() __________________ ;
```

(d) Sometimes Java’s use of erasure and casting has very bizarre effects. In this case, write the console output of the following program.

```java
class Container<T> {
    public T internal;
    public static Container lastInstance;
    Container(T value) {
        internal = value;
        lastInstance = this;
    }
}

class Main {
    public static void main(String args[]) {
        Container<String> str = new Container<String>("Happy");
        Container<Integer> myint = new Container<Integer>(31337);
        System.out.println(str.lastInstance.internal.toString());
    }
}
```

Explain why it behaves this way in terms of generics and Java erasure.

(e) C++ uses the expansion method for its templates instead of Erasure and inserted casts. What does the output of the same program translated into C++ look like. Write the output and explain why it is different or why it remains similar even with a different template expansion mechanism.

(f) It is sometimes useful to have a class that inherits from a template parameter like so:

```java
class MyClass <T> extends T {
    ...
}
```

Why is this construct ineffective when used with erasure and casting?

2. .......................................................... Templates and Generics

(a) Consider a C++ class of this form:
template <class T>
class C<T> {
    ...
    int f (T *x, T *y) { ... x->less(y) ... }
    ...
};

Suppose a C++ program contains a an object of type C<A> for some C++ class A. Explain what functions, operators, and so on, class A must define in order for the code shown to compile and link correctly.

(b) Suppose that an analogous class C is written in Java.

    class C<T> {
        ...
        int f (T x, T y) { ... x.compareTo(y) ... }
        ...
    }

Explain why the Java 1.5 compiler would not accept the code, based on the fragments shown.

(c) The Java code can be written so that it will compile-time type check. Instead of class C<T>, we will need to write something of the form

    class C<T extends ________________> {
        ...
        int f (T x, T y) { ... x.compareTo(y) ... }
        ...
    }

What would you write in the underlined region to make this correct Java? Explain, and describe any additional classes or interfaces you may need.

3. "Covariance, Java Arrays and Generics"

In Java, array types are covariant but generics are not. This problem asks you to explain one resulting limitation of what the Java compiler accepts. Assume that Integer is a subtype of Number, class List<T> is a Java generic class and, for every T, class type ArrayList<T> is a subtype of List<T>. An ArrayList is like a Vector, without some associated synchronization properties.

(a) Using the term “covariant” or “contravariant,” as appropriate, explain why the second line in the following code is rejected by the Java type checker.

    List<Integer> li = new ArrayList<Integer>();
    List<Number> ln = li; // illegal
    ln.add(new Float(3.1415));

(b) The following code illustrates why the designers of Java decided to prohibit code that instantiates an array of a generic type.

    The expression new List<String>[10] is not allowed. However, assume for purpose of discussion that all of the lines below compile and execute. If x is a List<String>, then x.get(0) is a String.

    1: List<String>[] lsa = new List<String>[10]; // illegal in real Java
    2: List<Integer> li = new ArrayList<Integer>();
    3: li.add(new Integer(3));
    4: Object[] oa = lsa; // OK because List<String> is a subtype of Object
    5: oa[0] = li;
    6: String s = lsa[0].get(0);
Explain why lines 5 and 6 of the code above type-check at compile time, given the declarations in the previous lines. What goes wrong at run-time when the last line is executed?

(c) Java does allow one way to instantiate an array of a generic type - when the type argument is an unbounded wildcard. Explain concisely why the following legal Java code does not lead to the same problems as line 1 above. Your answer should not require the whole page.

List<?>[] ll = new List<?>[10];

4. ........................................................................................................... Fairness

The guarded-command looping construct

\[
\begin{align*}
\text{do} & \\
\text{Condition} & \Rightarrow \text{Command} \\
... & \\
\text{Condition} & \Rightarrow \text{Command} \\
\text{od}
\end{align*}
\]

involves nondeterministic choice, as explained in the text. An important theoretical concept related to potentially nonterminating nondeterministic computation is fairness. If a loop repeats indefinitely, then fair nondeterministic choice must eventually select each command whose guard is true. For example, in the loop

\[
\begin{align*}
x & := 0 \\
\text{do} & \\
\text{true} & \Rightarrow x := x+1 \\
\text{true} & \Rightarrow x := x-1 \\
\text{od}
\end{align*}
\]

both commands have guards that are always true. It would be unfair to execute \(x := x+1\) repeatedly without ever executing \(x := x-1\). Most language implementations are designed to provide fairness, usually by providing a bounded form. For example, if there are \(n\) guarded commands, then the implementation may guarantee that each enabled command will be executed at least once in every \(2^n\) or \(3^n\) times through the loop. The number \(2^n\) or \(3^n\) is of course implementation-dependent.

(a) Suppose that an integer variable \(x\) can only contain an integer value with absolute value less than INTMAX. Will the do-od loop above cause overflow or underflow (or could it cause either) under a fair implementation that does not have a specific bound but just requires that every enabled action be executed eventually? What about an implementation that is not fair?

(b) What property of the following loop is true under a fair implementation but false under an unfair implementation?

\[
\begin{align*}
go & := \text{true;} \\
n & := 0; \\
\text{do} & \\
go & \Rightarrow n := n+1 \\
go & \Rightarrow go := \text{false} \\
\text{od}
\end{align*}
\]

(c) Is fairness easier to provide on a single-processor language implementation or a multiprocessor? Discuss briefly.
5. Message Passing

There are eight message-passing combinations involving synchronization, buffering, and message order, as shown in the following table.

<table>
<thead>
<tr>
<th>Synchronous</th>
<th>Asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffered</td>
<td>Ordered</td>
</tr>
<tr>
<td>Unbuffered</td>
<td>Unordered</td>
</tr>
</tbody>
</table>

Fill in the table with a programming language/protocol that uses this combination and explain why the combination makes some sense, or explain why you think the combination is either meaningless or too weak for useful concurrent programming.

You should fill in at minimum the following languages/protocols: CML, Java, TCP, UDP.

6. Java ConcurrentHashMap

ConcurrentHashMap allows concurrent access to a hash table with minimal locking. Without going into all of the details, this problem asks you to think about some aspects of the design. The relatively tricky design of ConcurrentHashMap is intended to allow read access to parts of the data structure that may be written concurrently, in a way that causes the read to try again if the state seems inconsistent or incomplete.

The hash table implementation uses a resizable array of hash buckets, each consisting of a linked list of Map.Entry elements. As with other hash table implementations you may be familiar with, the hashcode of an entry determines its bucket; when several entries hash to the same bucket, they are placed in a linked list. Here is part of the definition of a Map.Entry class.

```java
protected static class Entry implements Map.Entry {
    protected final Object key;
    protected volatile Object value;
    protected final int hash;
    protected final Entry next;
    ...
}
```

The volatile modifier asks the Java Virtual Machine to order accesses to the shared copy of the variable so that its most current value is always read.

Instead of a single lock governing access to the entire collection, ConcurrentHashMap uses a lock over each segment of buckets. The linked list used by ConcurrentHashMap is designed so that the implementation can detect that its view of the list is inconsistent or stale. If it detects that its view is inconsistent or stale, or simply does not find the entry it is looking for, it then synchronizes on the appropriate bucket lock and searches the chain again.

(a) What does the use of `final` in the `Map.Entry` class tell you about the way a linked list of `Map.Entry` elements may change when a ConcurrentHashMap is updated? (Don’t think too deeply - the purpose of this question is to point out something that is important for later questions.)

(b) There is one straightforward way to remove an item from a linked list of `Map.Entry` objects. Describe the steps involved in removing the second item from such a list. To provide clarity, write a Haskell statement that, given a list L, removes the second element. (Hint: Your statement should use only the functions `head`, `tail`, and the infix operator `:`)

(c) The ConcurrentHashMap retrieval operations first find the head pointer for the desired bucket. This is done without locking, so the value of the head pointer could be stale. The operation then traverses the linked list representing the bucket starting from the head pointer, without acquiring the lock for that bucket. If the operation does not find the value it is looking for, it acquires the lock for the bucket and tries again. Here is the code, in case you want to look at it; you may be able to answer the question without reading the code.
int hash = hash(key); // throws null pointer exception if key is null

// Try first without locking...
Entry[] tab = table;
int index = hash & (tab.length - 1);
Entry first = tab[index];
Entry e;

for (e = first; e != null; e = e.next) {
  if (e.hash == hash && eq(key, e.key)) {
    Object value = e.value;
    // null values means that the element has been removed
    if (value != null)
      return value;
    else
      break;
  }
}

// Recheck under synch if key apparently not there or interference
Segment seg = segments[hash & SEGMENT_MASK];
synchronized(seg) {
  tab = table;
  index = hash & (tab.length - 1);
  Entry newFirst = tab[index];
  if (e != null || first != newFirst) {
    for (e = newFirst; e != null; e = e.next) {
      if (e.hash == hash && eq(key, e.key))
        return e.value;
    }
  }
  return null;
}

Why do the second traversal? What advantage does this two-pass algorithm have over simply locking the linked list the first time and doing only one traversal?

(d) Removing an element from a ConcurrentHashMap poses several problems. First, because a thread could see stale values for the link pointers in a hash chain, simply removing an element from the chain would not be sufficient to ensure that other threads will not continue to see the removed value when performing a lookup. However, there’s a clue to how the implementation works in the get code above – the appropriate Entry object is found and its value field is set to null. Then the algorithm you may have discovered in part (b) is used. Explain why declaring the value field as volatile is useful here.

7. ................................................................. Java Concurrency

NPR's Car Talk radio show, in which Click and Clack, the Tappet brothers, give opinionated advice on all kinds of automotive ailments, wants to expand online. Their Computer Hardware Specialist, C. Colin Backslash, is trying to write a server-side Java program that can reply to user queries about car problems with auto-generated back and forth dialogue between the two show hosts. Early on in the project, Colin encounters a puzzling concurrency problem and asks for your help. His code exhibits unpredictable deadlocks:

```java
public class CarTalkGuy extends Thread {
  private String name;
  private CarTalkGuy brother;
  private static SummerIntern gladysOvernow = new SummerIntern();
```
When the program executes successfully, it produces output like the following:

Click: Clearly, your car needs a new radiator. Clack: And remember: don’t drive like my brother. Click: No, don’t drive like *my* brother!

Clack: Clearly, your car needs a right blinker bulb. Click: And remember: don’t drive like my brother. Clack: No, don’t drive like *my* brother!

Reminder: In Java, you can create a thread by extending the Thread class and overriding the Thread.run() method. A call to Thread.start() returns immediately in the calling thread and executes Thread.run() in a newly created thread.

(a) Locks protecting synchronized objects in Java are reentrant: threads that already own a particular lock are allowed to acquire the lock more than once. Which function calls, during a successful execution as given above, are only possible because of reentrant locks?

(b) Sometimes the given program deadlocks, especially for complicated cases when the call to researchAnswer() may take a while to return. Explain, in terms of threads, locks, and synchronized objects, why deadlock may occur here. Your answer should not make any assumptions about class SummerIntern.

(c) What will the program have output when the deadlock occurs?

(d) After some thought, Colin suggests taking some pressure off the SummerIntern by passing in a hint after calling researchAnswer() and presents the following code, in which the SummerIntern waits until the hint is given:
To provide the hint, class CarTalkGuy gets an extra synchronized method:

```java
public synchronized void provideHint() {
    gladysOvernow.provideHint("look under the hood");
}
```

Also, the last two lines of main are changed so that click will call researchAnswer() in one thread and also provide the hint in another thread:

```java
    click.start();
    click.provideHint();
```

This change avoids the previous deadlock problem, since Clack's thread is never started. To Colin's surprise though, the new program sometimes deadlocks without printing anything. In which lines of code do Click's threads get stuck now during a deadlock?

(e) Explain in one paragraph why the threads are deadlocked.