Reading

1. Chapter 10, section 10.2.3
2. Chapter 11, sections 11.3.2, and 11.7
3. Chapter 12, section 12.4
4. Chapter 13, section 13.3, 13.4 and 13.5
5. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches (pdf on CourseWare), pages 1–5.

Problems

1. Protocol Conformance

   We can compare Smalltalk interfaces to classes using **protocols**, which are lists of operation names (selectors). When a selector allows parameters, as in `at:` `put:`, the selector name includes the colons but not the spaces. More specifically, if `dict` is an updatable collection object, such as a dictionary, then we could send `dict` a message by writing `dict at:‘cross’ put:‘angry’`. (This makes our dictionary definition of “cross” the single word “angry.”) The protocol for updatable collections will therefore contain the seven-character selector name `at:put:`. Here are some example protocols.

   ```
   stack:{isEmpty, push:, pop }
   queue:{isEmpty, insert:, remove }  
   priority_queue:{isEmpty, insert:, remove }  
   dequeue:{isEmpty, insert:,insertFront:, remove, removeLast }  
   simple_collection:{isEmpty }
   ```

   Briefly, a stack can be sent the message `isEmpty`, returning `true` if empty, `false` otherwise; `push:` requires an argument (the object to be pushed onto the stack), `pop` removes the top element from the stack and returns it. Queues work similarly, except they are first-in/first-out instead of first-in/last-out. Priority queues are first-in/minimum-out and dequeues are doubly-ended queues with the possibility of adding and removing from either end. The `simple_collection` class just collects methods that are common to all the other classes. We say that the protocol for `A` **conforms** to the protocol for `B` if the set of `A` selector names contains the set of `B` selector names.

   (a) Draw a diagram of these classes, ordered by protocol conformance. You should end up with a graph that looks like William Cook’s drawing shown in the text.

   (b) Describe briefly, in words, a way of implementing each class so that you only make `B` a subclass of `A` if the protocol for `B` conforms to the protocol set for `A`.

   (c) For some classes `A` and `B` that are unrelated in the graph, describe a strategy for implementing `A` as a subclass of `B` in a way that keeps them unrelated.

   (d) Describe implementation strategies for two classes `A` and `B` (from the set of classes above) so that `B` is a subclass of `A`, but `A` conforms to `B`, not the other way around.
C++ allows the programmer to specify how to convert between different classes. This can be very useful when you are passing around objects by value, as the default behavior is not always what is desired. If you are unfamiliar with C++, you may find Bruce Eckel's *Thinking in C++, 2nd Ed* helpful. It is freely available on the web, via http://www.mindview.net/Books/.

(a) What is printed by the following code and why? Your answer should be three sentences or less.

```cpp
#include <iostream>
using namespace std;

class A {
public:
  virtual void foo() { cout << "A::foo\n" << endl; }
};

class B : public A {
public:
  virtual void foo() { cout << "B::foo\n" << endl; }
};

void test( A a );

void test( A a ) {
  a.foo();
}

int main() {
  A a;
  B b;
  test(a);
  test(b);
}
```

(b) C++ provides an interesting mix of features for converting between objects of different types. The two primary language features are copy constructors and operator conversion. Like many parts of C++ they can surprise people. Explain why the following code prints

```
bork
bork
```

instead of

```
bork
bork
```

```cpp
or
bork
bark
```

could this problem be solved by using a copy constructor instead? Explain briefly.

```cpp
#include <iostream>
using namespace std;

class A {
public:
  char *s;
  virtual void foo() { cout << s << endl; }
  A() : s('bork') {}  
```
(C) C++ is the only mainstream object oriented language that does not hide its objects behind pointers or references. You have just seen how this can cause an interesting set of problems when programmers pass objects by value. What two character edit can you do to the original code to make it work as expected?

3. ................................................................. Function Subtyping

Assume that Square <: Rectangle and Rectangle <: Shape. Which of the following subtype relationships hold in principle?

i) (Rectangle → Rectangle) <: (Rectangle → Rectangle)
ii) (Rectangle → Square) <: (Rectangle → Rectangle)
iii) (Rectangle → Shape) <: (Rectangle → Rectangle)
iv) (Shape → Rectangle) <: (Rectangle → Rectangle)
v) (Square → Rectangle) <: (Rectangle → Rectangle)
vi) (Shape → Square) <: (Rectangle → Rectangle)
vii) (Square → Square) <: (Rectangle → Rectangle)
viii) (Shape → Shape) <: (Rectangle → Rectangle)
ix) (Square → Rectangle) → Shape <: (Rectangle → Square) → Shape
x) (Square → Rectangle) → Shape <: (Square → Rectangle) → Rectangle
xi) (Square → Square) → Square <: (Rectangle → Rectangle) → Rectangle
xii) (Square → Square) → Square <: (Rectangle → Square) → Rectangle

4. ................................................................. Contravariant Method Specialization

Having done well in CS 242, you have been asked to join the C++ language standardization committee. Despite your reservations about the committee process, they order Chicago Pizza at every meeting, so you accept. A UPenn student, who also did well in CS 242, has an idea for the next version of C++. He wants to add contravariant method specialization. In this question, we will examine the consequences of making this change to the semantics of C++. Let's say we have the following class hierarchy in C++:

class Shape {
private:
  double area;

public:
  Shape(double a) { area = a; }
(a) Without contravariant method specialization, the version of update in class B is an overload of the version of update inherited from class A.

```cpp
int main() {
    Circle c(7); // Circle radius is 7
    B b;
    A* a = &b;
    a->update(c);
    return 0;
}
```

In the code above, without contravariant method specialization, which version of update is called by `a->update(c)`? Which version is called with contravariant method specialization?

(b) If we added contravariant method specialization, should the compiler accept class B as a valid subclass of class A, or should the compiler report that class B is incompatible with its base class? Give a clear reason why or why not in terms of principles we have studied in the course. Hint: think about what method contravariance means.

(c) The rest of the committee likes the proposal to add contravariant method specialization, leaving the UPenn student feeling pretty smug. When the UPenn student starts discussing how he found the inspiration for his proposal, you sneak out to think about how this change will affect existing code. You try this test program:

```cpp
// The classes Shape and Circle are the same as before.
class A {
    public:
        virtual void update(Circle& c, double area) {
            double r = std::sqrt(area / PI);
            c.setRadius(r);
        }
};

class B : public A {
    public:
        virtual void update(Shape& s, double area) { s.setArea(area); }
};

int main() {
    Circle c(7); // Circle radius is 7
    B b;
    A* a = &b;
    a->update(c, PI);
}```
What does this program print under the current version of C++? What does this program print when you have contravariant method specialization?

(d) Given that there is a large amount of existing C++ code, what should you tell the committee about the consequences of adding contravariant method specialization to the language?

5. ................................................................................................. “like current” in Eiffel

Eiffel is a statically-typed object-oriented programming language designed by Bertrand Meyer and his collaborators. The language designers did not intend the language to have any type loopholes. However, there are some problems surrounding an Eiffel type expression called like current. When the words like current appear as a type in a method of some class, they mean, “the class that contains this method”. To give an example, the following classes were considered statically type-correct in the language Eiffel.

class Point
x : int
method equals (pt : like current) : bool
    return self.x == pt.x

class ColPoint inherits Point
    color : string
    method equals (cpt : like current) : bool
        return self.x == cpt.x and self.color == cpt.color

In Point, the expression like current means the type Point, while in ColPoint, like current means the type ColPoint. However, the type checker accepts the redefinition of method equals because the declared parameter type is like current in both cases. In other words, the declaration of equals in Point says that the argument of p.equals should be of the same type as p, and the declaration of equals in ColPoint says the same thing. Therefore, the types of equals are considered to match.

(a) Suppose Point has a move method that requires an integer argument, and returns a result of type like current. Assume that this method is inherited in subclass ColPoint. Write the type of move in ColPoint two ways, once using like current and once replacing like current with the type that this refers to (Point or ColPoint). Use any reasonable syntax for types that you find convenient; for instance you could write the type of a function mapping a bool/int pair to a bool as follows:

    bool * int -> bool

You may or may not need the * operator here.

(b) If ColPoint is a subtype of Point, is the type of move in ColPoint a subtype of the type of move in Point? Explain.

(c) Write the type of equals in ColPoint two ways, once using like current and once replacing like current with the type that this refers to (Point or ColPoint).

(d) If ColPoint is a subtype of Point, is the type of equals in ColPoint a subtype of the type of equals in Point? Explain.

(e) The designers of Eiffel wanted the subclass ColPoint to be a subtype of Point. Do you think they made some kind of mistake here? Explain.
6. Array Covariance in Java

Java array types are covariant with respect to the types of array elements (i.e., if \( B <: A \), then \( B[] <: A[] \)). This can be useful for creating functions that operate on many types of arrays. For example, the following function takes in an array and swaps the first two elements in the array.

```java
public swapper (Object[] swappee){
    if (swappee.length > 1){
        Object temp = swappee[0];
        swappee[0] = swappee[1];
        swappee[1] = temp;
    }
}
```

This function can be used to swap the first two elements of an array of objects of any type. The function works as it is and does not produce any type errors at compile-time or run-time.

(a) Suppose \( a \) is declared as \( \text{Shape[]} a \), an array of shapes, where \( \text{Shape} \) is some class. Explain why covariance (i.e., if \( B <: A \), then \( B[] <: A[] \)) allows the type checker to accept the call \( \text{swapper}(a) \) at compile time.

(b) Suppose \( \text{Shape[]} a \) as in part (a). Explain why the call \( \text{swapper}(a) \) and execution of the body of \( \text{swapper} \) will not cause a type error or exception at run time.

(c) Java uses run-time checks, as needed, to make sure that certain operations respect the Java type discipline at run time. What run-time type checks occur in the compiled code for the \( \text{swapper} \) function and where? List the line number(s) and the check that occurs on that line.

(d) A friend of yours is aghast at the design of Java array subtyping. In his brilliance, he suggests that Java arrays should follow contravariance instead of covariance (i.e., if \( B <: A \), then \( A[] <: B[] \)). He states that this would eliminate the need for run-time type checks. Write three lines of code or less that will compile fine under your friend's new type system, but will cause a run-time type error (assuming no run-time type tests accompany his rule). You may assume you have two classes, \( A \) and \( B \), that \( A \) is a subtype of \( A \), and that \( B \) contains a method, \( \text{foo} \), not found in \( A \). We give you two declarations that you can assume before your three lines of code.

```java
B b[];
A a[] = new A[10];
```

(e) Your friend, now discouraged about his first idea, decides that covariance in Java is alright after all. However, he thinks that he can get rid of the need for run-time type tests through sophisticated compile time analysis. Explain in a sentence or two why enforcing covariance solely at compile time, without runtime type-checking, will still lead to run-time type errors. You may write a few lines of code similar to those in part (d) if it helps you make your point clearly.

7. Java Bytecode Analysis

Java programs are compiled into bytecode, a simple machine language that runs on the Java Virtual Machine. Note that it is possible that bytecode could be written by hand or that compiled bytecode gets corrupted when transmitted over the network. So, when a class is loaded by the JVM, it is first examined by the bytecode verifier. The verifier performs static analysis of the class to ensure that a program will not cause an unchecked runtime type error.

One kind of bytecode error that the verifier should catch is use of an uninitialized variable. Here is a code fragment in Java and some corresponding bytecode. We have added comments to the bytecode to help you figure out the effects of the instructions.
The first line of the Java source allocates space for a new `Point` object and calls the `Point` constructor to initialize this object. The second line invokes a method on this object and therefore can be allowed only if the object has been initialized. It is easy to verify from this Java source that `p` is initialized before it is used.

Checking that objects are initialized before use in the bytecode is more difficult. The Java implementation creates objects in several steps: First, space is allocated for the object. Second, arguments to the constructor are evaluated and pushed onto the stack. Finally, the constructor is invoked. In the bytecode for this example, the memory for `p` is allocated in line 1, but the constructor isn’t invoked until line 4. If several objects are passed as arguments to the constructor, there could be an even longer code fragment between allocation and initialization of an object, possibly allocating multiple new objects, duplicating pointers, and taking conditional branches.

To account for pointer duplication, some form of aliasing analysis is needed in the bytecode verifier. This problem will consider a simplified form of initialize-before-use bytecode verification that keeps track of pointer aliasing by keeping track of the line number at which an object was first created. When a pointer to an uninitialized object is copied, the alias analysis algorithm copies the line number where the object was created, so that both pointers can be recognized as aliases for a single object. Of course, if an instruction creating a new object is inside a loop, then there may be many different uninitialized objects created at the same line number. However, the bytecode verifier does not need to work for this case, since Java compilers do not generate code like this. We won’t consider any cases with conditional branches in this problem.

Let’s examine the contents of stack and any associated line numbers for alias detection after execution of each of the bytecode instructions from above. Note that we draw the stack growing downwards in this diagram:

<table>
<thead>
<tr>
<th>After line</th>
<th>Stack</th>
<th>line # where created</th>
<th>initialized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Point p</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Point p</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>alias for p</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Point p</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>alias for p</td>
<td>1</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>int 3</td>
<td>3</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>Point p</td>
<td>1</td>
<td>yes</td>
</tr>
</tbody>
</table>

By using line numbers as object identifiers associated with each pointer on the stack, we keep track of the fact that both pointers on the stack point to the same place after line 2. So when the constructor invoked on line 4 initializes the alias for `p`, we recognize that `p` is initialized as well. Thus the `Print()` method is applied to a properly initialized `Point` object, and this example code fragment passes our simple initialize-before-use verifier.

(a) Consider the following bytecode:
1: new #1 <Class Point>
2: new #1 <Class Point>
3: dup
4: iconst 3
5: invokespecial #4 <Method Point(int)>
6: invokevirtual #5 <Method void Print()>

When line 5 is reached, there will be more than one uninitialized Point objects on the stack. Use the static verification method described above to figure out which one gets initialized, and whether the subsequent invocation of the Print() method occurs on an initialized Point. You should draw the state of the stack after each instruction, using the original line number associated with any pointer to detect aliases.

(b) So far we have only considered bytecode operations that use the operand stack. For this problem we introduce two more bytecode instructions, load and store, which allow values to be moved between the operand stack and local variables.

load x loads the value from local variable x and places it on the top of the stack.

In the following table, we have begun applying the initialize-before-use verification procedure described above to the code fragment in the left column. Note that we are maintaining information about aliasing and initialization state for local variable x as well as the contents of the operand stack. The contents of the stack correspond to that obtained just after the code in the left column is executed.

<table>
<thead>
<tr>
<th>Code</th>
<th>local variable x</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>line #</td>
<td>init?</td>
</tr>
<tr>
<td>1: new #1 &lt;Class Point&gt;</td>
<td>n/a</td>
<td>no</td>
</tr>
<tr>
<td>2: new #1 &lt;Class Point&gt;</td>
<td>n/a</td>
<td>no</td>
</tr>
<tr>
<td>3: store x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: load x</td>
<td>2</td>
<td>no</td>
</tr>
<tr>
<td>5: load x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6: iconst 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7: invokespecial #4 &lt;Method Point(int)&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8: invokevirtual #5 &lt;Method void Print()&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9: invokevirtual #5 &lt;Method void Print()&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continue applying the procedure to fill in the missing information on lines 4-9. Based on the state information you have after instruction 7, is the Print() method on line 8 applied to a properly initialized object? What about the Print() method on line 9?