Homework 4: Objects

CSE 130: Programming Languages

Early deadline: July 25 23:59, Hard deadline: July 28 23:59

1 [12pts] Smalltalk Implementation Decisions

In class, we assumed that Smalltalk objects have (1) a pointer to the class and (2) a map of (instance variable name, instance variable value) pairs. Variable instances were then looked up by name in the map using the variable name as key. Though this is a reasonable representation of objects, it's not how Smalltalk actually represents objects. Instead, objects only have (1) a pointer to the class and (2) a set of instance variable values. Each class, in turn, contains a pointer to a *class template*, in addition to the method-dictionary pointer. This template stores the names of all the instance variables that belong to objects created by the class and information used to lookup the variables.

1. [4pts] Why are the names of instance variables stored in the template, instead of in the objects (next to the values for the instance variables)? (Hint: recall why the names of the methods are stored next to the method pointers in the method dictionary.)

   Answer:

   If the names of the instance variables were stored in each object, this would consume a lot of space. It is more space-efficient to simply store the data in each object in some standard order. For example, for Point objects, the x,y-coordinates can be stored in each object with x first and y second. If the names "x" and "y" are needed, they can be stored in the Point class.

   Also, by storing the instance variable names in the class, all compiled methods can expect instance variables to have the same position within all objects of a given class. So new methods that are compiled at a later time will use the same instance variable positions in the compiled code as in older compiled methods.

2. [4pts] Each class's method dictionary only stores the names of the methods explicitly written for that class; inherited methods are found by searching up the superclass pointers at run-time. One optimization approach is for each subclass to have all of the methods of its superclasses in its method dictionary as well as it's own methods. What are some of the advantages and disadvantages of this approach?

   Answer:

   If the method dictionary of a class contained all of the methods defined on the class (including those implemented in the class and those inherited from superclasses), then method lookup would be simpler. Specifically, when a method is called, only one method dictionary would need to be searched. An advantage of this would be that the time involved in method lookup would decrease. A disadvantage is that if a superclass is changed and recompiled, then all of the subclasses of that superclass might have to be recompiled to account for any changes in the list of inherited methods.

3. [4pts]The class template stores the names of all the instance variables, even those inherited from parent classes. These are used to compile the methods of the class. Specifically, when a subclass is added to a Smalltalk system, the methods of the new class are compiled so that when an instance variable is accessed, the method can access this directly without searching through the method dictionary and without searching through superclasses. Can you see some advantages and disadvantages of this implementation decision, in comparison with looking up the relative position of an instance variable in

the appropriate class template each time the variable is accessed? Keep in mind that in Smalltalk, a set of classes could remain running for days, while new classes are added incrementally and, conceivably, existing classes could be rewritten and recompiled.

Answer:

An advantage is that when a method is compiled, the template can be used to determine the address of each instance variable, relative to the starting address of the object. As a result, methods can locate instance variables without searching any dictionary-like structure. A disadvantage is that when a superclass is changed, all of its subclasses may need to be recompiled to account for any changes in the list of instance variables of objects of the subclass. In the extreme case that an instance variable is added to the `Object` class, every other class in the system may need to be recompiled. (This provides a convenient way of stress-testing a Smalltalk compiler.)

2 [30pts] Contravariant Method Specialization

Having done well in CSE 130, 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 programming languages, has an idea for the next version of C++. She wants to add contravariant method specialization. In this question, we will examine the consequences of making this change to the semantics of C++. In class we discussed function subtyping, but not how it pertains to method specialization. The lecture slides and textbook, however, contain more information about this. (Since you already understand function subtyping, you may be able to figure this out without additional aid! But since this question is not trivial, feel free to discuss this with your peers.)

Suppose we have the following class hierarchy in C++:

```
class Shape {
private:
  double area;

public:
  Shape(double a) { area = a; }
  double getArea() { return area; }
  void setArea(double a) { area = a; }
};

class Circle : public Shape {
private:
  double radius;

public:
  Circle(double r) : Shape(PI*r*r) { radius = r; }
  double getRadius() { return radius; }
  void setRadius(double r) { setArea(PI*r*r); radius = r; }
};

class A {
public:
  virtual void update(Circle& c) { ... }
};

class B : public A {
public:
  virtual void update(Shape& s) { ... }
};
```

1. [7pts] Without contravariant method specialization, the version of `update` in `class B` is an overload of the version of `update` inherited from `class A`.

   ```
   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?

> **Answer:**
>
> Without specialization, `B::update` is an overload, so `A::update` is called. With specialization, `B::update` overrides `A::update`, so `B::update` is stored in the vtable and called instead.

2. [8pts] 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.

> **Answer:**
>
> By the contravariance rule, `Circle <: Shape => update(Shape) <: update(Circle)` i.e. functions are contravariant in their argument types. Thus, with contravariant specialization, `class B` is still a subtype of `class A`, so it should it should be able to be derived from `class A`.

3. [8pts] 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 she found the inspiration for his proposal, you sneak out to think about how this change will affect existing code. You try this test program:

```
// 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);

  std::cout << "Circle radius: " << c.getRadius() << std::endl;
  return 0;
}
```

What does this program print under the current version of C++? What does this program print when you have contravariant method specialization?

> **Answer:**
>
> Under the current version of C++, it prints 1 for the circle radius. With contravariant method specialization, it calls `B::update`, which results in a call to `setArea` on the circle. The function `setArea` never actually changes the radius, so the program prints 7, the original radius.

4. [7pts] 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?

Answer:

We can see that the results of the test program change under the new semantics. You should tell the committee that contravariant specialization will change the semantics of C++ in a way that could potentially break existing C++ programs in a serious manner.

3 [10pts] Function Subtyping

Suppose that function `f` has type `A` $\rightarrow$ `B` and function `g` has type `C` $\rightarrow$ `D`. What must be the relationship between types `A` and `C` and between types `B` and `D` to guarantee that type `C` $\rightarrow$ `D` is a subtype of type `A` $\rightarrow$ `B`?

Answer:

Using the definition of subtyping given in class, $C \rightarrow D <: A \rightarrow B$ if and only if any expression of type $A \rightarrow B$ can be replaced by an expression of type $C \rightarrow D$, and still produce a valid expression. Therefore, in any expression containing f, which has type $A \rightarrow B$, f can be replaced by g, which has type $C \rightarrow D$. Consider the expression $f(x)$. Here x must have type A, since it is the argument to f. If we replace f by g, we get $g(x)$. Since g's argument must be of type C, x has type A, and $g(x)$ must be a valid expression, it must be the case the A is a subtype of C. (That is, we can replace any C expression with an A expression.) Now consider the expression $f(x)$ in some larger expression. The type of $f(x)$ must be B since this is the return type of f. Replace f with g again, the expression $g(x)$ must have type D, since this is the return type of g. Since the larger expression must be valid when we replace f with g, and we have changed the type of the subexpression from B to D, it must be the case that D is a subtype of B.

In short, $C \rightarrow D <: A \rightarrow B$ if and only if $A <: C$ and $D <: B$.

Acknowledgements

Any acknowledgements, crediting external resources or people should be listed below.