CS 242. Autumn 1995. Handout 14
November 16, 1995

Homework 6

Due Tuesday, November 28

Reading

1. Read the excerpt (pages 122-134) from the book *Simula Begin*. The main idea you should try to understand is that a Simula class is just a procedure that returns a closure, instead of popping the activation record off the run-time stack. There is no information hiding in Simula, since all of the local data of the closure (object) is accessible by the "dot" notation. You may want to look at the ML functions representing the Simula class examples; these ML functions were handed out as the last page of Homework 5.

2. Read "The Smalltalk-80 System."

3. Read "Using types and inheritance in object-oriented languages" for its general points about program design and the use of subtyping and inheritance. (The paper is not very explicit about the distinction between subtyping (similarity of interface) and inheritance (similarity of implementation), since this distinction became more widely appreciated after the paper was written.) Note that the use of "abstract type" in this paper is different from the concept of "abstract data type" used in ML and CLU and discussed in the historical overview section of the Ingalls tape. In object-oriented programming, the phrase "abstract class" refers a class that exists for the purpose of organizing the class hierarchy. A characteristic feature of such classes is that it is not possible to create instances of them. An abstract class might have several methods. Some might be implemented in the abstract class if they will be used in the same way in all subclasses. If some methods will be implemented in different ways in each subclass, then these are typically not implemented in the abstract class. If some methods are not implemented, then it is not possible to create instances. This is discussed in the paper.

Problems

1. (10 points) Explain why the two implementations of a *point* abstract data type below are equivalent. More explicitly, explain why any program using the first would give exactly the same result (except possibly for differences in the speed of computation and any consequences of round-off error) as the other.

   You may appeal to the principle of datatype induction, or related ideas, from lecture. You need not give a detailed formal inductive proof; a simple informal argument referring to a principle discussed in class will be sufficient. You may explain what needs to be calculated, or what conditions need to be checked, without doing the calculations. The point of this problem is not to review trigonometry. An answer consisting of 3-5 sentences should suffice.

```
(* ----------------- Trig function for arctan(y/x) ---------------- *)

fun atan(x,y) =
              let val pi = 3.14159265358979323844
               in
                if (x > 0.0) then arctan(y / x)
                              else (if (y > 0.0) then (arctan(y / x) + pi)
                                                  else (arctan(y / x) - pi))
                handle Div => (if (y > 0.0) then pi/2.0 else ~pi/2.0)
              end;

(* -------------------- Two point abstract types ------------------ *)

abstype point = Pt of (real ref)*(real ref)   (* rectilinear coordinates *)
     with fun mk_Point(x,y) = Pt(ref x, ref y)
     and x_coord   (Pt(x, y)) = !x
     and y_coord   (Pt(x, y)) = !y
     and direction (Pt(x, y)) = atan(!x, !y)
     and distance  (Pt(x, y)) = sqrt(!x * !x + !y * !y)
     and move (Pt (x, y), dx, dy) = (x := !x + dx; y := !y + dy)
end;

abstype point = Pt of (real ref)* (real ref)  (* polar coordinates *)
with fun mk_Point(x,y) = Pt(ref (sqrt(x*x + y*y)), ref (atan(x,y)))
     and x_coord   (Pt(r, t)) = !r * cos(!t)
     and y_coord   (Pt(r, t)) = !r * sin(!t)
     and direction (Pt(r, t)) = !t
     and distance  (Pt(r, t)) = !r
     and move (Pt(r, t), dx, dy) =
            let val x = !r * cos(!t) + dx
                and y = !r * sin(!t) + dy
            in r := sqrt(x*x + y*y); t := atan(x,y)  end
end;
```

The types of the functions given in either declaration are listed in the following output from the ML compiler.

```
type point
val mk_Point = fn : real * real -> point
val x_coord = fn : point -> real
val y_coord = fn : point -> real
val direction = fn : point -> real
val distance = fn : point -> real
val move = fn : point * real * real -> unit
```

2. (10 points) Explain why the two functions below that return a form of *point* "objects", represented as ML closures of the same type, are equivalent. More explicitly, explain why any program using the first would give exactly the same result (except possibly for differences in the speed of computation and any consequences of round-off error) as the other.

   We did not cover any induction or equivalence principle for closures in class. You should try to devise you own, at least for this example, using your solution to problem 4 as a guide. Again, a simple informal argument with no detailed calculations will suffice.

```
(* -------------- Two point objects (as closures) --------------- *)

fun mk_point(xval,yval) =
   let val x = ref xval and y = ref yval
   in   {
         x_coord   = fn () => !x,
         y_coord   = fn () => !y,
         direction = fn () => atan(!x, !y),
         distance  = fn () => sqrt(!x * !x + !y * !y),
         move   = fn (dx, dy) => (x := !x + dx; y := !y + dy)
        }
    end;

fun mk_point(x,y) =
   let val r = ref (sqrt(x*x + y*y)) and t = ref (atan(x,y))
   in   {
         x_coord   = fn () => !r * cos(!t),
         y_coord   = fn () => !r * sin(!t),
         direction = fn () => !t,
         distance  = fn () => !r,
         move   = fn (dx, dy) =>
            let val x = !r * cos(!t) + dx
                and y = !r * sin(!t) + dy
            in r := sqrt(x*x + y*y); t := atan(x,y) end
        }
    end;
```

3. (15 points) A priority queue (also known as a heap) is a form of "queue" that allows the minimum element to be accessed. Assuming that Simula has a list type, a Simula, a *pq* class might look something like this:

```
class pq();
begin
   (int list) contents;
   bool procedure isempty();
       < ... return true if pq empty, else false ... >
   procedure insert(x); int x;
       < ... put x at appropriate place in list ... >
   int procedure deletemin();
```

```
      < ... delete first list elt and return it ... >

   contents := nil
end
```

where the sections `< ... >` would be filled in with appropriate executable Simula code. A priority queue object would be created by writing

```
h :- new pq();
```

and the components accessed by writing something like

```
h.contents;
h.isempty;
h.insert(x);
h.deletemin();
```

In ML we can write a function that produces a closure with essentially the same behavior as a Simula priority queue object. This is written below, together with the associated exception declaration.

```
exception Empty;

fun new_pq() =
    let val contents = (ref nil)  : int list ref
        fun insert(x,nil)   = [x]
        |   insert(x, y::l) = if x<y then x::(y::l)
                                      else y::insert(x,l)
    in
        {
         emp_meth = fn () => !contents = nil,
         ins_meth = fn x => (contents := insert(x,!contents)),
         del_meth = fn () => case !contents of
                                 nil => raise Empty
                               | y::l => (contents := l; y)
         }
    end;
```

This question asks you to think about a "loophole" in the Simula object system that allows a client program to interfere with the correct behavior of a priority queue object.

(a) The main property of a priority queue is that if n elements are inserted, and k are removed (for any $k < n$), then these k elements are returned in increasing order. Explain in two or three sentences why both the Simula and ML forms of priority queue objects should exhibit this behavior in a "reasonable" program.

(b) Explain in a few sentences how a "devious" program using a Simula priority queue object h could cause the following behavior: After inserting 0 and 1 into the priority queue by calls $h.insert(0)$ and $h.insert(1)$, the first *deletemin* operation on the priority queue returns some number other than 0. (This will not be possible for an ML priority queue object.) Do not use any pointer arithmetic or other operations that would not be allowed in a language like Pascal.

(c) Write a short sequence of priority queue operations of the form

```
h :- new pq();
h.insert(0);
h.insert(1);
...
h.deletemin();
```

where the ellipsis (...) may contain h operations, but not *insert* or *deletemin*, demonstrating the behavior you described in part b.

4. (15 points) This question asks you about subverting the intended behavior of an object, or class of objects, through subtyping. It is similar to the failure of encapsulation that was considered in problem 1. We will discuss this using an example class mechanism that could be added to ML.

   Our simple class mechanism will have class declarations and object creation operations of the form

```
class t(<args>)=
private
  <decls>
public
  <decls>
end;

val x = new t(<args>);
```

   where it is assumed that any initialization can be carried out within the "private" part of the class body. Essentially, these "Object-ML" declaration and create operations will have the same semantics as the following function declaration and call:

```
fun new_t(<args>)=
let
  <decls>
in
  {
   <decls>
  }
end;

val x = new_t(<args>);
```

   where the "private" declarations go before the *in* and "public" ones after. In addition, we will use the Simula-style "dot" notation for sending a message. For example, the priority queue closure described in problem 1 could be defined and manipulated as follows.

```
exception Empty;
class pq() =
    private
        val contents = (ref nil)  : int list ref
        fun insert(x,nil)   = [x]
        |   insert(x, y::l) = if x<y then x::(y::l)
                                     else y::insert(x,l)
    public
         empty()   = !contents = nil,
         insert(x) = (contents := insert(x,!contents)),
         delete()  = case !contents of
                         nil => raise Empty
                       | y::l => (contents := l; y)
    end;

val h = new_pq();
h.insert(2);
h.insert(0);
h.insert(1)
h.delete();
```

The value of the final expression, *h.delete*(), would be 0, with 1 and 2 remaining in the priority queue.

Aside from syntactic differences between this "class" construct and ML functions returning closures, the reasons for adding this construct to ML would be to permit subtyping and inheritance. One possibility would be to allow a subclass declaration of the form illustrated by the following example subclass of *pq*.

```
class balanced_pq() = extend pq() with
    private
        fun median(nil) = raise Empty
        |   median([x]) = x
        |   median(x::l) =  (* code omitted *)
    public
         middle()  = median(!contents)
    end;
```

A *balanced_pq* object will be like a *pq* object, but with one additional method (*middle*) returning the median element of the priority queue. Note that like Smalltalk but unlike C++, the public function of a subclass is allowed to access the private data of the superclass. In this example, this shows up in the implementation of *middle*, which accesses the private variable *contents*. (Those of you familiar with C++ might prefer to think of the *private* part of a class as *protected* rather than *private*.)

Question: The visibility rules are that any private data or function declared in a superclass may be accessed in any subclass. Explain how a "devious" subclass could add a method that causes priority queues to behave incorrectly, in the sense described in problem 1. Give approximately the same amount of detail as you did in your answer to problem 1.

5. (10 points) This question is about the relationship between subtyping and inheritance. The main principal associated with subtyping is substitutivity: if A is a subtype of B, then wherever a B object is required in a program, an A object may be used instead without producing a type error. For the purpose of this question, we will use *Message not understood* as an example of a type error. This is the most common error message resulting from a dynamic type failure in Smalltalk; it is the object-oriented analog of the error *Cannot take car of an atom* that every Lisp programmer has generated at some time.

   Smalltalk is a dynamically-typed language. This means that unlike Simula and C++, a class is simply a mechanism for creating objects. There is no type-checking based on the class of an object and the language makes no attempt to statically guarantee that subclass objects are substitutable for superclass objects. (This is different from Simula and C++.) This question asks you to show how substitutivity can fail, using the fact that a method given in a superclass can be redefined in a subclass. To avoid any confusion about Smalltalk syntax, we will just use the syntax described in problem 2.

   If there are no restrictions on how a method (member function) may be redefined in a subclass, then it is easy to redefine a method so that it requires a different number of arguments. This will make it impossible to meaningfully substitute a subclass object for a superclass object. A more subtle fact is that subtyping may fail when a method is redefined in a way that appears natural and (unless you've seen this before) unproblematic. This is illustrated using the following class and "subclass."

```
class point(xval:real,yval:real) =
    private
        val x = (ref xval) and y = (ref yval)
    public
         x_coord() = !x,
         y_coord() = !y,
         move(dx, dy) = (x := !x + dx; y := !y + dy),
         equal(pt) = (x = pt.x_coord()) andalso (y = pt.y_coord())
     end;

datatype color = red | green | blue;

class colored_point(xval:real,yval:real,cval:color) = extend point with
    private
        val c = (ref cval)
    public
         col() = !c,
         change_col(cval) = (c := cval),
         equal(cpt) = (x = cpt.x_coord()) andalso (y = cpt.y_coord())
                         andalso (c = cpt.col())
     end;
```

   The important part here is the redefinition of *equal* in the colored point class, which would not be allowed in Simula or C++, but would be allowed in Smalltalk. (The C++compiler doesn't consider this an error, but it wouldn't treat it as redefinition of a member function either.) The intuitive reason for redefining *equal* is that two colored points are equal only if they have the same coordinates and are the same color.

Problem: Give a single expression, of the form $object1.method(object2)$, that is guaranteed not to produce *Message not understood* if both objects are created as either points or colored points, but may produce this error if one is a point and the other a colored point. Explain briefly.