CSE 505, Fall 2003, Assignment 5
Due: 9 December 2003, 10:30AM (firm)

Note: You may be disappointed to learn that this assignment has no programming requirement. Fortunately, you can write O'Caml programs any time you like. You can even ask me for help.

1. (Fragile Superclasses) Assume a class-based OO language where "subclassing *is* subtyping" and there is no static overloading. Consider a class C, a class D that extends C, and a client P that uses classes C and D. Now consider each of these potential source-code changes to class C:

   (a) We add a method f to C.

   (b) We take an existing method f of C and change f from taking one argument of type T_1 to taking one argument of type T_2, where $T_1 \leq T_2$.

   (c) We take an existing method f of C and change f from taking one argument of type T_1 to taking one argument of type T_2, where $T_2 \leq T_1$.

   (d) We take an existing method f of C and make it abstract (removing its implementation, but still requiring all objects of type C to have it).

   For each of these changes:

   - Describe the conditions under which D will no longer typecheck. That is, describe all D where the definition of class D should type-check before the change of C but should not type-check after the change.
   - Describe the conditions under which P will no longer typecheck. That is, describe all P where the code for P should type-check before the change of C but should not type-check after the change.

2. (Self Type) Consider a class-based OO language where we add a new type `Self`. Suppose the only expressions with type `Self` are `self` (the special variable for the object) and a local (intra-method) variable `x` with explicit type `Self`. (These restrictions make `Self` basically useless.) When typechecking a method m in class C, we allow `Self` $\leq C$, but *not* $C \leq$ `Self`.

   (a) Is it sound to extend the type system to allow a method to have return type `Self`? Explain.
   Note: If D extends C, a method inherited from C with return type `Self` will still have return type `Self`, but in D we have `Self` $\leq D$.
   Note: If o has type T and class T has a method f with return type `Self`, then $o.f(\ldots)$ has type T, not `Self`.

   (b) Is it sound to extend the type system to allow a method to have arguments of type `Self`? Explain.
   Note: If D extends C, a method inherited from C with an argument of type `Self` will still have an argument of type `Self`, but in D we have `Self` $\leq D$.
   Note: If o has type T and class T has a method f with argument type `Self`, then the argument in a call to $o.f$ must have type T, not `Self`.

3. (Encoding Fields) Consider an OO language with public fields. (Whether the language has classes or not is irrelevant.)

   (a) Explain how we can translate programs in this language into a similar one where all fields are private. (Hint: Add two methods per field to every object. Explain how to change method bodies to use these fields.)

   (b) Explain how we can translate programs in the language with private fields into one with *no fields at all*, but with *method update*. Method update, written `o.m := mbody`, mutates an object `o` such that its method name `m` is bound to `mbody`. Here is a silly example: This method changes `o.m` to multiply its argument by n and "optimizes" the case $n == 2$:

```
unit f(C o, int n) {
   if(n==2)
     o.m := int m(int x) { x + x; }
   else
     o.m := int m(int x) { x * n; }
}
```

4. (Visitor Pattern) The code below is a simple class hierarchy for evaluating arithmetic expressions. Assume fields are public, dynamic-dispatch for the receiver, and static-dispatch for overloaded methods. Your job is to implement the "visitor pattern" for this code so that it is possible to implement new (unforeseen) operations over expressions without changing the existing classes.

   (a) Define an `ExpVisitor` class with all abstract methods such that:
   - The subclasses in the expression hierarchy all typecheck.
   - The `visit` methods in `IntExp` and `AddExp` invoke *different* methods in the `ExpVisitor` class. (Recall we have static-dispatch for overloaded methods.)

   (b) Define a concrete `HasZero` class that extends `ExpVisitor` and has a method `haszero` that takes an `Exp e` and returns `true` if and only if `e` has a 0 in it (i.e, an `IntExp` with `x` holding 0 somewhere in the expression tree). (Hints: `HasZero` will need private state. Method `haszero` should contain the call `e.visit(self)`.)

   (c) Why is it incorrect to define a `visit` method in `Exp` and have `IntExp` inherit it? Would it be correct with dynamic-dispatch on arguments?

```
class Exp {
  abstract int  eval();
  abstract unit visit(ExpVisitor);
}
class IntExp extends Exp {
  int  x;
  int  eval() { x }
  unit visit(ExpVisitor v) { v.f(self) }
}
class AddExp extends Exp {
  Exp e1; Exp e2;
  int  eval() { e1.eval() + e2.eval() }
  unit visit(ExpVisitor v) {
    e1.visit(v);
    e2.visit(v);
    v.f(self);
  }
}
```