CSE 505 Assignment 5 Solution and Grading Guide

Andy Collins
acollins@cs.washington.edu

December 11, 2003

Point distribution

As will no doubt surprise nobody, assignment 5 is graded out of 30 points. Question one was worth nine points (3 points for (1a) and 2 points each for (1b), (1c), and (1d)). Questions two and three were worth six points each (3 points each for parts (2a), (2b), (3a), and (3b)); and question four was worth nine points (again 3 points each for (4a), (4b), and (4c)).

1 Fragile superclasses

Note that in all of these cases, when D fails to typecheck it is likely that any P that uses D will also fail to typecheck. When we say that P continues to typecheck after the change, we mean that P can be made to typecheck without changing P, i.e. it is sufficient to fix D in order to make P typecheck again.

I decided to allow the law of the excluded middle when talking about subtyping, i.e. I considered $T_2 < T_1$ as an acceptable way to say $T_1 \not\leq T_2$. This makes subtyping look like inequality over the reals. It is, of course, not a technically accurate way to talk about subtyping, for two reasons:

- We never really defined a strict subtyping relationship, although we could easily have done so as $T_1 < T_2 \iff (T_1 \leq T_2 \wedge T_2 \not\leq T_1)$
- Types can be incomparable. It is perfectly valid that $T_1 \not\leq T_2 \wedge T_2 \not\leq T_1$.

As with all things of this flavor, this only applies if I could easily figure out what you meant, which I interpreted to mean you basically understood the point.

(a) The class D will not type-check if D defined a method named f with a type that is not a subtype of the type of f added to C. A client P will always continue to typecheck.

Note that it is only partially correct to say that D will fail to typecheck if it defined a method f at a different type than that added to C. As long as D's existing f method is at a subtype of the new f in C everything will be hunky-dory. I gave partial credit here under the theory that this answer captures the gist of the problem, just not the precise characterization.

(b) The class D will not type-check if it overrode f with a method taking an argument of type T_1 (or any T_1' such that $T_1 \leq T_1' \leq T_2$). A client P will always continue to typecheck.

Again it is only partially correct to omit the possibility of a T_1' where $T_1 \leq T_1'$ but $T_1' \not\leq T_1$. Again I decided that this was a partial credit area (although definitely a more picky point than for part (a)). In fact, this turned out to be one of the more commonly missed points, but enough people did get it right so that I don't feel all *that* bad about it.

(c) The class D or client P will not typecheck if they used method f of C for arguments that are strict supertypes of T_2. Note that if class D overrides method f, the overriding can still soundly typecheck though many languages do not allow subtyping on overriding.

Here I made a point of you making a point of this affecting both D and P (many of you have a cryptic comment to the effect of "can also happen to D"). I allowed the use of "T_1" in place of "a strict supertype of T_2." This is actually an interesting point (at least to me) in that the problem definition admits the possibility that T_1 and T_2 are invariant, but in that case there isn't a problem with the change to C, so there isn't an error case to find.

(d) The class D or client P will not typecheck if C was previously a concrete class and they used this information. Specifically, calling C's constructor should not longer typecheck. Furthermore, if D was concrete but no longer is (because D does not override f), then calling D's constructor should no longer typecheck. Finally, an explicit resend (super call) in D to C's f method must no longer typecheck.

This was perhaps the most common place where people found some but not all of the ways in which D or P might not typecheck. Many people also talked about how it would no longer be possible to call the f method for C objects or D objects (if D didn't override f). This is actually beside the point, because to call a method you have to have your hands on an object, and you can't even achieve that with an abstract class.

Like part (c), it was possible to notice that this could happen to P but forget that it could also happen to D (which can use C just as easily as it can extend C). I tried to not dock the point twice.

2 Self type

(a) Yes, this extension is sound. A method with return type `Self` must return the object bound to `self`. (Note in particular the only subtype of `Self` is `Self`.) If D inherits a method f from C with return-type `Self`, the late-binding ensures the method will return an object of type D.

Note that this is *not* as simple as saying that we have covariant return types. The `Self` type is weirder than covariance. Think about the case

```
class C { Self f () { self } }
class D extends C {
  int myint = 37;
  unit foo () { f().myint = 42 }
}
```

here the return type of `f()` is `Self`, meaning that it has type D and can be used as such. Therefore we need that we actually do get back a runtime D, which we do. This happens even though (in fact *because*) we didn't override the method f.

(b) No, this extension is unsound. Suppose D overrides a method f from C taking one argument of type `Self`. In the body of the method D defines, we assume `Self` $\leq D$. So for example, if D extends C with a new method g, then the method body call invoke the method g of its argument. But elsewhere, if o has type C, then we can call $o.f$ with an argument of type C, and this argument may not have a g method. Late-binding will invoke the overridden f method, which will be stuck. In pseudocode:

```
class C { unit f(Self x) { () } }
class D extends C {
    unit g() { () }
    unit f(Self x) { x.g() }
}
class Client {
    unit main() {
       C o = new D();
       o.f(new C());
    }
}
```

3 Encoding fields

(a) If class C declares a field x of type T, add methods T `get_x(){x}` and `unit set_x(`T `y){ x:=y }`. Replace all field accesses $e.x$ (or at least non-self field accesses) with e.`get_x`(). Replace all field assignments $e.x := e'$ (or at least non-self field assignments) with e.`set_x`(e').

For this part we can be a little sloppy about which self accesses to fields we replace. Obviously we cannot replace the accesses inside the get and set methods, or things will get circular. Whether we can change accesses inside the constructor is a sticky issue depending on the language (since if we did then the fields would be uninitialized when we first call the set method, and, although that isn't a problem if we're careful, it's harder to check). Other self accesses (which include both explicit accesses of the form `self.`x and implicit accesses of the form x where the `self.` is implied) can be changed or not.

(b) Assume we have replaced all field accesses and field assignments (including self accesses and assignments) as described above. Then the only use of fields are in the get and set methods (and constructors), which we change as follows: If the constructor used to initialize field x to v, we instead have it update (i.e., initialize) `get_x` to T `get_x() {`v`}`. We also have it define:

```
unit set_x(T y) {
  self.get_x := T get_x() { y }
}
```

Note that we can still do any other processing (e.g. keeping access counts) in these accessor functions that we could do before, although the set functions must be careful that the new get functions they put in place will continue to perform the same steps as those put in place by the constructor.

4 Visitor pattern

(a)
```
class ExpVisitor {
  abstract unit f(IntExp);
  abstract unit f(AddExp);
}
```

The important points here are the two statically-overloaded `f` functions, and the declaration of these functions as abstract.

(b)
```
class HasZero extends ExpVisitor {
  bool ans;
  unit f(IntExp e) {
     if(e.x == 0)
       ans := true;
  }
  unit f(AddExp e) { () }
  unit haszero(Exp e) {
    ans := false;
    e.visit(self);
    return ans;
  }
}
```

We have to override both versions of `f` in order to make the class concrete, even though the version for `AddExp` doesn't do anything (recall that the recursion is handled outside the visitor, which only needs to process the current node). The `haszero` method pretty much has to look like this, although I wasn't concerned with the details of initializing the private state.

(c) It is incorrect to move the definition of `visit` in `IntExp` into `Exp` because static overloading means the method `v.f(self)` would then be to a method accepting an `Exp` and none is known to exist in an `ExpVisitor`. With dynamic-dispatch on arguments, this call would be correct. (However, it becomes more difficult to typecheck such languages because we can only inherit `visit` if `ExpVisitor` has a method f that matches.)

I wanted to see discussion of both the dynamic-dispatch on self only and the multimethod versions. The bit about typechecking is interesting but wasn't necessary. Yes/no (or rather no,yes) answers without any explanation were worth only partial credit.