Supplementary Notes on Inheritance and Subtyping

15-312: Foundations of Programming Languages
Jonathan Aldrich

Lecture 22
November 11, 2003

In this lecture we look at the relationship between inheritance and subtyping in more detail. First, we look at the tradeoffs between structural subtyping (the form of subtyping we have studied in functional languages) and by-name subtyping (the form of subtyping that is more common in object-oriented languages). Next, we look at the differences between subtyping and inheritance, and why many object-oriented languages merge the two. Finally, we look at the fragile base class problem, a breach of modularity that can occur due to the open recursion supported by the semantics of inheritance.

Structural vs. By-Name Subtyping.

In this class, we began our study of subtyping using structural rules. For example, we say that a pair $\tau_1 \times \tau_2$ is a subtype of another pair type $\tau_1' \times \tau_2'$ if the components of the pair also have the proper subtyping relationship ($\tau_1 <: \tau_1'$ and $\tau_2 <: \tau_2'$).

In contrast, the base rule for subtyping in EML, copied below, is very different. The rule says nothing about the representation of classes C and C'–instead it just relies on the declaration stating that C inherits from C'. This is known as "by-name" subtyping, or nominal subtyping. In EML, the inheritance relationship induces a subtyping relationship–this is made explicit by the rule below. Inheritance also implies subtyping in Java, but as we will see in the next subsection, this is not the case in every language.

$$\frac{([\texttt{abstract}]\ \texttt{class}\ C\ \texttt{extends}\ C'\ \ldots) \in \mathit{Decls}}{C <: C'}\ \mathit{SubBase}$$

Note that although the subtyping rule does not refer to the representation of C and C', the rules for inheritance impose constraints that ensure that the representations are compatible. For example, C has at least the fields the C' does (because it inherits them) and C cannot change their types in an incompatible way. These rules ensure that C is substitutable for C', which constitutes the core of subtyping.

So the rules imply that if C inherits from C', then C is substitutable for C'. Somewhat suprisingly, the converse is not the case: if there is some class D that is structurally substitutable for C', but does not inherit (directly or transitively) from C', it will not be a subtype according to the rules of EML. So nominal subtyping in EML implies structural subtyping, but structural subtyping does not imply nominal subtyping.

What are the tradeoffs between the two forms of subtyping? Well, consider the following code:

```
class Bag of {es:int list}
fun add : Bag * int -> Bag
fun size : Bag -> int
```

```
class Set of {es:int list}
fun add : Set * int -> Set
fun size : Set -> int
fun union : Set * Set -> Set
```

The classes Bag and Set seem to be in a natural subtyping relationship, as determined by the data members and the functions defined over the classes. For example, Set has the same fields as Bag, and every operation that applies to a Bag (add and size, in this simplified exmaple) can be applied to Set as well.

However, despite the fact that Set is a structural subtype of Bag, it would be a mistake to consider Set as a subtype. Set is not substitutable for Bag, because "add" has different semantics on sets. For example, adding an integer to a Bag always results in a Bag whose size is greater by one, while adding an integer to a Set does not always increase the size because of the check for duplicates. If a client thought it was using a Bag object, but the Bag was really behaving like a Set, the clients internal invariants could be broken.

As this example shows, nominal inheritance is useful to avoid "spurious" examples of subtyping, where the signatures of two classes make it appear that one is substitutable for the other, but there is not a semantic subtyping relationship between them.

Another attractive features of nominal inheritance is that it supports efficient type tests at run time. Because types correspond to classes, just looking up the inheritance relationship in a table is enough to determine if two types are in a subtyping relationship. These subtyping tests are used in object-oriented dispatch, as well as in constructs like casts and `instanceof` tests in Java.

Finally, nominal inheritance is useful for modeling abstract data types. We may not want to expose the internal representation of a class to clients, but we still may want clients to know the subtyping relationship between classes. Nominal subtyping provides a natural way to do this, whereas structural subtyping would require exposing the internal representation of these classes to clients.

The advantages of nominal inheritance are compelling enough in the object-oriented setting that nearly all object-oriented languages use nominal inheritance. However, structural subtyping also has advantages. For example, consider the code below:

```
abstract class Shape of {...}
class Circle extends Shape of
         {origin:Point, radius:int, color:Color}
(* much later in the code *)
abstract class Colored of {color:Color}
```

In the example, the Colored class was defined separately from the Shape hierarchy. Since the implementor of the shapes didn't know about the Colored class, Circle was not made a subtype of Colored (although EML does not have multiple inheritance, the full language supports a notion of interface inheritance similar to that in Java). A version of EML with structural subtyping would allow us to consider Circle to be a subtype of Colored, even though that relationship was not anticipated beforehand.

In addition to supporting unanticipated subtyping relationships, structural subtyping is valuable because it combines cleanly both with primitive features of a functional language (pairs, functions, etc.) and also with more advanced constructs like polymorphism. For example, the full version of EML includes polymorphic classes, which look something like this:

```
abstract class 'a Set of {}
class 'a ListSet extends 'a Set of {es : 'a list}
```

In this extension of EML we can use a more advanced subtyping rule which combines nominal and structural subtyping. Many object-oriented languages allow one to state properties such as "'a is a covariant parameter." The rule below, for subtyping of classes with covariant parameters, would imply that `'a ListSet` is a subtype of `'b Set` if and only if `'a` is a subtype of `'b`.

$$\frac{(\texttt{[abstract] class}\ 'a\ C\ \texttt{extends}\ 'a\ C'\ \ldots) \in \mathit{Decls} \qquad 'a <:\ 'b \quad C \text{ is covariant in } 'a}{'a\ C <:\ 'b\ C'}\ \mathit{SubBase}$$

As this example shows, realistic object-oriented languages need a subtyping relation that combines features of nominal and structural subtyping.

Subtyping and Inheritance.

Although most object-oriented languages equate subtyping and inheritance, it is important to recognize that these are two distinct concepts. Some languages (for example, Cecil) provide separate constructs for declaring inheritance and subtyping relationships between classes. To see the difference, consider how one might want inheritance without subtyping, or subtyping without inheritance.

```
class Bag of {es:int list}
fun add : Bag * int -> Bag
fun size : Bag -> int
extend fun add (b as Bag, i as int) = ...
extend fun size (b as Bag) = ...
```

```
class Set inherits Bag of {}
fun add : Set * int -> Set
fun size : Set -> int
fun union : Set * Set -> Set
extend fun add (s as Set, i as int) = ...
extend fun union ...
```

The code above describes an imaginary variant of EML that separates inheritance (written "inherits") from subtyping (written "implements"). It may be convenient for the Set class to inherit methods such as size from Bag, because these methods have the same semantics. Set will override the add method to check for duplicates, and define new methods such as union. However, as discussed above, it would be semantically incorrect to treat Set as a subtype of Bag. By keeping subtyping and inheritance separate, one can reuse the Bag code for size and other functions without making Set a subtyping of Bag.

Inheritance without subtyping is known as private inheritance in C++. One problem with private inheritance is that it is hard to completely hide the inheritance relationship from clients, thus violating a key criterion of modularity. For example, a client could use a cast to observe that HashSet inherits from HashSet, even if the subtyping relationship between them was not visible to that client.

The following code demonstrates a potential use of subtyping without inheritance:

```
class Set of {es : int list}
fun add : Set * int -> Set
fun size : Set -> int
fun union : Set * Set -> Set
```

```
class HashSet implements Set
      of {buckets : array of int ...}
(* definition of HashSet functions, etc... *)
```

In the code above, the Set class has been inefficiently implemented as a linked list. An enterprising developer decides to create a HashSet that will implement the same interface, but be more efficient. There is no reason for HashSet to inherit any of the fields of Set, because they would only waste space. However, it is important that HashSet be substitutable for the original Set class, so that clients that used Set in the past can use HashSet now without being changed. Separating subtyping from inheritance allows us to specify that HashSet is a subtype of Set but does not inherit from it.

Most object-oriented languages do not provide this clean separation between subtyping and inheritance, despite the advantages we have seen.

One reason is that these advantages can be obtained with only a little bit of additional planning or rewriting of code. For example, instead of having Set inherit from Bag (in the first example), we can define a common superclass Collection. The Collection class does not pin down the semantics of add–subclasses are free to implement it either with Bag semantics or with Set semantics. Thus, we can put the common code (the size method, etc.) into Collection and have both Bag and Set be subclasses that both inherit from and subtype class collection.

In the second example, the real problem is that Set should have been an abstract class or interface. If it was written in this way, it would be easy to define separate ListSet and HashSet classes that both implement the Set interface.

As this discussion shows, current object-oriented languages survive without cleanly separating inheritance and subtyping. However, when designing object-oriented software it is important to keep the distinctions in mind in order to avoid semantic problems (such as making Set a subtype of Bag).

Open Recursion: The Fragile Base Class Problem.

Object-oriented languages, including EML, have the feature that when a class C inherits a method m from class B, and m calls some other method m2 on the same object, then C's implementation of m2 is invoked (rather than B's). This feature, called open recursion, means that code in class C can depend on the implementation details of class B, breaking the encapsulation of the superclass. The example we show here is taken from Item 14 of Joshua Bloch: *Effective Java*, Addison-Wesley, 2001,

```
public class InstrumentedHashSet extends HashSet {
    // The number of attempted element insertions
    private int addCount = 0;
    public InstrumentedHashSet () {
    }
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
}
```

Now the following sequence

```
InstrumentedHashSet s = new InstrumentedHashSet();
s.addAll(Arrays.asList(new String[]
          {"Snap", "Crackle", "Pop"}));
s.getAddCount();
```

may return either 3 or 6, depending on whether the library implementation of `addAll` has internal calls to `add` or not.

There are a number of possible solutions to this problem. One solution is to carefully document the internal calling patterns of each class. In the example above, the implementor of HashSet would have to document whether addAll calls add as part of its implementation. This solution, however, still exposes implementation decisions that should rightfully be internal to HashSet. This means that implementation decisions in addAll cannot be changed without affecting subclasses–the only difference is that the dependency is now explicitly documented.

A second solution to this problem (suggested by Bloch) is to write a wrapper class that instruments the HashSet in this way, rather than making a subclass of HashSet. For example:

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet implements Set {
    private final Set s;
    private in addCount = 0;
    public InstrumentedSet(Set s) {
        this.s = s;
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
    // Forwarding methods
    public void clear()                 { s.clear ();           }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty()            { return s.isEmpty();   }
    // ...
    public String toString ()          { return s.toString();  }
}
```

In the place of `// ...` all the relevant public methods of `s` are exported again. In ML we would use a wrapper functor instead (assuming we really wanted the implementation of `Set` to be ephemeral):

```
signature InstrumentedSet =
sig
  include Set
  val getAddCount : unit -> int
end;
functor InstrumentWrapper (structure S : Set)
      : InstrumentedSet =
struct
  val addcount = ref 0
  open S
  fun add(o) = (addcount := !addcount+1; S.add(o))
  fun addAll(c) =
      (addcount := !addcount+List.length(c);
      S.addAll(c))
  fun getAddCount() = !addCount
end;
structure InstrumentedSet =
     InstWrapper (structure S = Set);
```

Unfortunately, this solution has drawbacks as well. In Java, the developer has to write all of the forwarding methods; this is a pain, and creates spurious forwarding code that has no functional purpose, making systems harder to maintain and evolve. The ML solution avoids this problem using the open construct, which automatically imports the definitions of functions from S into the InstrumentWrapper structure.

A more serious problem is that open recursion can often be beneficial as well as harmful. For example, graphical user interface libraries use open recursion to allow programmers to specify how a window object should react to events such as menu selections and mouse clicks. Using wrapping instead of inheritance throws the baby out with the bathwater by prohibiting open recursion entirely.

Warning: research ahead!

The fragile base class problem is a great example of an important research problem that has yet to be solved cleanly. Below, I'll sketch some (largely untested) ideas for a potential solution to this problem. Although this solution may or may not work out in practice, it's an example of a problem you might tackle if you decide to go on to do research in programming languages.

Consider the following change to the semantics of Java. Method dispatch is modified to distinguish "external" and "internal" calls to object methods. An external call is a call to a method from an outside object; these calls are dispatched exactly as they are in Java. An internal call is a call to a method from within the object. We change the semantics of dispatch for internal calls to call the method in the local class, rather than the run-time class of the object.

For example, consider the InstrumentedHashSet code described above. If a client called addAll on the InstrumentedHashSet, the version of addAll in InstrumentedHashSet would be invoked. This method calls super.addAll(c), which invokes the corresponding method in HashSet. Now, assume the addAll method in HashSet is implemented so as to call add internally. Since this is an internal call, we invoke HashSet's version of add, instead of using "open recursion" to call the InstrumentedHashSet version of add (as ordinary Java would do). Thus, we avoid calling the InstrumentedHashSet's version of add and so we don't increment count beyond the correct amount.

Essentially, we are building the forwarding technique into the language, so that the programmer does not have to write explicit forwarding methods. This solves the fragile base class problem in a cleaner way than manual forwarding, and is similar to the solution provided by opening a module in ML.

As noted before, however, the forwarding technique has the additional drawback that open recursion cannot be used even when it is useful, as in GUI libraries. To ameliorate this problem, we allow the programmer to annotate some methods with the keyword "open." Methods declared "open" use the normal, open-recursion semantics of Java. For example, we could write a GUI window base class as follows. In this code, the dispatch() method is called by the system. It interprets the SystemEvent as a MouseClick event or a MenuSelection event. These events are passed to subclasses by using the "open" methods mouseClick and menuSelection.

```
class Window {
    final void dispatch(SystemEvent e) { ... }
    open void mouseClick(MouseClickEvent e) { }
    open void menuSelection(MenuSelectionEvent e) { }
}
```

We need to follow a discipline in using open so that we can gain the benefits of open recursion without the drawbacks of the fragile base class problem. One such discipline is to only declare methods as "open" if they are called when some semantic event occurs–such as a mouse click or a menu selection in the example. By tying open methods to a semantic event, a superclass such as Window is making a promise that these open methods will always be called in the same way, even if other implementation details of Window are later changed. Methods which are not declared "open" are not subject to this promise. The way in which they are used can be freely changed without affecting subclasses.

We've presented the fragile base class problem in Java, along with a potential solution using the "open" construct. You might ask, how does this work in EML? That's a hard question to answer, because in EML there's not as much of a distinction between the internals and the externals of an object. Figuring out if this solution makes sense in EML is still an open research question. Another, more technical research question is, can we do better in proving properties of a program using this new feature? There ought to be some benefits in proofs, since we are making the relationship between a superclass and its subclasses more modular. When we talk about program equivalence, we'll have the beginnings of a tool for answering this question.