CS 411 Lecture 21 — Introduction to Polymorphism & Subtyping — 24 Oct 2003

Lecturer: Greg Morrisett

There are 3 basic kinds of polymorphism:

- Subtyping (or inclusion) polymorphism.
- Parametric polymorphism.
- Ad hoc polymorphism.

Subtyping arises when we can pass an object of type τ_1 to an operation (including a function or method) that expects a τ_2 and we don't get a type-error. Most modern languages, including Java and C#, have some notion of subtyping as the basic form of polymorphism.

Parametric polymorphism is a generalization of abstract data types and languages such as ML and Haskell are notable for their inclusion of parametric polymorphism. Parametric polymorphism arises naturally for functions, such as the identity, or list reversal, where we manipulate objects in a way that's oblivious to the type. Unlike subtyping, it's possible to express input-output relationships on type signatures which gives the caller more reasoning power. But, subtype polymorphism tends to work better (or at least more easily) for dealing with heterogeneous data structures.

Note, however, that you can simulate subtyping in a pretty straightforward way with parametric polymorphism, but it requires passing around coercion functions. You can also simulate parametric polymorphism in a language like Java, but it requires run-time type tests. So neither situation is ideal, and next-generation languages, including Java 1.5 and C# include a notion of "generics" which generalizes both subtyping and parametric polymorphism.

Ad hoc polymoprhism is everything else. Overloading is a good example. Something like "+" appears to work on both integers and reals, but note that we get different code for the different operations. Furthermore, "+" is only defined on some types and there's no way to abstract over those types. Languages such as C++ provide a lot of support for ad hoc polymorphism through templates, but anyone who has used template libraries can tell you that this was not well thought through. Language researchers, particularly in the Haskell camp, have been working on better ways to support the power of ad hoc polymorphism (sometimes called polytypism) and this, coupled with notions of reflection, are hot research topics. This is mostly because we don't have a clean handle on these concepts the same way we do subtyping and parametric polymorphism. The funny thing is that it's only in the last 15 years that we've gotten a good handle on those two!

Subtyping

This arises naturally if we think about the idea of "types as sets", for then "subtypes are subsets". More abstractly, the principle behind subtyping is that, if you can use a value of type τ_1 anywhere that you expect to have a value of type τ_2, then you ought to be able to treat τ_1 as a subtype of τ_2.

For example, given record types:

```
type point2d = {x:int, y:int}
type point3d = {x:int, y:int, z:int}
```

then we ought to be able to pass in a point3d to any function expecting a point2d, because all that function can do is select out the x or y components, and those operations are defined on point3d values as well as point2d values.

If we think of point2d and point3d as interfaces (à la Java or C#) then it's natural to think that anything that implements the point3d interface also implements the point2d interface.

We can extend our little functional language with subtyping by adding one new rule called subsumption:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

The rule says that if e has type τ_1 and τ_1 is a subtype of τ_2, then e also has type τ_2.

Of course, now we need to define what it means for one type to be a subtype of another. Remember the principle to follow is that $\tau_1 \leq \tau_2$ means that for any operation op you can perform on a τ_2 value, you should be able to perform op on a τ_1 value.

All subtyping frameworks yield a partial order so they include rules for reflexivity and transitivity:

$$\tau \leq \tau$$

$$\frac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

For something like (immutable) records, the subtyping rule might look like this:

$$\frac{\tau_1 \leq \tau_1' \qquad \ldots \qquad \tau_n \leq \tau_n'}{\{l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+1}\ \tau_{n+1}\} \leq \{l_1 : \tau_1', \ldots, l_n : \tau_n'\}}$$

This rule says that a record type R_1 is a subtype of a record type R_2 if for all $l : \tau'$ in R_2, there exists a label l with type τ in R_1, and furthermore, $\tau \leq \tau'$. This rule is sound because the only operation we can apply to an R_2 value is to extract a component using the $\#l$ operation. If $\#l$ is defined in R_1, then the operation will succeed. And intuitively, if that yields a value of type τ, then we can use subtyping to show that $\tau \leq \tau'$.

What about sums? Consider the datatypes:

```
datatype t1 = C1 of int
datatype t2 = C1 of int | C2 of bool
datatype t3 = C1 of int | C2 of bool | C3 of real
```

The only operation on sums is to do a case on them. If I have a case that can handle all of the constructors of t3, then it's safe to pass in a t2 or a t1 value because the set of constructors in those types is a subset of the set used in the definition of t3. So, for sums, the subtyping rule is the opposite – the subtype must have fewer labels.

$$\frac{\tau_1' \leq \tau_1 \qquad \ldots \qquad \tau_n' \leq \tau_n}{[l_1 \text{ of } \tau_1 \mid \ldots \mid l_n \text{ of } \tau_n] \leq [l_1 \text{ of } \tau_1' \mid \ldots \mid l_n \text{ of } \tau_n' \mid l_{n+1} \text{ of } \tau_{n+1}]}$$

Notice also that we assume that $\tau_1' \leq \tau_1$ instead of the other way around for records. So, once again, sums are pleasingly dual to products.

What about functions? The only operation on a function is to apply it to some argument. So, suppose I have a function:

$$f : (\{x : \mathsf{int}, y : \mathsf{int}\} \to \{x : \mathsf{int}, y : \mathsf{int}\}) \to \mathsf{int}$$

Suppose that I have another function:

$$g : \{x : \mathsf{int}\} \to \{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}$$

Does it make sense to pass g to f? That is, what can go wrong if we do $f(g)$?

Well, f might try to apply g to some argument. It thinks that g takes in $\{x : \mathsf{int}, y : \mathsf{int}\}$ arguments. So, it might pass an $\{x : \mathsf{int}, y : \mathsf{int}\}$ value to g. Now g thinks that it's being given an argument of type $\{x : \mathsf{int}\}$, so the only thing it can do is apply $\#x$ to the argument. That seems to be okay because conceptually, we can use subtyping to treat the $\{x : \mathsf{int}, y : \mathsf{int}\}$ value as if it's a $\{x : \mathsf{int}\}$ value.

What about the result? f thinks that g returns $\{x : \mathsf{int}, y : \mathsf{int}\}$ so it might do $\#x$ or $\#y$ on the result. But g returns a $\{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}$ value. Again, this is okay because the operations that f might do on g's result are okay. Or conceptually, we can apply subtyping to coerce g's result to $\{x : \mathsf{int}, y : \mathsf{int}\}$ if you like.

So it appears as if g's type is a subtype of the argument type of f. That is, we have:

$$(\{x : \mathsf{int}\} \to \{x : \mathsf{int}, y : \mathsf{int}, z : \mathsf{int}\}) \leq (\{x : \mathsf{int}, y : \mathsf{int}\} \to \{x : \mathsf{int}, y : \mathsf{int}\})$$

In general, the subtyping rule for functions is:

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{(\tau_1 \to \tau_2) \leq (\tau_1' \to \tau_2')}$$

So note that the argument types are **contra-variant** while the result types are **co-variant**. There are a lot of ways to see why contra-variance in the argument type is necessary. The example above demonstrates this, but so does drawing a Venn diagram of sets, subsets, and mappings from one set to another. Or from a logical standpoint, we can think of $(\tau_1 \to \tau_2)$ as similar to $(\neg\tau_1 \vee \tau_2)$ — the negation on the argument suggests that we'll have to flip the subtyping around.

Subtypes as Subsets vs. Coercions

Earlier, I said that we can think of subtypes as "subsets" and this is the general principle that most languages follow. However, it requires that we have a uniform representation for values that lie in related types. For instance, in Java, all Objects are one word so that we can pass them around, stick them in data structures, etc. But that means that double values are not Objects. A **boxed** double value (e.g., Double) is a subtype of Object, and we can coerce a double to a Double. So, in some sense, double is a "coerceable-subtype" of Object.

In fact, we can think of all of the subtyping rules as being witnessed by some coercion function. In particular, if we think of $\tau_1 \leq \tau_2$ as a function $C : \tau_1 \to \tau_2$, then the subsumption rule becomes:

$$\frac{\Gamma \vdash e : \tau_1 \qquad C : \tau_1 \to \tau_2}{\Gamma \vdash C\ e : \tau_2}$$

The other coercions can be calculated as follows:

$$\lambda x : \tau.\, x : \tau \to \tau \qquad \text{(reflexivity is the identity)}$$

$$\frac{C_1 : \tau_1 \to \tau_2 \qquad C_2 : \tau_2 \to \tau_3}{C_1 \circ C_2 : \tau_3} \qquad \text{(transitivity is composition)}$$

Then record subtyping is witnessed by this coercion:

$$\frac{C_1 : \tau_1 \to \tau_1' \qquad \ldots \qquad C_n : \tau_n \to \tau_n'}{\lambda r : \{l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+1} : \tau_{n+1}\}.\, \{l_1 = C_1(\#l_1\ r), \ldots, l_n = C_n\#l_n\ \}}$$

Coercion-based subtyping is simpler from the perspective of the language designer and implementor, and it doesn't require us to have uniform representations for objects. But coercions aren't so attractive for the programmer. First, they cost time. Second, they don't work for mutable objects since they fundamentally operate by making a copy of the object. (You can achieve this by providing a level of indirection. That's what Java interfaces do.)

Another issue with coercions is that you must have some notion of coherence. This means that, no matter how we prove that the code type-checks, its meaning doesn't change. In particular, if I have a proof $\mathcal{P}_1$ that $\vdash e : \tau$ and another proof $\mathcal{P}_2$ that $\vdash e : \tau$, whether I use $\mathcal{P}_1$ to insert the coercions or $\mathcal{P}_2$ shouldn't matter. I should get the same value out at the end of the day.

Note that the coercions of double to Double in Java are not coherent. The problem is that if d is a double, then

$$d == d$$

does not return the same thing as

$$(\mathsf{Double})d == (\mathsf{Double})d$$

Similar problems arise in C. In general, (implicit) coercions turn out to be a good idea only in pure, functional languages. In imperative languages, it's almost always a disaster (witness the problems of arithmetic in C/C++.)