

Coursera Programming Languages Course

Section 10 Summary

Standard Description: This summary covers roughly the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.

Contents

Introduction to Subtyping	1
A Made-Up Language of Records	2
Wanting Subtyping	3
The Subtyping Relation	3
Depth Subtyping: A Bad Idea With Mutation	4
Optional: The Problem With Java/C# Array Subtyping	5
Function Subtyping	7
Subtyping for OOP	9
Optional: Covariant self/this	10
Generics Versus Subtyping	10
Bounded Polymorphism	13
Optional: Additional Java-Specific Bounded Polymorphism	14

Introduction to Subtyping

We previously studied static types for functional programs, in particular ML’s type system. ML uses its type system to prevent errors like treating a number as a function. A key source of expressiveness in ML’s type system (not rejecting too many programs that do nothing wrong and programmers are likely to write) is *parametric polymorphism*, also known as *generics*.

So we should also study static types for object-oriented programs, such as those found in Java. If everything is an object (which is less true in Java than in Ruby), then the main thing we would want our type system to prevent is “method missing” errors, i.e., sending a message to an object that has no method for that message. If objects have fields accessible from outside the object (e.g., in Java), then we also want to prevent “field missing” errors. There are other possible errors as well, like calling a method with the wrong number of arguments.

While languages like Java and C# have generics these days, the source of type-system expressiveness most fundamental to object-oriented style is *subtype polymorphism*, also known as *subtyping*. ML does not have subtyping, though this decision is really one of language design (it would complicate type inference, for example).

It would be natural to study subtyping using Java since it is a well-known object-oriented language with a type system that has subtyping. But it is also fairly complicated, using classes and interfaces for types that describe objects with methods, overriding, static overloading, etc. While these features have pluses and minuses, they can complicate the fundamental ideas that underlie how subtyping should work in any language.

So while we will briefly discuss subtyping in OOP, we will mostly use a small language with *records* (like in ML, things with named fields holding contents — basically objects with public fields, no methods, and no class names) and functions (like in ML or Racket). This will let us see how subtyping should — and should not — work.

This approach has the disadvantage that we cannot use any of the language we have studied: ML does not have subtyping and record fields are immutable, Racket and Ruby are dynamically typed, and Java is too complicated for our starting point. So we are going to make up a language with just records, functions, variables, numbers, strings, etc. and explain the meaning of expressions and types as we go.

A Made-Up Language of Records

To study the basic ideas behind subtyping, we will use records with mutable fields, as well as functions and other expressions. Our syntax will be a mix of ML and Java that keeps examples short and, hopefully, clear. For records, we will have expressions for making records, getting a field, and setting a field as follows:

- In the expression `{f1=e1, f2=e2, ..., fn=en}`, each `fi` is a field name and each `ei` is an expression. The semantics is to evaluate each `ei` to a value `vi` and the result is the record value `{f1=v1, f2=v2, ..., fn=vn}`. So a record value is just a collection of fields, where each field has a name and a contents.
- For the expression `e.f`, we evaluate `e` to a value `v`. If `v` is a record with an `f` field, then the result is the contents of the `f` field. Our type system will ensure `v` has an `f` field.
- For the expression `e1.f = e2`, we evaluate `e1` and `e2` to values `v1` and `v2`. If `v1` is a record with an `f` field, then we update the `f` field to have `v2` for its contents. Our type system will ensure `v1` has an `f` field. Like in Java, we will choose to have the result of `e1.f = e2` be `v2`, though usually we do not use the result of a field-update.

Now we need a type system, with a form of types for records and typing rules for each of our expressions. Like in ML, let's write record types as `{f1:t1, f2:t2, ..., fn:tn}`. For example, `{x : real, y : real}` would describe records with two fields named `x` and `y` that hold contents of type `real`. And `{foo: {x : real, y : real}, bar : string, baz : string}` would describe a record with three fields where the `foo` field holds a (nested) record of type `{x : real, y : real}`. We then type-check expressions as follows:

- If `e1` has type `t1`, `e2` has type `t2`, ..., `en` has type `tn`, then `{f1=e1, f2=e2, ..., fn=en}` has type `{f1:t1, f2:t2, ..., fn:tn}`.
- If `e` has a record type containing `f : t`, then `e.f` has type `t` (else `e.f` does not type-check).
- If `e1` has a record type containing `f : t` and `e2` has type `t`, then `e1.f = e2` has type `t` (else `e1.f = e2` does not type-check).

Assuming the “regular” typing rules for other expressions like variables, functions, arithmetic, and function calls, an example like this will type-check as we would expect:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

In particular, the function `distToOrigin` has type `{x : real, y : real} -> real`, where we write function types with the same syntax as in ML. The call `distToOrigin(pythag)` passes an argument of the right type, so the call type-checks and the result of the call expression is the return type `real`.

This type system does what it is intended to do: No program that type-checks would, when evaluated, try to look up a field in a record that does not have that field.

Wanting Subtyping

With our typing rules so far, this program would not type-check:

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

In the call `distToOrigin(c)`, the type of the argument is `{x:real,y:real,color:string}` and the type the function expects is `{x:real,y:real}`, breaking the typing rule that functions must be called with the type of argument they expect. Yet the program above is safe: running it would not lead to accessing a field that does not exist.

A natural idea is to make our type system more lenient as follows: If some expression has a record type `{f1:t1, ..., fn:tn}`, then let the expression *also* have a type where some of the fields are removed. Then our example will type-check: Since the expression `c` has type `{x:real,y:real,color:string}`, it can also have type `{x:real,y:real}`, which allows the call to type-check. Notice we could also use `c` as an argument to a function of type `{color:string}->int`, for example.

Letting an expression that has one type also have another type that has less information is the idea of *subtyping*. (It may seem backwards that the *subtype* has *more* information, but that is how it works. A less-backwards way of thinking about it is that there are “*fewer*” values of the subtype than of the supertype because values of the subtype have more obligations, e.g., having more fields.)

The Subtyping Relation

We will now add subtyping to our made-up language, in a way that will not require us to change any of our existing typing rules. For example, we will leave the function-call rule the same, still requiring that the type of the actual argument *equal* the type of the function parameter in the function definition. To do this, we will add two things to our type system:

- The idea of one type being a subtype of another: We will write `t1 <: t2` to mean `t1` is a subtype of `t2`.
- One and only new typing rule: If `e` has type `t1` and `t1 <: t2`, then `e` (also) has type `t2`.

So now we just need to give rules for `t1 <: t2`, i.e., when is one type a subtype of another. This approach is good language engineering — we have separated the idea of subtyping into a single binary relation that we can define separately from the rest of the type system.

A common misconception is that if we are defining our own language, then we can make the typing and subtyping rules whatever we want. That is only true if we forget that our type system is allegedly preventing

something from happening when programs run. If our goal is (still) to prevent field-missing errors, then we cannot add any subtyping rules that would cause us to stop meeting our goal. This is what people mean when they say, “Subtyping is not a matter of opinion.”

For subtyping, the key guiding principle is *substitutability*: If we allow $t_1 <: t_2$, then any value of type t_1 must be able to be used in every way a t_2 can be. For records, that means t_1 should have all the fields that t_2 has and with the same types.

Some Good Subtyping Rules

Without further ado, we can now give four subtyping rules that we can add to our language to accept more programs without breaking the type system. The first two are specific to records and the next two, while perhaps seeming unnecessary, do no harm and are common in any language with subtyping because they combine well with other rules:

- “Width” subtyping: A supertype can have a subset of fields with the same types, i.e., a subtype can have “extra” fields
- “Permutation” subtyping: A supertype can have the same set of fields with the same types in a different order.
- Transitivity: If $t_1 <: t_2$ and $t_2 <: t_3$, then $t_1 <: t_3$.
- Reflexivity: Every type is a subtype of itself: $t <: t$.

Notice that width subtyping lets us forget fields, permutation subtyping lets us reorder fields (e.g., so we can pass a `{x:real,y:real}` in place of a `{y:real,x:real}`) and transitivity with those rules lets us do both (e.g., so we can pass a `{x:real,foo:string,y:real}` in place of a `{y:real,x:real}`).

Depth Subtyping: A Bad Idea With Mutation

Our subtyping rules so far let us drop fields or reorder them, but there is no way for a supertype to have a field with a different type than in the subtype. For example, consider this example, which passes a “sphere” to a function expecting a “circle.” Notice that circles and spheres have a `center` field that itself holds a record.

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
  c.center.y

val sphere:{center:{x:real,y:real,z:real}, r:real} = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = circleY(sphere)
```

The type of `circleY` is `{center:{x:real,y:real}, r:real}->real` and the type of `sphere` is `{center:{x:real,y:real,z:real}, r:real}`, so the call `circleY(sphere)` can type-check only if

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

This subtyping does not hold with our rules so far: We can drop the `center` field, drop the `r` field, or reorder those fields, but we cannot “reach into a field type to do subtyping.”

Since we might like the program above to type-check since evaluating it does nothing wrong, perhaps we should add another subtyping rule to handle this situation. The natural rule is “depth” subtyping for records:

- “Depth” subtyping: If $ta <: tb$, then $\{f_1:t_1, \dots, f_n:t_n\} <: \{f_1:t_1, \dots, f_m:t_m, \dots, f_n:t_n\}$.

This rule lets us use width subtyping on the field `center` to show

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

so the program above now type-checks.

Unfortunately, this rule breaks our type system, allowing programs that we do not want to allow to type-check! This may not be intuitive and programmers make this sort of mistake often — thinking depth subtyping should be allowed. Here is an example:

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
  c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real} = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = setToOrigin(sphere)
val _ = sphere.center.z
```

This program type-checks in much the same way: The call `setToOrigin(sphere)` has an argument of type `{center:{x:real,y:real,z:real}, r:real}` and uses it as a `{center:{x:real,y:real}, r:real}`. But what happens when we run this program? `setToOrigin` mutates its argument so the `center` field holds a record *with no z field!* So the last line, `sphere.center.z` will not work: it tries to read a field that does not exist.

The moral of the story is simple if often forgotten: In a language with records (or objects) with getters and setters for fields, depth subtyping is unsound — you cannot have a different type for a field in the subtype and the supertype.

Note, however, that if a field is not settable (i.e., it is immutable), then the depth subtyping rule is sound and, like we saw with `circleY`, useful. So this is yet another example of how not having mutation makes programming easier. In this case, it allows more subtyping, which lets us reuse code more.

Another way to look at the issue is that given the three features of (1) setting a field, (2) letting depth subtyping change the type of a field, and (3) having a type system actually prevent field-missing errors, you can have any two of the three.

Optional: The Problem With Java/C# Array Subtyping

Now that we understand depth subtyping is unsound if record fields are mutable, we can question how Java and C# treat subtyping for arrays. For the purpose of subtyping, arrays are very much like records, just with field names that are numbers and all fields having the same type. (Since $e_1[e_2]$ computes what index to access and the type system does not restrict what index might be the result, we need all fields to have the same type so that the type system knows the type of the result.) So it should very much surprise us that this code type-checks in Java:

```
class Point { ... } // has fields double x, y
class ColorPoint extends Point { ... } // adds field String color
...
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
```

```

}

String m2(int x) {
    ColorPoint[] cpt_arr = new ColorPoint[x];
    for(int i=0; i < x; i++)
        cpt_arr[i] = new ColorPoint(0,0,"green");
    m1(cpt_arr);
    return cpt_arr[0].color;
}

```

The call `m1(cpt_arr)` uses subtyping with `ColorPoint[] <: Point[]`, which is essentially depth subtyping even though array indices are mutable. As a result, it appears that `cpt_arr[0].color` will read the `color` field of an object that does not have such a field.

What actually happens in Java and C# is the assignment `pt_arr[0] = new Point(3,4);` will raise an exception if `pt_arr` is actually an array of `ColorPoint`. In Java, this is an `ArrayStoreException`. The advantage of having the store raise an exception is that no other expressions, such as array reads or object-field reads, need run-time checks. The invariant is that an object of type `ColorPoint[]` always holds objects that have type `ColorPoint` or a subtype, not a supertype like `Point`. Since Java allows depth subtyping on arrays, it cannot maintain this invariant statically. Instead, it has a run-time check on all array assignments, using the “actual” type of array elements and the “actual” class of the value being assigned. So even though in the type system `pt_arr[0]` and `new Point(3,4)` both have type `Point`, this assignment can fail at run-time.

As usual, having run-time checks means the type system is preventing fewer errors, requiring more care and testing, plus the run-time cost of performing these checks on array updates. So why were Java and C# designed this way? It seemed important for flexibility before these languages had generics so that, for example, if you wrote a method to sort an array of `Point` objects, you could use your method to sort an array of `ColorPoint` objects. Allowing this makes the type system simpler and less “in your way” at the expense of statically checking less. Better solutions would be to use generics in combination with subtyping (see bounded polymorphism in the next lecture) or to have support for indicating that a method will not update array elements, in which case depth subtyping is sound.

null in Java/C#

While we are on the subject of pointing out places where Java/C# choose dynamic checking over the “natural” typing rules, the far more ubiquitous issue is how the constant `null` is handled. Since this value has no fields or methods (in fact, unlike `nil` in Ruby, it is not even an object), its type should naturally reflect that it cannot be used as the receiver for a method or for getting/setting a field. Instead, Java and C# allow `null` to have *any* object type, as though it defines *every* method and has *every* field. From a static checking perspective, this is exactly backwards. As a result, the language definition has to indicate that *every* field access and method call includes a run-time check for `null`, leading to the `NullPointerException` errors that Java programmers regularly encounter.

So why were Java and C# designed this way? Because there are situations where it is very convenient to have `null`, such as initializing a field of type `Foo` before you can create a `Foo` instance (e.g., if you are building a cyclic list). But it is also very common to have fields and variables that should never hold `null` and you would like to have help from the type-checker to maintain this invariant. Many proposals for incorporating “cannot be `null`” types into programming languages have been made, but none have yet “caught on” for Java or C#. In contrast, notice how ML uses option types for similar purposes: The types `t option` and `t` are not the same type; you have to use `NONE` and `SOME` constructors to build a datatype where values might or might not actually have a `t` value.

Function Subtyping

The rules for when one function type is a subtype of another function type are even less intuitive than the issue of depth subtyping for records, but they are just as important for understanding how to safely override methods in object-oriented languages (see below).

When we talk about function subtyping, we are talking about using a function of one type in place of a function of another type. For example, if `f` takes a function `g` of type `t1->t2`, can we pass a function of type `t3->t4` instead? If `t3->t4` is a subtype of `t1->t2` then this is allowed because, as usual, we can pass the function `f` an argument that is a subtype of the type expected. But this is not “function subtyping” on `f` — it is “regular” subtyping on function arguments. The “function subtyping” is deciding that one function type is a subtype of another.

To understand function subtyping, let’s use this example of a higher-order function, which computes the distance between the two-dimensional point `p` and the result of calling `f` with `p`:

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},  
              p : {x:real,y:real}) =  
  let val p2 : {x:real,y:real} = f p  
    val dx : real = p2.x - p.x  
    val dy : real = p2.y - p.y  
  in Math.sqrt(dx*dx + dy*dy) end
```

The type of `distMoved` is

```
(({x:real,y:real}->{x:real,y:real}) * {x:real,y:real}) -> real
```

So a call to `distMoved` requiring no subtyping could look like this:

```
fun flip p = {x=~p.x, y=~p.y}  
val d = distMoved(flip, {x=3.0, y=4.0})
```

The call above could also pass in a record with extra fields, such as `{x=3.0,y=4.0,color="green"}`, but this is just ordinary width subtyping on the second argument to `distMoved`. Our interest here is deciding what functions with types other than `{x:real,y:real}->{x:real,y:real}` can be passed for the first argument to `distMoved`.

First, it is safe to pass in a function with a return type that “promises” more, i.e., returns a subtype of the needed return type for the function `{x:real,y:real}`. For example, it is fine for this call to type-check:

```
fun flipGreen p = {x=~p.x, y=~p.y, color="green"}  
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

The type of `flipGreen` is

```
{x:real,y:real} -> {x:real,y:real,color:string}
```

This is safe because `distMoved` expects a `{x:real,y:real}->{x:real,y:real}` and `flipGreen` is substitutable for values of such a type since the fact that `flipGreen` returns a record that also has a `color` field is not a problem.

In general, the rule here is that if $ta <: tb$, then $t \rightarrow ta <: t \rightarrow tb$, i.e., the subtype can have a return type that is a subtype of the supertype's return type. To introduce a little bit of jargon, we say return types are *covariant* for function subtyping meaning the subtyping for the return types works “the same way” (co) as for the types overall.

Now let us consider passing in a function with a different argument type. It turns out argument types are NOT covariant for function subtyping. Consider this example call to `distMoved`:

```
fun flipIfGreen p = if p.color = "green"
    then {x=~p.x, y=~p.y}
    else {x=p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

The type of `flipIfGreen` is

```
{x:real,y:real,color:string} -> {x:real,y:real}
```

This program should not type-check: If we run it, the expression `p.color` will have a “no such field” error since the point passed to `flipIfGreen` does not have a `color` field. In short, $ta <: tb$, does NOT mean $ta \rightarrow t <: tb \rightarrow t$. This would amount to using a function that “needs more of its argument” in place of a function that “needs less of its argument.” This breaks the type system since the typing rules will not require the “more stuff” to be provided.

But it turns out it works just fine to use a function that “needs less of its argument” in place of a function that “needs more of its argument.” Consider this example use of `distMoved`:

```
fun flipX_Y0 p = {x=~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

The type of `flipX_Y0` is

```
{x:real} -> {x:real,y:real}
```

since the only field the argument to `flipX_Y0` needs is `x`. And the call to `distMoved` causes no problem: `distMoved` will always call its `f` argument with a record that has an `x` field and a `y` field, which is more than `flipX_Y0` needs.

In general, the treatment of argument types for function subtyping is “backwards” as follows: If $tb <: ta$, then $ta \rightarrow t <: tb \rightarrow t$. The technical jargon for “backwards” is *contravariance*, meaning the subtyping for argument types is the reverse (contra) of the subtyping for the types overall.

As a final example, function subtyping can allow contravariance of arguments and covariance of results:

```
fun flipXMakeGreen p = {x=~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

Here `flipXMakeGreen` has type

```
{x:real} -> {x:real,y:real,color:string}
```

This is a subtype of

```
{x:real,y:real} -> {x:real,y:real}
```

because $\{x:real,y:real\} <: \{x:real\}$ (contravariance on arguments) and $\{x:real,y:real,color:string\} <: \{x:real,y:real\}$ (covariance on results).

The general rule for function subtyping is: If $t_3 <: t_1$ and $t_2 <: t_4$, then $t_1 \rightarrow t_2 <: t_3 \rightarrow t_4$. This rule, combined with reflexivity (every type is a subtype of itself) lets us use contravariant arguments, covariant results, or both.

Argument contravariance is the least intuitive concept in the course, but it is worth burning into your memory so that you do not forget it. Many very smart people get confused because it is *not* about calls to methods/functions. Rather it is about the methods/functions themselves. We do not need function subtyping for passing non-function arguments to functions: we can just use other subtyping rules (e.g., those for records). Function subtyping is needed for higher-order functions or for storing functions themselves in records. And object types are related to having records with functions (methods) in them.

Subtyping for OOP

As promised, we can now apply our understanding of subtyping to OOP languages like Java or C#.

An object is basically a record holding fields (which we assume here are mutable) and methods. We assume the “slots” for methods are immutable: If an object’s method m is implemented with some code, then there is no way to mutate m to refer to different code. (An instance of a subclass could have different code for m , but that is different than mutating a record field.)

With this perspective, sound subtyping for objects follows from sound subtyping for records and functions:

- A subtype can have extra fields.
- Because fields are mutable, a subtype cannot have a different type for a field.
- A subtype can have extra methods.
- Because methods are immutable, a subtype can have a subtype for a method, which means the method in the subtype can have contravariant argument types and a covariant result type.

That said, object types in Java and C# do not look like record types and function types. For example, we cannot write down a type that looks something like:

```
{fields : x:real, y:real, ...
methods: distToOrigin : () -> real, ...}
```

Instead, we reuse class names as types where if there is a class `Foo`, then the type `Foo` includes in it all fields and methods implied by the class definition (including superclasses). And, as discussed previously, we also have interfaces, which are more like record types except they do not include fields and we use the name of the interface as a type. Subtyping in Java and C# includes only the subtyping explicitly stated via the subclass relationship and the interfaces that classes explicitly indicate they implement (including interfaces implemented by superclasses).

All said, this approach is more restrictive than subtyping requires, but since it does not allow anything it should not, it soundly prevents “field missing” and “method missing” errors. In particular:

- A subclass can add fields but not remove them

- A subclass can add methods but not remove them
- A subclass can override a method with a covariant return type
- A class can implement more methods than an interface requires or implement a required method with a covariant return type

Classes and types are different things! Java and C# purposely confuse them as a matter of convenience, but you should keep the concepts separate. A class defines an object's behavior. Subclassing inherits behavior, modifying behavior via extension and override. A type describes what fields an object has and what messages it can respond to. Subtyping is a question of substitutability and what we want to flag as a type error. So try to avoid saying things like, “overriding the method in the supertype” or, “using subtyping to pass an argument of the superclass.” That said, this confusion is understandable in languages where every class declaration introduces a class and a type with the same name.

Optional: Covariant self/this

As a final subtle detail and advanced point, Java's `this` (i.e., Ruby's `self`) is treated specially from a type-checking perspective. When type-checking a class `C`, we know `this` will have type `C` or a subtype, so it is sound to assume it has type `C`. In a subtype, e.g., in a method overriding a method in `C`, we can assume `this` has the subtype. None of this causes any problems, and it is essential for OOP. For example, in class `B` below, the method `m` can type-check only if `this` has type `B`, not just `A`.

```
class A {
    int m(){ return 0; }
}
class B extends A {
    int x;
    int m(){ return x; }
}
```

But if you recall our manual encoding of objects in Racket, the encoding passed `this` as an extra explicit *argument* to a method. That would suggest *contravariant* subtyping, meaning `this` in a subclass could not have a *subtype*, which it needs to have in the example above.

It turns out `this` is special in the sense that while it is like an extra argument, it is an argument that is covariant. How can this be? Because it is not a “normal” argument where callers can choose “anything” of the correct type. Methods are always called with a `this` argument that is a subtype of the type the method expects.

This is the main reason why coding up dynamic dispatch manually works much less well in statically typed languages, even if they have subtyping: You need special support in your type system for `this`.

Generics Versus Subtyping

We have now studied both subtype polymorphism, also known as subtyping, and parametric polymorphism, also known as generic types, or just generics. So let's compare and contrast the two approaches, demonstrating what each is designed for.

What are generics good for?

There are many programming idioms that use generic types. We do not consider all of them here, but let's reconsider probably the two most common idioms that came up when studying higher-order functions.

First, there are functions that combine other functions such as `compose`:

```
val compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
```

Second, there are functions that operate over collections/containers where different collections/containers can hold values of different types:

```
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

In all these cases, the key point is that if we had to pick non-generic types for these functions, we would end up with significantly less code reuse. For example, we would need one `swap` function for producing an `int * bool` from a `bool * int` and another `swap` function for swapping the positions of an `int * int`.

Generic types are much more useful and precise than just saying that some argument can “be anything.” For example, the type of `swap` indicates that the second component of the result has the same type as the first component of the argument and the first component of the result has the same type as the second component of the argument. In general, we reuse a type variable to indicate when multiple things can have any type but must be the same type.

Optional: Generics in Java

Java has had subtype polymorphism since its creation in the 1990s and has had parametric polymorphism since 2004. Using generics in Java can be more cumbersome without ML’s support for type inference and, as a separate matter, closures, but generics are still useful for the same programming idioms. Here, for example, is a generic `Pair` class, allowing the two fields to have any type:

```
class Pair<T1,T2> {
    T1 x;
    T2 y;
    Pair(T1 _x, T2 _y){ x = _x; y = _y; }
    Pair<T2,T1> swap() {
        return new Pair<T2,T1>(y,x);
    }
    ...
}
```

Notice that, analogous to ML, “`Pair`” is not a type: something like `Pair<String, Integer>` is a type. The `swap` method is, in object-oriented style, an instance method in `Pair<T1, T2>` that returns a `Pair<T2, T1>`. We could also define a static method:

```
static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> p) {
    return new Pair<T2,T1>(p.y,p.x);
}
```

For reasons of backwards-compatibility, the previous paragraph is not quite true: Java also has a type `Pair` that “forgets” what the types of its fields are. Casting to and from this “raw” type leads to compile-time warnings that you would be wise to heed: Ignoring them can lead to run-time errors in places you would not expect.

Subtyping is a Bad Substitute for Generics

If a language does not have generics or a programmer is not comfortable with them, one often sees generic code written in terms of subtyping instead. Doing so is like painting with a hammer instead of a paintbrush: technically possible, but clearly the wrong tool. Consider this Java example:

```
class LamePair {  
    Object x;  
    Object y;  
    LamePair(Object _x, Object _y){ x=_x; y=_y; }  
    LamePair swap() { return new LamePair(y,x); }  
    ...  
}  
  
String s = (String)(new LamePair("hi",4).y); // error caught only at run-time
```

The code in `LamePair` type-checks without problem: the fields `x` and `y` have type `Object`, which is a supertype of every class and interface. The difficulties arise when clients use this class. Passing arguments to the constructor works as expected with subtyping.¹ But when we retrieve the contents of a field, getting an `Object` is not very useful: we want the type of value we put back in.

Subtyping does not work that way: the type system knows only that the field holds an `Object`. So we have to use a *downcast*, e.g., `(String)e`, which is a run-time check that the result of evaluating `e` is actually of type `String`, or, in general, a subtype thereof. Such run-time checks have the usual dynamic-checking costs in terms of performance, but, more importantly, in terms of the possibility of failure: this is not checked statically. Indeed, in the example above, the downcast would fail: it is the `x` field that holds a `String`, not the `y` field.

In general, when you use `Object` and downcasts, you are essentially taking a dynamic typing approach: any object could be stored in an `Object` field, so it is up to programmers, without help from the type system, to keep straight what kind of data is where.

What is Subtyping Good For?

We do not suggest that subtyping is not useful: It is great for allowing code to be reused with data that has “extra information.” For example, geometry code that operates over points should work fine for colored-points. It is certainly inconvenient in such situations that ML code like this simply does not type-check:

```
fun distToOrigin1 {x=x,y=y} =  
    Math.sqrt (x*x + y*y)  
  
(* does not type-check *)  
(* val five = distToOrigin1 {x=3.0,y=4.0,color="red"} *)
```

A generally agreed upon example where subtyping works well is graphical user interfaces. Much of the code for graphics libraries works fine for any sort of graphical element (“paint it on the screen,” “change the background color,” “report if the mouse is clicked on it,” etc.) where different elements such as buttons, slider bars, or text boxes can then be subtypes.

Generics are a Bad Substitute for Subtyping

In a language with generics instead of subtyping, you can code up your own code reuse with higher-order functions, but it can be quite a bit of trouble for a simple idea. For example, `distToOrigin2` below uses

¹Java will automatically convert a 4 to an `Integer` object holding a 4.

getters passed in by the caller to access the `x` and `y` fields and then the next two functions have different types but identical bodies, just to appease the type-checker.

```
fun distToOrigin2(getx, gety, v) =
  let
    val x = getx v
    val y = gety v
  in
    Math.sqrt (x*x + y*y)
  end

fun distToOriginPt (p : {x:real,y:real}) =
  distToOrigin2(fn v => #x v,
               fn v => #y v,
               p)

fun distToOriginColorPt (p : {x:real,y:real,color:string}) =
  distToOrigin2(fn v => #x v,
               fn v => #y v,
               p)
```

Nonetheless, without subtyping, it may sometimes be worth writing code like `distToOrigin2` if you want it to be more reusable.

Bounded Polymorphism

As Java and C# demonstrate, there is no reason why a statically typed programming language cannot have generic types and subtyping. There are some complications from having both that we will not discuss (e.g., static overloading and subtyping are more difficult to define), but there are also benefits. In addition to the obvious benefit of supporting separately the idioms that each feature supports well, we can combine the ideas to get even more code reuse and expressiveness.

The key idea is to have *bounded generic types*, where instead of just saying “a subtype of `T`” or “for all types `'a`,” we can say, “for all types `'a` that are a subtype of `T`.” Like with generics, we can then use `'a` multiple times to indicate where two things must have the same type. Like with subtyping, we can treat `'a` as a subtype of `T`, accessing whatever fields and methods we know a `T` has.

We will show an example using Java, which hopefully you can follow just by knowing that `List<Foo>` is the syntax for the type of lists holding elements of type `Foo`.

Consider this `Point` class with a `distance` method:

```
class Pt {
  double x, y;
  double distance(Pt pt) { return Math.sqrt((x-pt.x)*(x-pt.x)+(y-pt.y)*(y-pt.y)); }
  Pt(double _x, double _y) { x = _x; y = _y; }
}
```

Now consider this static method that takes a list of points `pts`, a point `center`, and a radius `radius` and returns a new list of points containing all the input points within `radius` of `center`, i.e., within the circle defined by `center` and `radius`:

```

static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
        if(pt.distance(center) <= radius)
            result.add(pt);
    return result;
}

```

(Understanding the code in the method body is not important.)

This code works perfectly fine for a `List<Pt>`, but if `ColorPt` is a subtype of `Pt` (adding a `color` field and associated methods), then we cannot call `inCircle` method above with a `List<ColorPt>` argument. Because depth subtyping is unsound with mutable fields, `List<ColorPt>` is not a subtype of `List<Pt>`. Even if it were, we would like to have a result type of `List<ColorPt>` when the argument type is `List<ColorPt>`.

For the code above, this is true: If the argument is a `List<ColorPt>`, then the result will be too, but we want a way to express that in the type system. Java's bounded polymorphism lets us describe this situation (the syntax details are not important):

```

static <T extends Pt> List<T> inCircle(List<T> pts, Pt center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
        if(pt.distance(center) <= radius)
            result.add(pt);
    return result;
}

```

This method is polymorphic in type `T`, but `T` must be a subtype of `Pt`. This subtyping is necessary so that the method body can call the `distance` method on objects of type `T`. Wonderful!

Optional: Additional Java-Specific Bounded Polymorphism

While the second version of `inCircle` above is ideal, let us now consider a few variations. First, Java does have enough dynamically checked casts that it is possible to use the first version with a `List<ColorPt>` argument and cast the result from `List<Pt>` to `List<ColorPt>`. We have to use the “raw type” `List` to do it, something like this where `cps` has type `List<ColorPt>`.

```
List<ColorPt> out = (List<ColorPt>)(List) inCircle((List<Pt>)(List)cps, new Pt(0.0,0.0), 1.5);
```

In this case, these casts turn out to be okay: if `inCircle` is passed a `List<ColorPt>` the result will be a `List<ColorPt>`. But casts like this are dangerous. Consider this variant of the method that has the same type as the initial non-generic `inCircle` method:

```

static List<Pt> inCircle(List<Pt> pts, Pt center, double radius) {
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
        if(pt.distance(center) <= radius)
            result.add(pt);
    else

```

```

        result.add(center);
    return result;
}

```

The difference is that any points not within the circle are “replaced” in the output by `center`. Now if we call `inCircle` with a `List<ColorPt>` `cps` where one of the points is not within the circle, then the result is *not* a `List<ColorPt>` — it contains a `Pt` object! You might expect then that the cast of the result to `List<ColorPt>` would fail, but Java does not work this way for backward-compatibility reasons: even this cast succeeds. So now we have a value of type `List<ColorPt>` that is not a list of `ColorPt` objects. What happens instead in Java is that a cast will fail later when we get a value from this alleged `List<ColorPt>` and try to use it as `ColorPt` when it is in fact a `Pt`. The blame is clearly in the wrong place, which is why using the warning-inducing casts in the first place is so problematic.

Last, we can discuss what type is best for the `center` argument in our bounded-polymorphic version. Above, we chose `Pt`, but we could also choose `T`:

```

static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
        if(pt.distance(center) <= radius)
            result.add(pt);
    return result;
}

```

It turns out this version allows *fewer* callers since the previous version allows, for example, a first argument of type `List<ColorPt>` and a second argument of type `Pt` (and, therefore, via subtyping, also a `ColorPt`). With the argument of type `T`, we require a `ColorPt` (or a subtype) when the first argument has type `List<ColorPt>`. On the other hand, our version that sometimes adds `center` to the output requires the argument to have type `T`:

```

static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
        if(pt.distance(center) <= radius)
            result.add(pt);
        else
            result.add(center);
    return result;
}

```

In this last version, if `center` has type `Pt`, then the call `result.add(center)` does not type-check since `Pt` may not be a subtype of `T` (what we know is `T` is a subtype of `Pt`). The actual error message may be a bit confusing: It reports there is no `add` method for `List<T>` that takes a `Pt`, which is true: the `add` method we are trying to use takes a `T`.