1 Subtyping

We will now add subtyping to MF, our made-up language, in a way that will not require us to change any of our existing SML 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 add two things to the MF type system:

- The idea of one type being a subtype of another: write $t_1 <: t_2$ to mean $t_1$ is a subtype of $t_2$.
- One new typing rule: if $e$ has type $t_1$ and $t_1 <: t_2$, then $e$ also has type $t$.

So now we just need to give rules for $t_1 <: t_2$, i.e., when is one type a subtype of another. Here are two rules that are perfectly reasonable in any language:

- **Transitivity:** If $t_1 <: t_2$ and $t_2 <: t_3$, then $t_1 <: t_3$.
- **Reflexivity:** Every type is a subtype of itself: $t <: t$.

But what about rules related to fields?

A common misconception is that when we are defining our own language, we can make the subtyping (or typing) rules whatever we want. Remember, our type system is allegedly preventing something bad from happening when programs run. If our goal is to prevent field-missing errors, then we cannot add subtyping rules that would compromise that goal.

For subtyping, the key guiding principle is **substitutability:** $t_1$ can be a subtype of $t_2$ only if anywhere a value of type $t_2$ is expected in a program, a value of type $t_1$ can be safely substituted. This principle is also called **behavioral subtyping** and the **Liskov substitution principle**. The notion of what “safely” means can vary from language to language. In MF, it means that no new run time errors could be caused by the substitution. For records, that in turn means that $t_1$ should have all the fields that $t_2$ has, and with the same types.

So let’s look at some subtyping rules for fields, and see whether they obey substitutability.

2 Width and depth subtyping

Here’s a good subtyping rule for records that obeys substitutability:

- **Width subtyping:** If $t_1$ is the same as $t_2$, except that $t_1$ perhaps adds extra fields that $t_2$ doesn’t have, then $t_1$ is a subtype of $t_2$—i.e., $t_1 <: t_2$. More precisely, if $m \geq n$, then $\{f_1:t_1,\ldots,f_m:t_m\} <: \{f_1:t_1,\ldots,f_n:t_n\}$.

According to width subtyping, a subtype can be “wider” than its supertype, because the subtype can add new fields. This is the rule we saw at the end of the last lecture. It’s also a rule that OO programmers are quite used to, because they get to define subclasses with more fields than in the superclass, and objects of the subclass are treated as subtypes of objects of the superclass.

Width subtyping doesn’t let us change the types of fields in a record, though. Consider the following example, which passes a “sphere” to a function expecting a “circle.” Note that circles and spheres have a **center** field that itself holds a record.

---

1Named after Barbara Liskov, who articulated the principle in 1987.
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}.

That subtyping relationship does not hold with our rules so far: we can drop the center field, drop the r
field, or even reorder those fields (since field order never matters in SML), but we cannot “reach into” a field
type for subtyping.

But we might want the program above to type-check—it certainly is useful and safe. So could add another
subtyping rule to handle this situation:

• **Depth subtyping:** Suppose that t1 is the same as t2, and both have a field f, except that t1 gives f
the type ta but t2 gives f the type tb. Further suppose that ta is a subtype of tb. Then t1 is a
subtype of t2—i.e., t1 <: t2. More precisely, if ta <: tb, then
  \{f1:t1,...,f:ta,...,fn:tn\} <: \{f1:t1,...,f:tb,...,fn:tn\}.

Combined with width subtyping, depth subtyping allows us to conclude
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}, so the example
above now type-checks.

Unfortunately, **depth subtyping does not obey substitutability**, so it can lead to run-time errors! 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 (* run-time error: missing field z *)

This program type-checks: 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}—we
substitute a value of the subtype where a value of the supertype is expected. But what happens when we run
this program? Well, 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. Substitutability
therefore does not hold.

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 causes runtime errors. When a (sub)typing rule is supposed to prevent
errors, but nonetheless allows them, we say that rule is **unsound**. Depth subtyping is unsound.

**Immutable fields.** Suppose we changed MF to be more like SML by making fields immutable. Then our
example above would no longer be a legal program, because setToOrigin couldn’t mutate the c fields of
its argument. It turns out that depth subtyping is sound when fields are immutable. 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.

So given the three features of (1) mutable fields, (2) depth subtyping, and (3) having a type system actually
prevent missing-field errors, you can choose any two of the three.
3 Case study: Java array subtyping

In Java, when A is a subtype of B, should an array type A[] be a subtype of B[]? The Java language designers decided the answer is “yes”. For the purpose of subtyping, Java arrays are like records whose field names are numbers, and with all fields having the same type. So Java allows depth subtyping on mutable fields! Wait... didn’t we just decide that’s unsound? Yes, we did. Consider the following Java code:

class Point { double x; double y; ... }
class ColorPoint extends Point { 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;
}

Since ColorPoint <: Point, Java’s depth subtyping on arrays says that ColorPoint[] <: Point[]. So it looks like cpt_arr[0].color will attempt to read a non-existent color field, resulting in a run-time error. Java has chosen two features (mutable fields, depth subtyping) and given up the third (having the type system prevent missing-field errors).

But that’s not the whole story. The above code won’t actually result in a missing-field error on the last line. Instead, the Java run-time gets involved, too. At run-time, assignment pt_arr[0] = new Point(3,4); will raise an ArrayStoreException if Point is not a subtype of whatever the element type of pt_arr is. Here, the element type is ColorPoint, and Point is not a subtype of ColorPoint, so an exception is raised at the assignment. The invariant being maintained by the Java run-time is that an object of type ColorPoint[] always holds objects that have type ColorPoint or a subtype, not a supertype like Point. So Java trades off one kind of run-time error for another. Why? It’s a little more flexible: code like the example above can be allowed as long as m1 is careful not to assign a supertype into the array.

But having run-time checks means that the type system is preventing fewer errors, that programs require more testing, and that extra run-time costs are incurred for performing these checks on array updates. So why was Java designed this way? Code reuse, for one reason: if you wrote a method to sort an array of Point objects, you could reuse your method to sort an array of ColorPoint objects. But the same could have been achieved without run-time checks by introducing fancier type-checking features.

By the way, all the above is true for C#, too.

4 Case study: null in Java and C#

While we are pointing out places where Java and C# choose dynamic checking over static typing rules, we should talk about how null is handled. Since null has no fields or methods (unlike nil in Ruby, null is not an object), it’s basically the empty record {}. According to width subtyping, it therefore should be a supertype of every record type, and not a subtype of anything other than itself. But Java and C# do exactly the opposite.

2For example, by using generics in combination with subtyping—see bounded polymorphism in the next lecture—or by adding support for indicating that a method will not update array elements, in which case depth subtyping is sound.
Java and C# allow `null` to be a subtype of 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 in Java you have surely encountered.

So why were Java and C# designed this way? Because there are two common situations where it is very convenient to have `null` be a subtype of everything:

- When representing a situation where you might have a value, or might not. For example, a search method might return an object, or `null` if it can’t find the object. In contrast, ML uses option types for this purpose: 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. If Java and C# added option types, they wouldn’t need `null` for this purpose—and they would possibly eliminate a lot of null-pointer exceptions.

- When initializing an object. It’s convenient to leave some fields “empty” and fill them in later. But it is also very common to have fields and variables that should never hold `null`, and it would be great to have help from the type-checker in maintaining this invariant. In fact, many proposals for incorporating “can’t be `null`” types into programming languages have been made, but none have yet caught on for Java or C#.

## 5 Function subtyping

So far we’ve considered subtyping only for records. But we can have subtyping on other values, too, including functions. **Function subtyping** involves using a function of one type in place of a function of another type. For example, if function `f` takes as argument another function `g` of type `t1->t2`, subtyping would allow us to pass a function of a different type `t3->t4` instead. But now we need to figure out how to decide when one function type is a subtype of another—that is, when `t3->t4 <: t1->t2`.

Consider the following MF example, which computes the distance between (i) the two-dimensional point `p` and (ii) the result of calling `f` with `p`:

```mf
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
  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`. Here’s a call to `distMoved` that doesn’t require any subtyping:

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

(Note that 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`.)

So, what other function types could be passed as the first argument to `distMoved` while maintaining substitutability?

---

4
Subtyping on return types. It is safe to pass in a function with a return type that “promises” more, i.e., returns a subtype of the declared return type. For example, this code is safe:

```haskell
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}`, but `distMoved` expects a `{x:real,y:real}->{x:real,y:real}`. Nonetheless, `flipGreen` is substitutable for values of that type, because the extra color field that `flipGreen` returns can’t cause a missing-field error.

The rule here is: if `ta <: tb`, then `t -> ta <: t -> tb`. That is, the subtype can have a return type that is a subtype of the supertype’s return type. We say return types are covariant, meaning that subtyping for return types works “in the same direction” (co) as for the overall function types.

Subtyping on argument types. Can we do the same thing for argument types as we did for return types? Consider this example call to `distMoved`:

```haskell
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}`. Since `{x:real,y:real,color:string} <: {x:real,y:real}`, we might be tempted to allow this program to type check. But this program would actually produce a run-time error! Expression `p.color` will result in a missing-field error, because the point passed to `flipIfGreen` does not have a color field. So `flipIfGreen` is not substitutable for a value of type `{x:real,y:real}->{x:real,y:real}`.

That means we cannot have a covariant subtyping rule for function arguments. That is, the following rule is unsound: if `ta <: tb`, then `t -> ta <: t -> tb`. This rule 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” to be provided.

Surprisingly, it actually is safe 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`:

```haskell
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}`, because the only field the argument to `flipX_Y0` needs is `x`. The call to `distMoved` therefore 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.

So subtyping for argument types is “backwards.” The sound subtyping rule is: if `tb <: ta`, then `ta -> t <: tb -> t`. We say that argument types are contravariant, meaning the subtyping for argument types works in the reverse direction (contra) as for the overall function type.

Argument contravariance might well be the least intuitive concept in this course, so it’s worth burning into your memory so that you do not forget it.

Subtyping on both argument and return types. As a final example, function subtyping can simultaneously allow contravariance of argument types and covariance of return types:

```haskell
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 \( \{x: \text{real}\} \rightarrow \{x: \text{real}, y: \text{real}, \text{color: string}\} \), which is a subtype of \( \{x: \text{real}, y: \text{real}\} \rightarrow \{x: \text{real}, y: \text{real}\} \), because \( \{x: \text{real}, y: \text{real}\} <: \{x: \text{real}\} \) (by contravariance on arguments) and \( \{x: \text{real}, y: \text{real}, \text{color: string}\} <: \{x: \text{real}, y: \text{real}\} \) (by covariance on results).

Putting it all together, the general rule for function subtyping is:

- **Function subtyping:** 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.