DEFINING SUBTYPING

We write \( T \triangleleft U \) for “\( T \) is a subtype of \( U \)”.

The intended meaning is that a value of type \( T \) may be used wherever a value of type \( U \) is needed.

We can describe valid subtyping using inference rules.

Fundamental rules:

\[
\frac{T \triangleleft T}{T \triangleleft U} \\
\frac{T \triangleleft U \ U \triangleleft W}{T \triangleleft W}
\]

Typical subtyping rules for primitive types:

\[
\text{int} \triangleleft \text{real}
\]

and others, depending on language.

RECORD SUBTYPING

Can be structural or nominal (just like type equivalence).

Basic structural rule is:

\[
R_1 \text{ has all the fields of } R_2 \text{ and maybe more} \\
\frac{}{R_1 \triangleleft R_2}
\]

(Depending on how record accesses are implemented, the extra fields in \( R_1 \) may need to be added at the end of the record to ensure safety.)

Under nominal equivalence, we require the record subtyping relation to be explicitly declared. E.g. in \texttt{fab}, given these declarations:

\[
\begin{align*}
\text{record A} \ {\{a:}\text{integer}\}} \\
\text{record B extends A} \ {\{b:}\text{boolean}\}} \\
\text{record C extends B} \ {\{c:}\text{real}\}}
\end{align*}
\]

we have \( C \triangleleft B \) and \( B \triangleleft A \).

Don’t get confused: \( B \) is a subtype of \( A \) even though a \( B \) value has more fields than an \( A \) value.

MORE STRUCTURAL SUBTYPING RULES

Pairs

Given immutable pair types \( T_1 \times T_2 \), whose values are constructed with \( (e_1, e_2) \) and dereferenced with \( e.fst \) and \( e.snd \), we have this covariant rule:

\[
\frac{T_1 \triangleleft U_1 \ T_2 \triangleleft U_2}{T_1 \times T_2 \triangleleft U_1 \times U_2}
\]

Functions

Given function types of the form \( T_1 \times T_2 \times \ldots \times T_n \rightarrow T \), we have

\[
\frac{U_1 \triangleleft T_1 \ U_2 \triangleleft T_2 \ \ldots \ U_n \triangleleft T_n}{T_1 \times T_2 \times \ldots \times T_n \rightarrow T \triangleleft U_1 \times U_2 \times \ldots \times U_n \rightarrow U}
\]

This rule is covariant on the result type but contravariant on the argument types.
FUNCTION SUBTYPING EXAMPLES
To see why the function rule is appropriate, consider the following fab code fragments (with the definitions of A, B, C above):

```plaintext
func f (g : B -> B) {
    var b0 : B = B {a = 100, b = true};
    var b1 = g (b0);
    if b1.b then ... else ...
}
```

```plaintext
func g1 (x:A) : C {
    if x.a = 0 then ... else ...;
    return C {a = 100, b = true, c = 3.14} }
```

```plaintext
func g2 (x:C) : B { if x.c > 2.71 then ... else ...; }
```

```plaintext
func g3 (x:B) : A { return A {a = 100} }
```

The call `f(g1)`, which is legal (matches the subtyping rule), works fine. The call `f(g2)`, which illegally treats the argument as covariant, fails because `f` passes a `B` (rather than a `C`) to `g2`, so the lookup `x.c` fails. The call `f(g3)`, which illegally treats the result as contravariant, fails because `g3` returns only an `A` (rather than a `B`) to `f`, so the lookup `b1.b` fails.

SUBTYPING ARRAYS (CONTINUED)
Clearly we cannot let subtyping be contravariant in `T`: if it were, the call `f(za)` would be legal, but it would fail when `f` tries to look up the `b` component.

But also, we cannot let subtyping be covariant in `T`. If it were, the sequence

```plaintext
g(zc);
if (zc[0].c > 2.71) then ... else ...
```

would be legal. But this sequence fails because `g` updates the 0'th element of `zc` to contain a `B` rather than a `C`; after the return the lookup of `zc[0].c` fails.

Note that Java actually does permit covariant subtyping of arrays. To avoid safety problems, every store into an array (of reference types) – such as the assignment in `g` – is checked at runtime to ensure that the stored value is of the same type as the array. So in this example, calling `g(zc)` would pass the type checker but generate a checked runtime error (exception).

SUBTYPING ARRAYS
For the fab array types `@T` (i.e. array of `T`), the safe structural subtyping rule is:

```
@T <: @T
```

This rule is invariant on the array element type.

To see why neither covariance nor contravariance is appropriate, consider the following fab fragments (with the definitions of A, B, C above):

```plaintext
func f (x: @B) { if x[0].b then ... }
```

```plaintext
func g (x: @B) { x[0] = B{a = 10, b = true} }
```

```plaintext
var wa = A {a = 10};
var za = @A {1 of wa};
var wc = C {a = 10, b = true, c = 3.14};
var zc = @C {1 of wc}
```