# CS321 Languages and Compiler Design I

Winter 2012

Lecture 14a



## Defining Subtyping

We write $T \lhd U$ for "$T$ is a subtype of $U$."

The intended meaning is that a value of type $T$ may be used wherever a valud of type $U$ is needed.

We can describe valid subtyping using inference rules.

Fundamental rules:

$$\frac{}{T \lhd T}$$

$$\frac{T \lhd U \quad U \lhd W}{T \lhd W}$$

Typical subtyping rules for primitive types:

$$\frac{}{\texttt{int} \lhd \texttt{real}}$$

and others, depending on language.



## Record Subtyping

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

Basic **structural** rule is:

$$\frac{R_1 \text{ has all the fields of } R_2 \text{ and maybe more}}{R_1 \lhd R_2}$$

(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 **fab**, given these declarations:

```
record A {a:integer}
record B extends A {b:boolean}
record C extends B {c:real}
```

we have $\texttt{C} \lhd \texttt{B}$ and $\texttt{B} \lhd \texttt{A}$.

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 $e.\texttt{fst}$ and $e.\texttt{snd}$, we have this **covariant** rule:

$$\frac{T_1 \lhd U_1 \quad T_2 \lhd U_2}{T_1 \times T_2 \lhd U_1 \times U_2}$$

### Functions

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

$$\frac{U_1 \lhd T_1 \quad U_2 \lhd T_2 \quad \ldots \quad U_n \lhd T_n \qquad T \lhd U}{T_1 \times T_2 \times \ldots \times T_n \rightarrow T \lhd U_1 \times U_2 \times \ldots \times U_n \rightarrow U}$$

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):

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

func g1 (x:A) : C {
   if x.a = 0 then ... else ...;
   return C {a = 100, b = true, c = 3.14} }
func g2 (x:C) : B { if x.c > 2.71 then ... else ...;   }
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

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

$$\frac{}{@T \lhd @T}$$

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):

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

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

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}
```



## 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

```
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).