COMP2212
PROGRAMMING LANGUAGE CONCEPTS

Julian Rathke and Pawel Sobocinski
Suppose that we are in a language of functions and record types.

Consider the expression \( (\text{fun } (r : \{x : \text{int}\}) \rightarrow r.x) \{x = 2, y = 3\} \).

The function, call it F, just needs a record with a field labelled x, but the argument also has a field labelled y.

Is this a problem?

These two types need to match for this rule to be a valid instantiation.

This expression is **not** well-typed. Even though it seems perfectly sensible.
OVER SPECIFYING TYPE PARAMETERS

• There are occasions where type systems forces us to be too specific.
  • e.g. example on previous slide
  • function F morally needs an argument of any type that has at least field \( x: \text{Int} \)
  • another example: a function that returns the length of a array doesn’t need to care about what type of elements are in the array.

• What we are looking for in examples such as the above is some sort of generic type, or **polymorphic** type for the type of the parameters to functions.

• Many mainstream programming languages support generics or polymorphism.

• They do so in different ways - in terms of both specification and implementation.
POLYMORPHISM

• ‘Polymorphic’ literally means “many shaped”
• A polymorphic function may be applied to many different types of data
• There are different varieties of polymorphism:
  • **Parametric polymorphism** (C++ Templates, Java Generics, OCaml)
  • **Subtype polymorphism** (Obj Oriented, C++, Java etc)
  • **Ad hoc polymorphism** (overloading of functions and methods)
• The latter, while useful, is often not so interesting as it simply boils down to clever naming schemes that include types for internal representation of functions and methods.
• The former is a very rich topic that is closely related to type inference and unification a la ML.
• This lecture is about subtype polymorphism
SUBTYPING

- We could think of types as (structured) sets and simply say that a type $T$ is a subtype of $U$ if, in their interpretations as sets $[[T]]$ is a subset of $[[U]]$.
  - This is a nice general definition but it doesn’t give us a convenient syntactic description of subtyping.
- Two other strong possibilities are evident:
  - We could look at the capabilities of the types and say that $T$ is a subtype of $U$ if every operation that can be performed on $U$ can also be performed on $T$.
    - This definition incorporates lots of structural properties of the types. E.g., pairs must be subtypes of pairs because of the projection operations.
    - This is called structural subtyping
  - We could explicitly declare what types we want to be subtypes of others and then make sure that any operations valid on a supertype are valid on the subtype.
    - This is the approach taken in object oriented languages, via inheritance. It is often called nominal subtyping.
Either way we look at it, we can extract the following property of subtyping:

- If $T$ is a subtype of $U$ then every value of type $T$ can also be considered as a value of type $U$.

This property is called **subsumption**.

It can be formalised in the following very general type rule:

$$
\begin{array}{c}
\vdash E : T \\
\vdash T <: U
\end{array}
\quad \text{TSub}
\quad \vdash E : U
$$

- this relies on the subtyping relation $T <: U$ between types.
- The same rule is used for both structural and nominal subtyping systems.

So the obvious next question is how to define the subtyping relation.

This is where the structural and nominal subtyping systems differ greatly.
• Types are distinguished by their names - even if they represent the same structure!

• e.g.: struct Foo = \{ x : int , y : int \} and struct Bar = \{ x : int , y : int \} refer to different named types.

• A function that expects a Foo value cannot accept a Bar value, even though they are structurally identical.

• Type names hide the underlying structure.

• Consider the following example:

  • type Address= \{ name : String , address : String \}
  • type Email = \{ name : String , address : String \}

• These two types are conceptually different and are intended to be used differently.

• Distinct names for the same structure allows the programmer to enforce the distinction at the type level.
NOMINAL SUBTYPING RELATION

• Need to explicitly specify the relationships between the named types.
• A common approach is to provide the subtyping relation with the declaration of the type names themselves. e.g. `type Foo subtypes Bar = { ... }`
• Or as we know from OO languages: `class Foo extends Bar { ... }`
• As a type rule this looks something like (depending on syntax):

\[
\frac{\text{type } T \text{ subtypes } U = \{ \ldots \}}{T <: U}_{\text{SUBDECL}}
\]

• Such declarations alone aren’t typically enough to define the subtyping relation
• We also ask that this relation is **reflexive** and **transitive**

\[
\frac{T <: T}_{\text{SubRefl}} \quad \frac{T <: U \quad U <: V}{T <: V}_{\text{SubTrans}}
\]

• Java adds an extra rule in that states every type is a subtype of `Object`. 
• Of course, just declaring that one type is a subtype of another may break the important subsumption property that for \( T <: U \) every value of type \( T \) can be considered as a value of type \( U \) also.

• e.g. if \( T \) is a pair type and \( U \) is a triple — which two of the three values do we take?

• Java overcomes such difficulties by only allowing subtyping between a single form of structuring - classes.

• Inheritance simply forces that every member in the supertype also exists in the subtype.
  • This approach is typical of nominal type systems.
STRUCTURAL SUBTYPING

• Relies purely on the structure of the type to define the subtyping relation.
• First, the relationship between base, or primitive types, needs to be specified.
  • For example, one could want \texttt{short <: int}, \texttt{float <: double}
• Then structure determines the rest. For example, a subtype of the pair type $\mathbf{T \times U}$
is a pair of subtypes of $\mathbf{T}$ and $\mathbf{U}$ separately.

\[
\frac{T_1 <: T_2 \quad U_1 <: U_2}{T_1 \times U_1 <: T_2 \times U_2} \quad \text{SUBPAIR}
\]

• Record types are a generalisation of pair types, and the subtype relation on records
generalises in an interesting way too.
• Record types don’t rely on syntactic positioning in the values for their indexing. This
means we can write a record with some fields missing and have it be perfectly well
formed as a record value (of another type). We can’t do the same with a tuple.
The rules for subtyping on records can be built up in stages.

First we have the generalisation of what we saw above for pairs.

This is called **depth** subtyping for records:

\[ T_i <: U_i \quad 1 \leq i \leq n \]
\[ \{l_1 : T_1, \ldots l_n : T_n\} <: \{l_1 : U_1, \ldots, l_n : U_n\} \]

Then we have the notion of **width** subtyping in which there may be extra fields in the subtype:

\[ \{l_1 : T_1, \ldots, l_n : T_n, l_{n+1} : T_{n+1}\} <: \{l_1 : T_1, \ldots l_n : T_n\} \]

And finally, we allow re-ordering of the listed fields:

\[ \sigma \text{ a permutation of } 1 \ldots n \]
\[ \{l_1 : T_1, \ldots l_n : T_n\} <: \{l_{\sigma(1)} : T_{\sigma(1)}, \ldots, l_{\sigma(n)} : T_{\sigma(n)}\} \]

Not all languages adopt all of these principles. Indeed, Java does not allow depth subtyping. Methods or fields in a subclass cannot have subtypes of that with which they are declared in the supertype.
EXAMPLE OF RECORD SUBTYPING

\[
\begin{align*}
\{ x : \text{int}, y : \text{int}, z : \text{int} \} & \ll \{ x : \text{int}, y : \text{int} \} \\
\{ x : \text{int}, y : \text{int} \} & \ll \{ x : \text{int} \} \\
\{ x : \text{int}, y : \text{int}, z : \text{int} \} & \ll \{ x : \text{int} \} \\
\{ w : \text{int} \} & \ll \{ \} \\
\{ S : \{ x : \text{int}, y : \text{int}, z : \text{int} \}, R : \{ w : \text{int} \} \} & \ll \{ S : \{ x : \text{int} \}, \ R : \{ \} \} 
\end{align*}
\]
STRUCTURAL SUBTYPING FOR VARIANTS

• For sum types $T + U$, it is intuitive that any value of type $T$ can be considered as also being of type $T + U + V$ as the value of type $T$ is still just injected into the sum, albeit into a larger sum.

• For generalised, variant, types. We have the same notions of width, depth and permutation subtyping as we do for records.

• There is however, a subtle inversion in the rule for width subtyping:

$$
\langle l_1 : T_1, \ldots l_n : T_n \rangle <: \langle l_1 : T_1, \ldots, l_n : T_n, l_{n+1} : T_{n+1} \rangle
$$

$\text{SubVarWidth}$

• We see that we can inject into a larger variant type.

• The notions of depth and permutations are identical though.

$$
T_i <: U_i \quad 1 \leq i \leq n
\langle l_1 : T_1, \ldots l_n : T_n \rangle <: \langle l_1 : U_1, \ldots, l_n : U_n \rangle
$$

$\text{SubVarDepth}$

$\sigma$ a permutation of $1 \ldots n$

$$
\langle l_1 : T_1, \ldots l_n : T_n \rangle <: \langle l_{\sigma(1)} : T_{\sigma(1)}, \ldots, l_{\sigma(n)} : T_{\sigma(n)} \rangle
$$

$\text{SubVarPerm}$
COVARIANCE AND CONTRAVARIANCE

- We have seen that in the type formers for pairs (records) and sums (variants) there is a relationship between the subtyping on the types of substructure and the subtyping of the structure itself.
  - For example, if $T <: U$ and $V <: V$ then $T \times V <: U \times V$
- Notice that the ordering between $T$ and $U$ is somehow ‘preserved’ in the latter subtyping relation.
- This preservation of order has a particular name. We say that the pair type former is a **covariant** type constructor.
- We saw above that records and variants are both covariant type formers.
- There are type formers that do not behave like this.
- Indeed, suppose we had a type former called $\text{Foo}$.
  - So that given a type $T$, then $T \text{ Foo}$ is a new type.
  - If subtyping for $\text{Foo}$ is such that: if $T <: U$ then $U \text{ Foo} <: T \text{ Foo}$ then we call $\text{Foo}$ a **contravariant** type former.
- Let’s look at real example of a contravariant type former.
FUNCTION TYPES AND CONTRAVARIANCE

- Subtyping interacts with function types in a very interesting way.
- Let’s just look at the (general) type rule:

\[
\frac{U_1 <: T_1 \quad T_2 <: U_2}{T_1 \rightarrow T_2 <: U_1 \rightarrow U_2} \text{ SUBFUN}
\]

- What we see is that the function type former is covariant in its return type and contravariant in its argument type!

- Let’s try and explain why this is:
  - Suppose we have a function \( f : T_1 \rightarrow T_2 \) then what can we do with it?
  - We can apply it to an argument \( x \), say of type \( T_1 \),
  - But every \( U_1 \) is also a \( T_1 \) so \( f \) will accept any argument of type \( U_1 \) also.
  - Now, what does \( f \) return. It returns a value of type \( T_2 \).
  - But every value of type \( T_2 \) is also of type \( U_2 \), so \( f \) returns a value of type \( U_2 \).
  - That is, \( f \) can accept any argument of type \( U_1 \) and will return a value of type \( U_2 \).
  - That is, \( f \) is also a function of type \( U_1 \rightarrow U_2 \)
OTHER STRUCTURES

• The List type former is covariant.
• This gives the simple subtyping rule

\[
T <: U \quad \Rightarrow \quad T \text{List} <: U \text{List}^\text{SubList}
\]

• The situation for Arrays is a little more complicated.
• An array is non-structural in the sense that we don’t build elements of the type using data constructors. Elements in the array can be modified.
• That is, the array must both consume data that it is given (array write) and produce data when requested (array read).
• These two operations have different variance requirements!
Arrays and Subtyping

When an array is read, the Array type should be **covariant**

\[ T <: U \text{ would imply } T[ ] <: U[ ] \text{ so I can treat a } T \text{ array as a } U \text{ array. If I fetch data from the } U \text{ array and actually get a value of } T, \text{ that is okay because I can treat this } T \text{ value as a } U \text{ value.} \]

When an array is written, the Array type should be **contravariant**

If \( T[ ] <: U[ ] \) and I write to what I think is a \( U[ ] \) (but is actually a \( T[ ] \)) then this will be fine if I write a \( U \) value and \( U <: T \) because the \( T[ ] \) can accept this \( U \) value by pretending it is a \( T \) value.

\[
\frac{T <: U \quad U <: T}{T[ ] <: U[ ]} \quad \text{SubArray}
\]

We say that arrays are an **invariant** type former. With subtyping rule:
The type rule for Arrays in Java is not the rule I showed on the previous slide.
Instead Java supports covariant array types:

\[
T <: U \\
T[ ] <: U[ ] 
\]

This may seem appealing as it certainly looks more flexible than invariant arrays.
Yes, but it has the small drawback of being unsafe!
That’s right. Java is not type safe! Due to covariant array types.
Here is proof:

```java
class A {
    public static void main(String[] args){
        B[] b = new B[1];
        A.oops(b);
    }
    static void oops(A[] a){ a[0] = new A(); }
}
class B extends A {
}
```

This code passes the Java type checker but throws a runtime type error! OOPS.
NEXT LECTURE: TYPES FOR OBJECTS