**Subtyping**

Why subtyping?
- eliminates the need to explicitly convert between elements of different types
- can be used to express program properties
- essential in OO-languages: closely related to subclass-relationship

Subtype relation:

We write $\tau <: \sigma$ to express that $\tau$ is a subtype of $\sigma$.
If $\tau <: \sigma$ then wherever a value of type $\sigma$ is required, we can use a value of type $\tau$ instead.

Two different forms of subtyping:

- **Subset Interpretation** If $\tau <: \sigma$ then every value of $\tau$ is also a value of $\sigma$
  - even integers and integers
  - empty lists and lists
- **Coercion Interpretation** If $\tau <: \sigma$ then every value of $\tau$ can be coerced to a value of type $\sigma$ in a unique way.
  - integer and floating point values (almost...)
    - 3 to 3.0
    - characters and strings
  - ‘a’ to “a”

We include the type `Float` into (type class-less) `MinHs`
- operations on `Float`: `+Float`, `*Float`, `And so on`

We want to be able to write\(^2\)
- $3 + _\text{Float} 4.0$
- $3.0 + _\text{Float} 4$

This requires that either
- the floating point operation detects at run time presence of integer arguments and converts them to floating point values, or
- the static semantic analysis detect the integer value and inserts coercion operations.

\(^2\)Note that this is different to overloading as discussed in the previous lecture

**Properties of Subtyping**

As coercion interpretation is more expressive than subset interpretation, we will discuss the coercion interpretation in more detail.

The subtype relation should be reflexive and transitive:
- **Reflexivity:** $\tau <: \tau$
- **Transitivity:** $\tau_1 <: \tau_2$, $\tau_2 <: \tau_3$\(\Rightarrow\) $\tau_1 <: \tau_3$
  - obvious for subset interpretation (reflexivity and transitivity of subset relation)
  - holds also for coercion interpretation (identity and composition of functions)
**Coherence**

We have to be careful, the subtyping relation is **coherent**
- coerced value has to be unique

**Counter example:**
If we define `Int` and `Float` to be subtypes of `String`, with the coercion yielding the string representation, then `3 : Int` can be coerced to
- "3" (by converting it directly), but also to
- "3.0" (first coercing to `3.0 : Float`)

**Subsumption**

An important property of subtyping is expressed by the rule of **subsumption**
- implicit subsumption:

\[
\Gamma \vdash e : \tau \quad \tau <: \sigma \\
\Gamma \vdash e : \sigma
\]

- rule not syntax directed
- explicit subtyping (we add a cast expression `(σ)`):

\[
\Gamma \vdash e : \tau \quad \tau <: \sigma \\
\Gamma \vdash \text{cast}(\sigma) e : \sigma
\]

- syntax directed

**Subtyping**

What is the relation between the types
- `(Int, Int)`
- `(Float, Int)`
- `(Int, Float)`
- `(Float, Float)`

**Subtyping rule for products (depth subtyping):**

\[
\tau_1 <: \sigma_1 \\
\tau_2 <: \sigma_2 \\
\langle \tau_1, \tau_2 \rangle <: \langle \sigma_1, \sigma_2 \rangle
\]

**Subtyping rule for sums:**

\[
\tau_1 <: \sigma_1 \\
\tau_2 <: \sigma_2 \\
\tau_1 + \tau_2 <: \sigma_1 + \sigma_2
\]

**Safety**

Explicit Subtyping: if \( \tau <: \sigma \) then casting a value of type \( \sigma \) to \( \tau \) must yield a value of type \( \tau \)

Implicit Subtyping: the dynamic semantics must ensure that the value of each primitive operation is defined for the values of any subtype of the expected argument types.


**Subtyping**

What is the relation between the types

- \( \text{Int} \rightarrow \text{Int} \)
- \( \text{Float} \rightarrow \text{Int} \)
- \( \text{Int} \rightarrow \text{Float} \)
- \( \text{Float} \rightarrow \text{Float} \)

---

**Variance**

Rules which specify how a type constructor interacts with subtyping are called **variance** principles.

- If a constructor preserves subtyping in a given argument position, it is called **covariant**
  - Product type constructor is covariant in both arguments
  - Sum type constructor is covariant in both arguments
- If a constructor inverts subtyping in a given argument position, it is called **contravariant**
  - Function type constructor is contravariant in first argument, and
  - Covariant in second argument
- Some constructors are **invariant**

---

Given the coercion function:

- \( \text{intToFloat} :: \text{Int} \rightarrow \text{Float} \)

Can we use them to define conversion functions of type

- \( (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Float}) \)
- \( (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Float} \rightarrow \text{Float}) \)
- \( (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Float} \rightarrow \text{Int}) \)
- \( (\text{Int} \rightarrow \text{Float}) \rightarrow (\text{Float} \rightarrow \text{Int}) \)
- \( \ldots \)