What Is Subtyping?

The subsumption principle codifies a principle of **code re-use**.

Allows you to "re-use" a value of type $\sigma$ in a $\tau$ context whenever $\sigma <: \tau$.

Subsumption is sometimes called **inheritance**, but it is best not to confuse notions.

### Subsumption

To ensure that subtyping is a pre-order we insist that the following structural rules be admissible:

$$
\frac{\sigma <: \tau}{\rho <: \sigma \quad \sigma <: \tau} \quad \frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau}
$$

Either tacitly include these rules as primitive or prove they are admissible for a given set of subtyping rules.

The subsumption rule is the fundamental definition of subtyping:

$$
\Gamma \vdash e : \sigma \quad \sigma <: \tau
$$

**NB**: the typing relation is no longer syntax-directed!

### Numeric Subtyping

Good way to get subtyping wrong is to equate it with subsetting of values.

For example, it’s tempting to postulate the subtyping relationship:

$$
\text{i32} <: \text{rat} <: \text{real}
$$

Motivated by the inclusion $\mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$.

But this is unrealistic due to nature of floating point representation!
Product Subtyping

One form of subtyping for tuples, called width subtyping, is specified by the subtyping rule:

\[ J \subseteq I \quad \text{where} \quad \prod_{i \in I} \tau_i < : \prod_{j \in J} \tau_j \]

The wider tuple is a subtype of the narrower!

- Projections from a narrow tuple apply also to a wide tuple.
- Conversely, a 9-tuple has no 10th component.

Record Subtyping

Width subtyping rule for records, where the index sets are finite sets of symbols, is:

\[ m \geq n \]

\[ \exists i : \tau_1, \ldots, \tau_m : \tau_m \rightarrow \tau_i \]

Meaning depends on whether record types are ordered (C-like) or unordered (ML-like):

- **Ordered**: can drop fields at the end of a record.
- **Unordered**: can drop fields anywhere within a record.

Sum Subtyping

What is the appropriate width subtyping rule for finite sums?

\[ \sum_{i \in I} \tau_i < : \sum_{j \in J} \tau_j \]

The smaller sum is the subtype. Why?

- An element in \( \{ i \} \) of the smaller is also an element of the larger.
- Case analysis on the supertype covers the subtype.

Product Subtyping

In the common case where the index sets are initial segments of the natural numbers, this specializes to:

\[ m \geq n \]

\[ \exists \tau_1, \ldots, \tau_n : \tau_i \rightarrow \tau_j \]

The wider tuple is a subtype of the narrower!

- Projections from a narrow tuple apply also to a wide tuple.
- Conversely, a 9-tuple has no 10th component.

Record Subtyping

Evaluation of record projection:

\[ \exists i : \tau_1, \ldots, \tau_n : \tau_n \rightarrow \tau_i \]

But how do we find the field labelled \( i \)?

- Without subtyping we can use the type of the record to predict the position of any field.
- With subtyping the type does not reveal the shape of the record; there may be many more fields than the type specifies! Options include mapping (e.g., hash) functions and coercion (immutable records only).

Sum Subtyping

In the common case where the index sets are initial segments of the natural numbers, this specializes to:

\[ m \leq n \]

\[ \exists \tau_1, \ldots, \tau_n : \tau_n \rightarrow \tau_i \]

- An element in \( \{ i \} \) of the smaller is also an element of the larger.
- Case analysis on the supertype covers the subtype.
Variance Principles

A variance principle tells how a type constructor interacts with subtyping in each position.

- **Covariance**: the constructor preserves subtyping.
- **Contravariance**: the constructor reverses subtyping.
- **Invariance**: the constructor precludes subtyping.

Product Subtyping

Subtyping rule for tuple types specifies that tuples are **covariant**:

\[
\forall i \in I. \sigma_i \ll \tau_i \Rightarrow \Pi_i \in I. \sigma_i \ll \Pi_i \tau_i
\]

Subtyping is preserved in each field of the tuple.

Called **depth subtyping** since it applies subtyping within components.

Record Subtyping

Covariant depth subtyping also applies to record types:

\[
\forall i \in I. \sigma_i \ll \tau_i \Rightarrow \left( \begin{array}{c} \sigma_i \ll \tau_i \\ \vdots \\ \sigma_n \ll \tau_n \end{array} \right) \ll \left( \begin{array}{c} \tau_i \\ \vdots \\ \tau_n \end{array} \right)
\]

Subtyping is preserved in each field of the record.

Product Variance

Why is covariance safe?

- Must check that it validates subsumption.
- What can we do with a value of type \( \langle \tau_1, \ldots, \tau_n \rangle \)?
  - Extract \( i \)th component and use it as a value of type \( \tau_i \).
- If we actually have a value of type \( \sigma_1, \ldots, \sigma_n \), the \( i \)th component is a \( \sigma_i \).
  - But we can use it as a \( \tau_i \).

Sum Subtyping

Depth subtyping for sums is also based on covariance:

\[
\forall i \in I. \sigma_i \ll \tau_i \Rightarrow \Sigma_i \in I. \sigma_i \ll \Sigma_i \tau_i
\]

Check: case analysis on the supertype.

- The \( i \)th case expects a value of type \( \tau_i \).
- By subsumption it is OK to provide a value of type \( \pi_i \).
Sum Subtyping

When specialized to symbolic labels as index sets, the covariance principle for sum types takes the following form:

\[
\sigma_d : \tau_d, \ldots, \sigma_n : \tau_n \leq \sigma'_d : \tau'_d, \ldots, \sigma'_n : \tau'_n
\]

Function Subtyping

What variance principles should apply to \( \sigma \rightarrow \tau \)?

- When is it sensible to have \( \sigma \rightarrow \tau ; \sigma' \rightarrow \tau' \)?
- What can we do with a value of the supertype? Is a value of the subtype acceptable?

Function Variance

What can we do with a value of type \( \sigma' \rightarrow \tau' \)?

- Apply it to an argument of type \( \sigma' \).
- Use the result as a value of type \( \tau' \).

Function Variance

Suppose now that \( f : \sigma \rightarrow \tau \).

- When does it make sense to apply it to a value of type \( \sigma' \)?
  Only if \( \sigma' \leq \sigma \)
- When does it make sense to use its result as a value of type \( \tau' \)?
  Only if \( \tau \leq \tau' \)

Function Variance

The function type constructor is

- **Covariant** in the range.
- **Contravariant** in the domain.

The variance rule for functions is

\[
\sigma_d \leq \sigma'_d, \sigma \leq \sigma', \tau_d \leq \tau'_d, \tau \leq \tau' \implies \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'
\]

Safety for Subtyping

Proving safety in the presence of subtyping is a bit delicate.

Subsumption means that static type only reveals partial information regarding underlying values.

So proof of preservation and progress, and statements and proofs of underlying lemmas, change accordingly.

Illustrated here by considering safety of product types in isolation.
Safety for Tuples

Lemma 1 (Structurality)
1. The tuple subtyping relation is reflexive and transitive.
2. The typing judgement $\Gamma \vdash e : \tau$ is closed under weakening and substitution.

Safety for Tuples

Lemma 2 (Inversion)
1. If $e : j : \tau$ then $e : \Pi_{j \in J} \sigma_j$ where $\sigma_j : \tau_j$ for each $j \in J$.
2. If $\sigma_i : \Pi_{i \in I} \tau_i$ where $\tau_i : \tau$ for each $i \in I$.
3. If $\sigma : \Pi_{i \in I} \tau_i$, then $\sigma = \sigma_i$ for some $I$ and some types $\tau_i$ for each $i \in I$.
4. If $\Pi_{i \in I} \sigma_i \in \Pi_{j \in J} \tau_j$, then $J \subseteq I$ and $\sigma_j : \tau_j$ for each $j \in J$.

Proof: By induction on the typing rules, taking special care with the subsumption rule.

Safety for Tuples

Theorem 3 (Preservation)
If $e : \tau$ and $e \Rightarrow e'$, then $e' : \tau$.

Proof: By induction on dynamic semantics.

Safety for Tuples

Theorem 4 (Canonical Forms)
If $e \text{ val}$ and $e : \Pi_{j \in J} \tau_j$, then $e$ is of the form $\langle \epsilon_i \rangle_{i \in I}$ where $J \subseteq I$ and $\epsilon_i : \tau_i$ for each $i \in J$.

Proof: By induction on static semantics, taking account of the definition of values. Note that the value of a tuple type is, in general, larger than is predicted by its type.
Safety for Records

Theorem 5 (Progress)
If \( e : \tau \), then either \( e \) val, or there exists \( e' \) such that \( e \mapsto e' \).

Proof: By induction on static semantics.

Summary

Subtyping supports code reuse by subsumption.

Choosing subtyping principles is tricky.

- Unsoundness.
- Potential for inefficiency.