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 $\sigma \subseteq \tau$.

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

What Is Subtyping?

To decide whether a subtype relation $\sigma \subseteq \tau$ is reasonable, we must check the subsumption principle.

- Every use of a value of type $\tau$ must make sense when given a value of type $\sigma$.
- Must consider all possible uses of values of type $\tau$.

MinML With Subtyping

We will consider the extension of MinML with subtyping.

1. Operational semantics of subtyping.
2. Type safety in the presence of subtyping.
3. Decidability of type checking.

MinML With Subtyping

Two extensions:

- A subtype relation $\sigma \subseteq \tau$, given by a set of subtyping rules.
- The subsumption rule for typing:

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

NB: the typing relation is no longer syntax-directed!
Specifying A Subtype Relation

A subtype relation is often specified in two parts:

- **Axioms** stating the fundamental forms of subtyping.
- **Variance Principles** stating how type constructors interact with subtyping.

Arithmetic Subtyping

Arithmetic subtyping axiom:

\[ \text{int} \subseteq \text{float} \]

Motivated by the inclusion \( \mathbb{Z} \subseteq \mathbb{R} \).

(There are no variance rules for atomic types.)

Safety of Arithmetic Subtyping

Assume we have floating point constants \( f \).

Assume we have (at least) floating point addition:

\[
\Gamma \vdash e_1 : \text{float}, \quad \Gamma \vdash e_2 : \text{float}
\]

\[
\Gamma \vdash e_1 + e_2 : \text{float}
\]

Is the resulting language type safe?

Arithmetic Subtyping

**Theorem 1 (Preservation)**

If \( e : \tau \) and \( e \rightarrow e' \), then \( e' : \tau \).

The proof is exactly as before, because it proceeds by induction on evaluation.

Just add new cases for floating point arithmetic, e.g., \( f_1 + f_2 \rightarrow f \), where \( f = f_1 \oplus f_2 \).

Arithmetic Subtyping

**Theorem 2 (Progress)**

If \( e : \tau \), then either \( e \) value or \( e \rightarrow e' \) for some \( e' \).

The proof is by induction on typing, so a new case must be considered!

Suppose that \( e : \tau \) because \( e : \tau' \) and \( \tau' <: \tau \).

By induction either \( e \) value or there exists \( e' \) such that \( e \rightarrow e' \).
**Arithmetic Subtyping**

Suppose that \( e_1 \cdot e_2 : \text{float} \) because \( e_1 : \text{float} \) and \( e_2 : \text{float} \).

Suppose further that \( e_1 \) value and \( e_2 \) value. Is there a value \( v \) such that \( e_1 \cdot e_2 \rightarrow v \)?

This is not obvious!

**Tuple Subtyping**

Tuple subtyping axiom:

\[
(m > n) \\
\tau_1 \ast \cdots \ast \tau_m \downarrow \tau_1 \ast \cdots \ast \tau_n
\]

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**

Record subtyping axiom:

\[
(m > n) \\
\{l_1 : \tau_1, \ldots, l_m : \tau_m\} \downarrow \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\]

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 subtyping axiom for \( n \)-ary sums?

\[
(m < n) \\
\tau_1 + \cdots + \tau_m \downarrow \tau_1 + \cdots + \tau_n
\]

The **smaller** sum is the subtype. Why?

- An element \( 1_a \cdot \langle v \rangle \) 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.

- **Co-variance**: the constructor preserves subtyping.
- **Contra-variance**: the constructor reflects subtyping.
- **Invariance**: the constructor precludes subtyping.

Tuple Variance

To allow these subtyping relationships, specify that tuples are covariant:

\[
\forall 1 \leq i \leq n \quad \sigma_i <: \tau_i
\]

Subtyping is preserved in each field of the tuple.

Tuple Variance

Without further specification, tuples are invariant.

For example,

\[\text{int}*\text{float} \not<: \text{float}*\text{float}\]

even if int <: float!

Sum Variance

Sums are also covariant:

\[
\forall 1 \leq i \leq n \quad \sigma_i <: \tau_i
\]

Check: case analysis on the supertype.

- The i\textsuperscript{th} case expects a value of type \(\tau_i\).
- By subsumption it is OK to provide a value of type \(\sigma_i\).

Function Subtyping

What variance principles should apply to \(\tau_1 \rightarrow \tau_2\)?

- When is it sensible to have

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

- What can we do with a value of the supertype? Is a value of the subtype acceptable?
Function Variance

Suppose that \( \text{int} <: \text{float} \). Which should hold? Which should not? Why?

- \( \text{int} \to \text{int} <: \text{float} \to \text{float} \)?
- \( \text{float} \to \text{float} <: \text{int} \to \text{int} \)?
- \( \text{float} \to \text{int} <: \text{int} \to \text{float} \)?
- \( \text{int} \to \text{float} <: \text{float} \to \text{int} \)?

Reference Variance

What is the appropriate variance principle for reference types? \( \text{ref}(\sigma) <: \text{ref}(\tau) \) if...

- \( \text{ref}(\sigma) <: \text{ref}(\tau) \) if ...

How can a value of type \( \text{ref}(\tau) \) be used?

- Fetch its contents, use as a value of type \( \tau \).
- Replace its contents with a value of type \( \tau \).

Function Variance

What can we do with a value of type \( \tau_1 \to \tau_2 \)?

- Apply it to an argument of type \( \tau_1 \).
- Use the result as a value of type \( \tau_2 \).

The function type constructor is

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

The variance rule for functions is

\[
\tau_1 <: \sigma_1 \rightarrow \sigma_2 \leftrightarrow \tau_2 \rightarrow \tau_3
\]
Reference Variance

Suppose that \( r \) has type \texttt{ref(int)}. If reference types were covariant,

- Then \( r \) would have type \texttt{ref(float)} too.
- So we could store \( \pi \) into \( r \).
- But the contents of \( r \) must be an integer!

Reference Variance

Suppose that \( r \) has type \texttt{ref(float)}. If reference types were contravariant,

- Then \( r \) would have type \texttt{ref(int)} too.
- Storing an integer is OK, since every integer is a float.
- But the contents of \( r \) might be \( \pi \)!

Reference Variance

Either way we lose type safety.

Conclusion: reference types are \textbf{invariant}!

Type of a reference cell does not change by subtyping!

Similar conclusions apply to

- \textbf{Mutable record} types whose fields are assignable (e.g., \texttt{struct}'s).
- \textbf{Mutable arrays} whose elements may be assigned.

Summary

Subtyping supports code reuse by \textbf{subsumption}.

Choosing subtyping principles is tricky.

- Unsoundness.
- Potential for inefficiency.