SUBTYPING

Intuitively, A is a subtype of B if any object of type A can be used in place of an object of type B.

Another way to look at it, informally, is:

"If A is a subtype of B, then every expression with type A also has type B."

-This is the subsumption property -to be formalized later.
There are two ways to view subtyping w.r.t. practical considerations:

Conversion vs. containment

Conversion: If A is a subtype of B, then their values may have different implementations, but there is a conversion (automatic) from A's to B's.

c.g. int to real in Fortran mixed-mode arithmetic
    (involves actual conversion operations)

Containment: A denotes a subset of the set of values denoted by B.

c.g. In Ada
    subtype my-int is integer range 1..10
Remember that **subtyping** is orthogonal to **inheritance**

- Subtyping is a relationship between types
- Inheritance is a relationship between implementations.
  - Simply a convenience to avoid cutting & pasting in the editor.

Unfortunately, many languages confuse them.
Formal treatment of subtyping

Start with the simply-typed lambda calculus with subtyping

\[ \lambda \rightarrow \]

\[ \lambda < \]

A signature \( \Sigma \) is a triple

\[ \Sigma = (B, \text{Sub}, C) \]

where \( B \) is a set of type constants, \( C \) a set of term constants, and \( \text{Sub} \) is a set of subtyping assertions of the form

\[ b <: b' \]

between type constants \( b, b' \in B \).

(subtyping assertions are between atomic type names)
Note that Sub will never contain assertions of the form

\[ b \ll : b_1 \rightarrow b_2 \]

for type constants \( b, b_1, \) and \( b_2 \).

Type Expressions in \( \lambda_{\ll} \):

- same as in \( \lambda \rightarrow \)

\[ \tau ::= b \mid \tau \rightarrow \tau \]
The subtype relation $\tau : \tau$ is defined by axioms and inference rules:

$$\tau : \tau \quad \text{(ref \,} \tau : \tau \text{)}$$

$$\rho : \sigma, \sigma : \tau \quad \text{(trans \,} \tau : \tau \text{)}$$

This makes $\tau : \sigma$ a preorder — a reflexive, transitive relation.

$$\rho : \tau, \tau : \rho' \quad \text{(trans \,} \tau : \tau \text{)}$$

$$\tau : \tau', \tau : \rho \rightarrow \rho'$$

is antimonotonic in its second argument! "contravariance"
In the "containment" (subset) interpretation, it does not seem intuitive, given \( \text{int} \subset \text{real} \), that the set of functions denoted by \( \text{real} \rightarrow \text{int} \) is a subset of the set of functions in \( \text{int} \rightarrow \text{int} \).


- becomes more intuitive when you read \( \text{int} \rightarrow \text{int} \) as the set of functions whose domain is at least the set of integers, and whose range is the integers.

To formalize this (since it isn't a proper set in set theory), we consider a single domain VALUE and a function apply: \( \text{VALUE} \times \text{VALUE} \rightarrow \text{VALUE} \). Then

\[
\text{A} \rightarrow \text{B} = \{ f \in \text{Value} \mid \forall x \in \text{Value}, \text{if } x \in \text{A} \text{ then apply } f(x) \in \text{B} \}
\]
Back to $\lambda \prec$:

Terms in $\lambda \prec$:
- identical to $\lambda \rightarrow$ terms, plus

$$\Gamma \vdash M : \sigma \Rightarrow \Gamma \vdash \sigma < : \tau$$

$$\Gamma \vdash M : \tau.$$ (subsumption)

- also have (var), ($\rightarrow$ intro), ($\rightarrow$ elim), and (add var).
Example: A type derivation for
\[ \Phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) \; 2 : \text{real} \]

where \(2.0 : \text{real}, \; 2 : \text{int}, \; \text{div} : \text{real} \rightarrow \text{real} \rightarrow \text{real} \)
and \(\text{int} \prec \text{real} \).

Two ways:

1) \[
\phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) : \text{real} \rightarrow \text{real} \\
| \text{(subsumption)} \\
\phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) : \text{int} \rightarrow \text{real} \\
| \rightarrow \text{elim} \\
\phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) 2 : \text{real} \\
\]

2) \[
\phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) : \text{real} \rightarrow \text{real} \\
| \text{(subsumption)} \\
\phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) : \text{int} \\
| \rightarrow \text{elim} \\
\phi \vdash (\lambda x: \text{real}. \text{div} \; x \; 2.0) 2 : \text{real} \\
\]

2 : \text{int}
Record Subtyping

The subtyping relation is defined by the signature \( \Sigma \), as in \( \lambda \Sigma \), along with the axioms and rules from \( \lambda \Sigma \), plus

\[
\tau_1 \triangleleft \rho_1, \ldots, \tau_n \triangleleft \rho_n
\]

\[
\langle l_1 : \tau_1, \ldots, l_n : \tau_n, l_{n+1} : \sigma_1, \ldots, l_{n+m} : \sigma_m \rangle \triangleleft \langle l_1 : \rho_1, \ldots, l_n : \rho_n \rangle
\]

-a subtype is obtained by adding components or restricting the type \( \rho_i \) of a component to a subtype \( \tau_i \triangleleft \rho_i \).
The containment interpretation of record subtyping can be seen if we view records as partial functions from labels to values.

- think of a record as a finite set of ordered label-value pairs.

The type `<a:int, b:bool>` denotes the set of all functions mapping label `a` to an integer and label `b` to a boolean (among other mappings).

Since the record `<a=3, b=true, c=2.7>` does map `a` to an int and `b` to a bool, it is also an element of `<a:int, b:bool>`
Typing Rules for $\lambda_{\downarrow}^{\text{record}}$ terms:

- same as $\lambda_{\downarrow}$ with

$$
\Gamma \vdash M_1 : \tau_1, \ldots, \Gamma \vdash M_n : \tau_n
$$

$$
\Gamma \vdash \langle l_1 = M_1, \ldots, l_n = M_n \rangle : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle
$$

(record intro)

$$
\Gamma \vdash M : \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle
$$

$$
\Gamma \vdash M \cdot l_i : \tau_i
$$

(record elim)
A Record Model of Objects

Since, in OOPL's, an object type may have methods over that type, the object type must be defined recursively.

- Using recursive types

eg. Type \( \text{point} = \langle x: \text{int}, y: \text{int}, 
    \text{move}: \text{int} \rightarrow \text{int} \rightarrow \text{point} \rangle \)

We can write this type in \( \lambda \)-calculus if we extend type expressions to include type variables and the recursive form \( \mu\). Forming the language \( \lambda^{\mu} \):

\[
\tau ::= \tau \rightarrow \tau | \langle e_1 : \tau_1, \ldots, e_k : \tau_k \rangle | \mu \tau
\]

Thus,

\( \text{point} \overset{\text{def}}{=} \mu \tau. \langle x: \text{int}, y: \text{int}, \text{move}: \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{point} \rangle \)
Since μ is a binding operator, it doesn't matter what name we choose for the bound type variable:

\[ \mu t. \text{true} = \text{true} \]

Also, remember the equational axiom (from a long-ago lecture):

(undef) \[ \mu t. \text{true} = [\mu t. \text{true}/x] \text{true} \]

Finally, we can use a fixpoint operator on terms to define object constructors:

\[ \text{make_point} \overset{df}{=} \text{fix } (\lambda f: \text{int} \rightarrow \text{int} \rightarrow \text{point}. \lambda x: \text{int}. \lambda y: \text{int}. \langle x = x_v, y = y_v, \text{move} = (\lambda dx: \text{int}. \lambda dy: \text{int}. +(x_v + dx, y_v + dy)) \rangle ) \]
Does this mean that fix has to be added to $\lambda Z:\text{record.}m$?

- No, it can be constructed using recursive types!
- from early lecture.

**Subtyping for recursive types**

Since objects are modeled by records, object subtyping is determined by record subtyping
- extended w/ subtyping on $\texttt{mt.}$

First, some intuitive examples:

```plaintext
type point = \langle x: \text{int}, y: \text{int}, move: \text{int} \to \text{int} \to \text{point}\rangle
```

- shorthand for $\texttt{mt.}$

```plaintext
type color_point = \langle x: \text{int}, y: \text{int}, c: \text{color},
move: \text{int} \to \text{int} \to \text{color_point}\rangle
```

Should `color_point < : point`? - Yes
What about

type eq-point = \langle x : \text{int}, y : \text{int}, \text{eq : eq-point} \rightarrow \text{bool} \rangle

type eq-col-point = \langle x : \text{int}, y : \text{int}, \text{eq : eq-col-point} \rightarrow \text{bool} \rangle

Is eq-col-point \leq eq-point?

Seems reasonable, but for this to be, it must be that

eq-col-point \rightarrow \text{bool} \leq eq-point \rightarrow \text{bool}

which is only true if

eq-point \leq eq-col-point

Since eq-point \neq eq-col-point, this clearly can't be the case.

Fix: Make eq-col-point.eq : eq-pt \rightarrow \text{bool}.

- unsatisfactory
  (see Adag95, others)