

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. containmentment

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.

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

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

e.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

$$\lambda_{<}^{\rightarrow}$$

A $\lambda_{<}^{\rightarrow}$ signature Σ is a triple

$$\Sigma = \langle B, \text{sub}, C \rangle$$

where B is a set of type constants, C a set of term constants, and 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

$$b \Leftarrow: b_1 \rightarrow b_2$$

for type constants b , b_1 , and b_2 .

Type Expressions in $\lambda_{\Leftarrow}^{\rightarrow}$:

- same as in λ^{\rightarrow}

$$\tau ::= b \mid \tau \rightarrow \tau$$

The subtype relation \leq is defined by axioms and inference rules:

$$\tau \leq \tau \quad (\text{ref } \leq)$$

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

This makes \leq a preorder
 - a reflexive, transitive relation.

$$\frac{\rho \leq \tau, \tau' \leq \rho'}{\tau \rightarrow \tau' \leq \rho \rightarrow \rho'}$$

\rightarrow is antimonotonic in its
~~second~~ ^{first} 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 $\text{apply} : \text{VALUE} \times \text{VALUE} \rightarrow \text{VALUE}$.

Then

$$A \rightarrow B = \left\{ f \in \text{Value} \mid \forall x \in \text{Value}, \text{ if } x \in A \text{ then } \text{apply } f x \in B \right\}$$

Back to $\lambda_{\leftarrow}^{\rightarrow}$:

Terms in $\lambda_{\leftarrow}^{\rightarrow}$:

- identical to λ^{\rightarrow} terms, plus

$$\frac{\Gamma \triangleright M : \sigma \quad \Sigma \vdash \sigma \leftarrow : \tau}{\Gamma \triangleright M : \tau}$$

(subsumption)

- also have (var), (\rightarrow intro), (\rightarrow elim),
and (add var).

Record Subtyping

The subtyping relation is defined by the signature Σ , as in $\lambda_{<i>i>}$, along with the axioms and rules from $\lambda_{<i>i>}$, plus

$$\frac{\tau_1 <i>i> \rho_1, \dots, \tau_n <i>i> \rho_n}{\langle l_1 : \tau_1, \dots, l_n : \tau_n, l_{n+1} : \sigma_1, \dots, l_{n+m} : \sigma_m \rangle <i>i> \langle l_1 : \rho_1, \dots, l_n : \rho_n \rangle}$$

- a subtype is obtained by adding components or restricting the type ρ_i of a component to a subtype $\tau_i <i>i> \rho_i$.

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 $\langle a:\text{int}, b:\text{bool} \rangle$ denotes the set of all functions mapping label a to an integer and label b to a boolean (among other mappings).

Since the record

$$\langle a=3, b=\text{true}, c=2.7 \rangle$$

does map a to an int and b to a bool, it is also an element of $\langle a:\text{int}, b:\text{bool} \rangle$

128
Typing Rules for $\lambda_{<:}^{\rightarrow, \text{record}}$ terms:

- same as $\lambda_{<:}^{\rightarrow}$ with

$$\Gamma \triangleright M_1 : \tau_1, \dots, \Gamma \triangleright M_n : \tau_n$$

$$\Gamma \triangleright \langle l_1 = M_1, \dots, l_n = M_n \rangle : \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle$$

(record intro)

$$\frac{\Gamma \triangleright M : \langle l_1 : \tau_1, \dots, l_n : \tau_n \rangle}{\Gamma \triangleright M.l_i : \tau_i}$$

(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 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 $\lambda_{\leq}^{\rightarrow, \text{record}}$ if we extend type expressions to include type variables and the recursive form $\mu t. \tau$, forming the language $\lambda_{\leq}^{\rightarrow, \text{record}, \mu}$

$$\tau ::= \underline{t} \mid b \mid \tau \rightarrow \tau \mid \langle l_1: \tau_1, \dots, l_k: \tau_k \rangle \mid \underline{\mu t. \tau}$$

Thus,

$$\text{point} \stackrel{\text{def}}{=} \mu t. \langle x: \text{int}, y: \text{int}, \text{move}: \text{int} \rightarrow \text{int} \rightarrow t \rangle$$

Since μ is a binding operator, it doesn't matter what name we choose for the bound type variable:

$$\alpha) \mu t. \tau = \mu s. [s/t] \tau \quad s \text{ not free in } \tau$$

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

$$\text{(unfold)} \quad \mu t. \tau = [\mu t. \tau / t] \tau$$

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

$$\text{make_point} \stackrel{\text{def}}{=} \text{fix} (\lambda f: \text{int} \rightarrow \text{int} \rightarrow \text{point}. \lambda x_v: \text{int}. \lambda y_v: \text{int}. \\ \langle x = x_v, y = y_v, \\ \text{move} = (\lambda dx: \text{int}. \lambda dy: \text{int}. \\ f(x_v + dx, y_v + dy)) \rangle$$

Does this mean that fix has to be added to $\lambda_{\leq}^{\rightarrow, \text{record}, \mu}$?

- 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 μ t. τ .

First, some intuitive examples:

type point = $\langle x:\text{int}, y:\text{int}, \text{move}:\text{int} \rightarrow \text{int} \rightarrow \text{point} \rangle$

↳ shorthand for μ definition

type color_point = $\langle x:\text{int}, y:\text{int}, c:\text{color}, \text{move}:\text{int} \rightarrow \text{int} \rightarrow \text{color_point} \rangle$

SHOULD color_point \leq : point? - yes

What about

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

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

Is $\text{eq-col-point} <: \text{eq-point}$?

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

$\text{eq-col-point} \rightarrow \text{bool} <: \text{eq-point} \rightarrow \text{bool}$

which is only true if

$\text{eq-point} <: \text{eq-col-point}$.

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

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

- unsatisfactory

(see Ada95, others)