Part III
Chapter 15: Subtyping

Subsumption

Subtype relation

Properties of subtyping and typing

Subtyping and other features

Intersection and union types

Subtyping

Motivation

With the usual typing rule for applications

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \rightarrow \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

the term

$$(\lambda \mathtt{r}{:}\{\mathtt{x}{:}\mathtt{Nat}\}.\ \mathtt{r.x})\ \{\mathtt{x}{=}0,\mathtt{y}{=}1\}$$

is *not* well typed.

Motivation

With the usual typing rule for applications

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-App)}$$

the term

$$(\lambda r{:}\{x{:}Nat\}.\ r.x)\ \{x{=}0, y{=}1\}$$

is *not* well typed.

This is silly: all we're doing is passing the function a better argument than it needs.

Subsumption

More generally: some types *are better* than others, in the sense that a value of one can always safely be used where a value of the other is expected.

We can formalize this intuition by introducing:

Subsumption

More generally: some types *are better* than others, in the sense that a value of one can always safely be used where a value of the other is expected.

We can formalize this intuition by introducing:

1. a *subtyping* relation between types, written $S <: T$

Subsumption

More generally: some types *are better* than others, in the sense that a value of one can always safely be used where a value of the other is expected.

We can formalize this intuition by introducing:

1. a *subtyping* relation between types, written $S <: T$
2. a rule of *subsumption* stating that, if $S <: T$, then any value of type S can also be regarded as having type T

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \qquad \text{(T-SUB)}$$

Subsumption

More generally: some types *are better* than others, in the sense that a value of one can always safely be used where a value of the other is expected.

We can formalize this intuition by introducing:

1. a *subtyping* relation between types, written $S <: T$
2. a *rule of subsumption* stating that, if $S <: T$, then any value of type S can also be regarded as having type T

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \qquad \text{(T-SUB)}$$

Principle of safe substitution

Subtyping

Intuitions: $S <: T$ means...

“An element of S may safely be used wherever an element of T is expected.” (Official.)

Subtyping

Intuitions: $S <: T$ means...

"An element of S may safely be used wherever an element of T is expected." (Official.)

- S is "better than" T
- S is a subset of T
- S is more informative / richer than T.

Example

We will define subtyping between record types so that, for example

$$\{x: Nat, y: Nat\} <: \{x: Nat\}$$

So, by subsumption,

$$\vdash \{x = 0, y = 1\} : \{x: Nat\}$$

Example

We will define subtyping between record types so that, for example

$$\{x\colon Nat, y\colon Nat\} \ <\colon \ \{x\colon Nat\}$$

So, by subsumption,

$$\vdash \ \{x = 0, y = 1\} : \ \{x\colon Nat\}$$

and hence

$$(\lambda r\colon \{x\colon Nat\}.\, r.x)\ \{x = 0, y = 1\}$$

is *well* typed.

The Subtype Relation: Records

"Width subtyping" (forgetting fields on the right):

$$\{l_i : T_i{}^{i \in 1..n+k}\} \ <: \ \{l_i : T_i{}^{i \in 1..n}\} \qquad \text{(S-RcdWidth)}$$

Intuition: $\{x\colon Nat\}$ is the type of all records with *at least* a numeric x field.

The Subtype Relation: Records

"Width subtyping" (forgetting fields on the right):

$$\{l_i : T_i^{\,i\in 1..n+k}\} <: \{l_i : T_i^{\,i\in 1..n}\} \qquad \text{(S-RcdWidth)}$$

Intuition: $\{x\text{: Nat}\}$ is the type of all records with *at least* a numeric x field.

Note that the record type with *more* fields is a *subtype* of the record type with fewer fields.

Reason: the type with more fields places a *stronger constraint* on values, so it describes *fewer values*.

The Subtype Relation: Records

Permutation of fields:

$$\frac{\{\mathtt{k}_j : \mathtt{S}_j \;^{j \in 1..n}\} \text{ is a permutation of } \{\mathtt{l}_i : \mathtt{T}_i \;^{i \in 1..n}\}}{\{\mathtt{k}_j : \mathtt{S}_j \;^{j \in 1..n}\} <: \{\mathtt{l}_i : \mathtt{T}_i \;^{i \in 1..n}\}} \quad \text{(S-RcdPerm)}$$

Order of fields in Records

The order of fields in a record does *not make any difference* to *how we can safely use it*, since the only thing that we can do with records (projecting their fields) is *insensitive* to the order of fields.

Order of fields in Records

The order of fields in a record does *not make any difference* to *how we can safely use it*, since the only thing that we can do with records (projecting their fields) is *insensitive* to the order of fields.

S-RcdPerm tells us that

{c:Top, b: Bool, a: Nat} <: {a: Nat, b: Bool, c:Top}

and

{a: Nat, b: Bool, c:Top} <: {c:Top, b: Bool, a: Nat}

The Subtype Relation: Records

Permutation of fields:

$$\frac{\{k_j : S_j\ ^{j \in 1..n}\} \text{ is a permutation of } \{l_i : T_i\ ^{i \in 1..n}\}}{\{k_j : S_j\ ^{j \in 1..n}\} <: \{l_i : T_i\ ^{i \in 1..n}\}} \quad \text{(S-RcdPerm)}$$

By using S-RcdPerm together with S-RcdWidth and S-Trans allows us to *drop arbitrary fields* within records.

The Subtype Relation: Records

"Depth subtyping" within fields:

$$\frac{\text{for each } i \quad \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:}\mathtt{S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:}\mathtt{T}_i\ ^{i\in 1..n}\}} \quad \text{(S-RcdDepth)}$$

The types of individual fields may change, as long as the type of each corresponding field in the two records are in the subtype relation.

Example

$$\dfrac{\dfrac{}{\texttt{\{a:Nat,b:Nat\}} <: \texttt{\{a:Nat\}}}\ \textsc{S-RcdWidth} \qquad \dfrac{}{\texttt{\{m:Nat\}} <: \texttt{\{\}}}\ \textsc{S-RcdWidth}}{\texttt{\{x:\{a:Nat,b:Nat\},y:\{m:Nat\}\}} <: \texttt{\{x:\{a:Nat\},y:\{\}\}}}\ \textsc{S-RcdDepth}$$

Example

We can also use S-RcdDepth to refine the type of just a single record field (instead of refining every field), by using S-RefL to obtain trivial subtyping derivations for other fields.

$$\dfrac{\dfrac{}{\{a: Nat, b: Nat\} <: \{a: Nat\}}\ S-RCDWIDTH \quad \dfrac{}{\{m: Nat\} <: \{m: Nat\}}\ S-REFL}{\{x: \{a: Nat, b: Nat\}, y: \{m: Nat\}\} <: \{x: \{a: Nat\}, y: \{m: Nat\}\}}\ \mathrm{S-RcdDepth}$$

Variations

Real languages often choose not to adopt *all of these record subtyping rules*. For example, in Java,

- A subclass may not change the argument or result types of a method of its superclass (i.e., *no depth subtyping*)
- Each class has just one superclass ("*single inheritance*" of classes)

  each class member (field or method) can be assigned a single index, adding new indices "on the right" as more members are added in subclasses (i.e., no permutation for classes)

- A class may implement multiple interfaces ("*multiple inheritance*" of interfaces)

  I.e., permutation is allowed for interfaces.

The Subtype Relation: Arrow types

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \qquad \text{(S-ARROW)}$$

The Subtype Relation: Arrow types

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-ARROW)}$$

Note the *order* of T_1 and S_1 in the first premise. The subtype relation is *contravariant* in the left-hand sides of arrows and *covariant* in the right-hand sides.

The Subtype Relation: Arrow types

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-Arrow)}$$

Note the *order* of T_1 and S_1 in the first premise. The subtype relation is *contravariant* in the left-hand sides of arrows and *covariant* in the right-hand sides.

Intuition: if we have a function f of type $S_1 \longrightarrow S_2$, then we know that f accepts elements of type S_1; clearly, f will also accept elements of any subtype T_1 of S_1. The type of f also tells us that it returns elements of type S_2; we can also view these results belonging to any supertype T_2 of S_2. That is, any function f of type $S_1 \longrightarrow S_2$ can also be viewed as having type $T_1 \longrightarrow T_2$.

The Subtype Relation: Top

It is *convenient* to have a type that is a *supertype of every type*. We introduce a new type constant Top, plus a rule that makes Top a maximum element of the subtype relation.

$$S <: \mathtt{Top} \qquad \text{(S-TOP)}$$

The Subtype Relation: Top

It is *convenient* to have a type that is a *supertype of every type*. We introduce a new type constant Top, plus a rule that makes Top a maximum element of the subtype relation.

$$S <: Top \qquad \text{(S-TOP)}$$

Cf. Object in Java.

Subtype Relation: General rules

$$\texttt{S <: S} \qquad \text{(S-REFL)}$$

$$\frac{\texttt{S <: U} \qquad \texttt{U <: T}}{\texttt{S <: T}} \qquad \text{(S-TRANS)}$$

Subtype Relation

$$\mathtt{S} <: \mathtt{S} \qquad \text{(S-Refl)}$$

$$\frac{\mathtt{S} <: \mathtt{U} \qquad \mathtt{U} <: \mathtt{T}}{\mathtt{S} <: \mathtt{T}} \qquad \text{(S-Trans)}$$

$$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n+k}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{\text{for each } i \qquad \mathtt{S}_i <: \mathtt{T}_i}{\{\mathtt{l}_i\mathtt{:S}_i\ ^{i\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdDepth)}$$

$$\frac{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} \text{ is a permutation of } \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\{\mathtt{k}_j\mathtt{:S}_j\ ^{j\in 1..n}\} <: \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(S-RcdPerm)}$$

$$\frac{\mathtt{T}_1 <: \mathtt{S}_1 \qquad \mathtt{S}_2 <: \mathtt{T}_2}{\mathtt{S}_1 \rightarrow \mathtt{S}_2 <: \mathtt{T}_1 \rightarrow \mathtt{T}_2} \qquad \text{(S-Arrow)}$$

$$\mathtt{S} <: \mathtt{Top} \qquad \text{(S-Top)}$$

Properties of Subtyping

Safety

Statements of progress and preservation theorems are unchanged from $\lambda_{\rightarrow}$.

Safety

Statements of progress and preservation theorems are unchanged from $\lambda_{\to}$.

Proofs become a bit more involved, because the typing relation is no longer *syntax directed*.

Given a derivation, we don't always know what rule was used in the last step. The rule T-SUB could appear anywhere.

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

Syntax-directed rules

When we say a set of rules is syntax-directed we mean two things:

1. There is exactly one rule in the set that applies to each syntactic form. (We can tell by the syntax of a term which rule to use.)
   - In order to derive a type for t_1 t_2, we must use T-App.
2. We don't have to “guess" an input (or output) for any rule.
   - To derive a type for t_1 t_2, we need to derive a type for t_1 and a type for t_2.

Preservation

Theorem: If $\Gamma \vdash \mathrm{t}: \mathrm{T}$ *and* $\mathrm{t} \longrightarrow \mathrm{t}'$, *then* $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

Proof: By induction on *typing derivations*.

Which cases are likely to be hard?

Subsumption case

Case T-Sub: $t : S \quad S <: T$

By the induction hypothesis, $\Gamma \vdash t' : S$. By T-Sub , $\Gamma \vdash t$.

Not hard!

Application case

Case T-APP :

$$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$$

By the inversion lemma for evaluation, there are three rules by which $t \longrightarrow t'$ can be derived: E-APP1, E-APP2, and E-APPABS . Proceed by cases.

Application case

Case T-APP :

$$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$$

By the inversion lemma for evaluation, there are three rules by which $t \longrightarrow t'$ can be derived: E-APP1, E-APP2, and E-APPABS . Proceed by cases.

Subcase E-APP1: $t_1 \longrightarrow t'_1$ $\qquad t' = t'_1\ t_2$

The result follows from the induction hypothesis and T-APP .

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

Application case

Case T-APP :

$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$

Subcase E-APP2 : $t_1 = v_1 \quad t_2 \longrightarrow t'_2 \quad t' = v_1\ t'_2$

Similar.

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{(T-APP)}$$

$$\frac{t_2 \longrightarrow t'_2}{v_1\ t_2 \longrightarrow v_1\ t'_2} \qquad \text{(E-APP2)}$$

Application case

Case T-APP:

$$t = t_1\ t_2 \quad \Gamma \vdash t_1 : T_{11} \longrightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12}$$

Subcase E-APPABS :

$$t_1 = \lambda x{:}S_{11}.t_{12} \quad t_2 = v_2 \quad t' = [x \mapsto v_2]\, t_{12}$$

By the inversion lemma for the typing relation...

$T_{11} <: S_{11}$ and $\Gamma, x{:}S_{11} \vdash t_{12} : T_{12}$.

By T-Sub, $\Gamma \vdash t_2 : S_{11}$.

By the substitution lemma, $\Gamma \vdash t' : T_{12}$.

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

Inversion Lemma for Typing

Lemma: If $\Gamma \vdash \lambda x{:}S_1.s_2 : T_1 \longrightarrow T_2$, then $T_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : T_2$.

Inversion Lemma for Typing

Lemma: If $\Gamma \vdash \lambda x{:}S_1.s_2 : T_1 \longrightarrow T_2$, then $T_1 <: S_1$ and $\Gamma, x{:}S_1 \vdash s_2 : T_2$.

Proof: Induction on typing derivations.

Case T–Sub: $\lambda x{:}S_1.s_2 : U$ $\quad$ $U <: T_1 \longrightarrow T_2$

We want to say "By the induction hypothesis...", but the IH does not apply (we do not know that U is an arrow type). Need another lemma...

Lemma: If $U <: T_1 \longrightarrow T_2$, then U has the form: $U_1 \longrightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$. (Proof: by induction on subtyping derivations.)

Inversion Lemma for Typing

By this lemma, we know $U = U_1 \longrightarrow U_2$, with $T_1 <: U_1$ and $U_2 <: T_2$.

The IH now applies, yielding $U_1 <: S_1$ and $\Gamma, x: S_1 \vdash s_2: U_2$.

From $U_1 <: S_1$ and $T_1 <: U_1$, rule S-Trans gives $T_1 <: S_1$.

From $\Gamma, x: S_1 \vdash s_2: U_2$ and $U_2 <: T_2$, rule T-Sub gives $\Gamma, x: S_1 \vdash s_2: T_2$, and we are done

Subtyping with Other Features

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \qquad \text{(T-Ascribe)}$$

$$v_1 \text{ as } T \longrightarrow v_1 \qquad \text{(E-Ascribe)}$$

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \quad \text{(T-Ascribe)}$$

$$v_1 \text{ as } T \longrightarrow v_1 \quad \text{(E-Ascribe)}$$

Casting (cf. Java):

$$\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \quad \text{(T-Cast)}$$

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \longrightarrow v_1} \quad \text{(E-Cast)}$$

Subtyping and Variants

$$\texttt{<}\mathtt{l}_i : \mathtt{T}_i \; ^{i \in 1..n}\texttt{>} \quad <: \quad \texttt{<}\mathtt{l}_i : \mathtt{T}_i \; ^{i \in 1..n+k}\texttt{>} \qquad \text{(S-VariantWidth)}$$

$$\frac{\text{for each } i \quad \mathtt{S}_i <: \mathtt{T}_i}{\texttt{<}\mathtt{l}_i : \mathtt{S}_i \; ^{i \in 1..n}\texttt{>} \quad <: \quad \texttt{<}\mathtt{l}_i : \mathtt{T}_i \; ^{i \in 1..n}\texttt{>}} \qquad \text{(S-VariantDepth)}$$

$$\frac{\texttt{<}\mathtt{k}_j : \mathtt{S}_j \; ^{j \in 1..n}\texttt{>} \text{ is a permutation of } \texttt{<}\mathtt{l}_i : \mathtt{T}_i \; ^{i \in 1..n}\texttt{>}}{\texttt{<}\mathtt{k}_j : \mathtt{S}_j \; ^{j \in 1..n}\texttt{>} \quad <: \quad \texttt{<}\mathtt{l}_i : \mathtt{T}_i \; ^{i \in 1..n}\texttt{>}} \qquad \text{(S-VariantPerm)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \texttt{<}\mathtt{l}_1\texttt{=}\mathtt{t}_1\texttt{>} : \texttt{<}\mathtt{l}_1 : \mathtt{T}_1\texttt{>}} \qquad \text{(T-Variant)}$$

Subtyping and Lists

$$\frac{\mathtt{S_1} <: \mathtt{T_1}}{\mathtt{List}\ \mathtt{S_1} <: \mathtt{List}\ \mathtt{T_1}} \qquad (\text{S-List})$$

i.e., List is a covariant type constructor.

Subtyping and References

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \quad \text{(S-REF)}$$

i.e., Ref is not a *covariant* (nor a *contravariant*) type constructor.

Subtyping and References

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \quad \text{(S-REF)}$$

i.e., Ref is not a *covariant* (nor a *contravariant*) type constructor.

Why?

- When a reference is *read*, the context expects a T_1, so if $S_1 <: T_1$ then an S_1 is ok.

Subtyping and References

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \qquad \text{(S-REF)}$$

i.e., Ref is not a *covariant* (nor a *contravariant*) type constructor.

Why?

- When a reference is *read*, the context expects a T_1, so if $S_1 <: T_1$ then an S_1 is ok.
- When a reference is *written*, the context provides a T_1 and if the actual type of the reference is Ref S_1, someone else may use the T_1 as an S_1. So we need $T_1 <: S_1$.

Subtyping and Arrays

Similarly...

$$\frac{S_1 <: T_1 \qquad T_1 <: S_1}{\texttt{Array } S_1 <: \texttt{Array } T_1} \qquad \text{(S-ARRAY)}$$

$$\frac{S_1 <: T_1}{\texttt{Array } S_1 <: \texttt{Array } T_1} \qquad \text{(S-ARRAYJAVA)}$$

This is regarded (even by the Java designers) as a mistake in the design.

References again

Observation: a value of type Ref T can be used in two different ways: as a *source* for values of type T and as a *sink* for values of type T.

References again

Observation: a value of type Ref T can be used in two different ways: as a *source* for values of type T and as a *sink* for values of type T.

Idea： Split Ref T into three parts:

- Source T: reference cell with “read capability”
- Sink T: reference cell with “write capability”
- Ref T: cell with both capabilities

Modified Typing Rules

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Source}\ \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{!t}_1 : \mathtt{T}_{11}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \mathtt{t}_1 : \mathtt{Sink}\ \mathtt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \mid \Sigma \vdash \mathtt{t}_1 \mathtt{:=} \mathtt{t}_2 : \mathtt{Unit}} \quad \text{(T-Assign)}$$

Subtyping rules

$$\frac{\texttt{S}_1 <: \texttt{T}_1}{\texttt{Source S}_1 <: \texttt{Source T}_1} \quad \text{(S-SOURCE)}$$

$$\frac{\texttt{T}_1 <: \texttt{S}_1}{\texttt{Sink S}_1 <: \texttt{Sink T}_1} \quad \text{(S-SINK)}$$

$$\texttt{Ref T}_1 <: \texttt{Source T}_1 \quad \text{(S-REFSOURCE)}$$

$$\texttt{Ref T}_1 <: \texttt{Sink T}_1 \quad \text{(S-REFSINK)}$$

Capabilities

Other kinds of capabilities (e.g., send and receive capabilities on communication channels, encrypt/decrypt capabilities of cryptographic keys, ...) can be treated similarly.

Intersection and Union Types

Intersection Types

The inhabitants of $T_1 \wedge T_2$ are terms belonging to *both* S and T —i.e., $T_1 \wedge T_2$ is an order-theoretic meet (greatest lower bound) of T_1 and T_2.

$$T_1 \wedge T_2 <: T_1 \qquad \text{(S-INTER1)}$$

$$T_1 \wedge T_2 <: T_2 \qquad \text{(S-INTER2)}$$

$$\frac{S <: T_1 \qquad S <: T_2}{S <: T_1 \wedge T_2} \qquad \text{(S-INTER3)}$$

$$S \to T_1 \wedge S \to T_2 <: S \to (T_1 \wedge T_2) \qquad \text{(S-INTER4)}$$

Intersection Types

Intersection types permit a very flexible form of finitary overloading.

$+ : (\mathtt{Nat} \rightarrow \mathtt{Nat} \rightarrow \mathtt{Nat}) \wedge (\mathtt{Float} \rightarrow \mathtt{Float} \rightarrow \mathtt{Float})$

This form of overloading is extremely powerful.

Every strongly normalizing untyped lambda-term can be typed in the simply typed lambda-calculus with intersection types.

type reconstruction problem is undecidable

Intersection types have not been used much in language designs (too powerful!), but are being intensively investigated as type systems for intermediate languages in highly optimizing compilers (cf. Church project).

Union types

Union types are also useful.

$T_1 \vee T_2$ is an untagged (non-disjoint) union of T_1 and T_2.

No tags : no *case* construct. The only operations we can safely perform on elements of $T_1 \vee T_2$ are ones that make sense for both T_1 and T_2.

N.b: untagged union types in C are a source of type safety violations precisely because they ignores this restriction, allowing any operation on an element of $T_1 \vee T_2$ that makes sense for either T_1 or T_2.

Union types are being used recently in type systems for XML processing languages (cf. Xduce, Xtatic).

Varieties of Polymorphism

- Parametric polymorphism (ML-style)
- Subtype polymorphism (OO-style)
- Ad-hoc polymorphism (overloading)

Homework☺

- Read chapter 14 & 15
- Read and chew over the codes of chap 17.

- HW: 14.3.1, 15.5.2