编程语言的设计原理

Design Principles of Programming Languages

Haiyan Zhao, Di Wang

赵海燕，王迪

Peking University, Spring Term 2024

Part III

Chap 15: Subtyping

Subsumption

Subtype relation

Properties of subtyping and typing

Subtyping and other features

Intersection and union types

Subtyping

Recap for subtyping

$\rightarrow$ {} *Extends $\lambda_\rightarrow$ (9-1)*

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	*record*
		$\mathtt{t.l}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$	*record value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	*type of records*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \quad \text{(E-ProjRcd)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.l} \longrightarrow \mathtt{t}_1'\mathtt{.l}} \quad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1},\mathtt{l}_j\mathtt{=t}_j,\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\} \\ \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1},\mathtt{l}_j\mathtt{=t}_j',\mathtt{l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\end{array}} \quad \text{(E-Rcd)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \quad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1\mathtt{.l}_j : \mathtt{T}_j} \quad \text{(T-Proj)}$$

$\rightarrow$ {} <: *Extends $\lambda_{<:}$ (15-1) and simple record rules (15-2)*

New subtyping rules $\boxed{\mathtt{S} <: \mathtt{T}}$

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

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

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

Recap for subtyping

$\rightarrow$ $<:$ Top *Based on $\lambda_{\rightarrow}$ (9-1)*

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
v	::=		*values:*
		λx:T.t	*abstraction value*
T	::=		*types:*
		Top	*maximum type*
		T$\rightarrow$T	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		Γ, x:T	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

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

Subtyping $\boxed{S <: T}$

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

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

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

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

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(T-Abs)}$$

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

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

Subtype Relation: General rules

A subtyping is *a binary relation* between *types* that is closed under the following rules

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

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

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

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 $\lambda_{\rightarrow}$.

However, 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*.

e.g., the rule T-SUB could appear anywhere

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

Aside: 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.)
   – *e.g.,* 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.
   – *e.g.,* To derive a type for $t_1\ t_2$, we need to derive a type for t_1 and a type for t_2.

An *Inversion Lemma* for subtyping

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.

Case S-Arrow: $U = U_1 \longrightarrow U_2 \qquad T_1 <: U_1, U_2 <: T_2$

Immediate.

Case S-Refl: $U = T_1 \longrightarrow T_2$

- By S-Refl (twice), $T_1 <: T_1$ and $T_2 <: T_2$, as required.

Case S-Trans: $U <: W \qquad W <: T_1 \longrightarrow T_2$

- Applying the IH to the second subderivation, we find that W has the form $W_1 \longrightarrow W_2$, with $T_1 <: W_1$ and $W_2 <: T_2$.
- Now the IH applies again (to the first subderivation), telling us that U has the form $U_1 \longrightarrow U_2$, with $W_1 <: U_1$ and $U_2 <: W_2$.
- By S-Trans, $T_1 <: U_1$, and, by S-Trans again, $U_2 <: T_2$, as required.

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-Abs: $T_1 = S_1$, $T_2 = S_2$ $\Gamma, x{:}S_1 \vdash s_2 : S_2$

Case T-Sub: $\Gamma \vdash \lambda x{:}S_1.\, s_2 : U$ $U : T_1 \longrightarrow T_2$

- By the subtyping inversion lemma, U has the form of $U_1 \rightarrow U_2$, 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 $\Gamma, x{:}S_1 \vdash s_2 : T_2$, thus we are done

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* ?

Preservation - Subsumption case

Case T-Sub: $\mathrm{t} : \mathrm{S} \quad \mathrm{S} <: \mathrm{T}$

By the induction hypothesis, $\Gamma \vdash \mathrm{t}' : \mathrm{S}$.

By T-Sub, $\Gamma \vdash \mathrm{t}' : \mathrm{T}$.

Not hard!

Preservation - Application case

Case T-App :

$$\mathrm{t} = \mathrm{t}_1\ \mathrm{t}_2 \quad \Gamma \vdash \mathrm{t}_1 : \mathrm{T}_{11} \longrightarrow \mathrm{T}_{12} \quad \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_{11} \quad \mathrm{T} = \mathrm{T}_{12}$$

By the inversion lemma for evaluation, there are

three rules

by which $\mathrm{t} \longrightarrow \mathrm{t}'$ can be derived:

E-App1, E-App2, and E-AppAbs.

Proceed by cases

Preservation - 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-App1 : $t_1 \longrightarrow t'_1 \qquad t' = t'_1\ t_2$

The result follows from **the induction hypothesis** and T-App

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

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

Preservation - 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.

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

$$\frac{t_2 \longrightarrow t_2'}{v_1 \; t_2 \longrightarrow v_1 \; t_2'} \qquad \text{(E-App2)}$$

Preservation - Application case

Case T-App :

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

Subcase E-AppAbs :

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

by the *inversion lemma* for the typing relation ...

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

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

by the *substitution lemma*, $\Gamma \vdash t' : T_{12}$

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

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

Progress

Lemma for Canonical Forms

1. If v is a closed value of type $T_1 \longrightarrow T_2$, then v has the form $\lambda x{:}S_1.t_2$.
2. If v is a closed value of type $\{l_i{:}T_i^{\,i\in 1..n}\}$, then v has the form $\{k_j = v_j^{\,j\in 1..m}\}$ with $\{l_i^{\,i\in 1..n}\} \subseteq \{k_a^{\,a\in 1..m}\}$

- *Possible shapes of values* belonging to *arrow* and *record* types.
- Based on this *Canonical Forms Lemma*, we can still has the progress theorem and its proof quite close to that in the simply typed lambda-calculus

Subtyping with Other Features

Ascription and Casting

Ordinary ascription:

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

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

(T) T

up-cast

down-cast

Ascription and Casting

Ordinary ascription:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}}{\Gamma \vdash \mathtt{t}_1 \text{ as } \mathtt{T} : \mathtt{T}} \qquad \text{(T-Ascribe)}$$

$$\mathtt{v}_1 \text{ as } \mathtt{T} \longrightarrow \mathtt{v}_1 \qquad \text{(E-Ascribe)}$$

Casting (cf. Java):

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1 \text{ as } \mathtt{T} : \mathtt{T}} \qquad \text{(T-Cast)}$$

$$\boxed{\frac{\vdash \mathtt{v}_1 : \mathtt{T}}{\mathtt{v}_1 \text{ as } \mathtt{T} \longrightarrow \mathtt{v}_1} \qquad \text{(E-Cast)}}$$

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

List is a covariant type constructor

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

Subtyping and References

Ref is *not a covariant* (nor *a contravariant*) type constructor, but an *invariant*

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

Subtyping and References

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

Why?

- When a reference is *read*, the context expects a T_1, so if $\mathrm{S}_1 <: \mathrm{T}_1$ 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 $\mathrm{Ref}\ \mathrm{S}_1$, someone else may use the T_1 as an S_1. So we need $\mathrm{T}_1 <: \mathrm{S}_1$.

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

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

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

$$\mathtt{Ref\ T_1 <: Source\ T_1} \qquad \text{(S-RefSource)}$$

$$\mathtt{Ref\ T_1 <: Sink\ T_1} \qquad \text{(S-RefSink)}$$

Subtyping and Arrays

Similarly...

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

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

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

Capabilities

Other kinds of capabilities can be treated similarly, e.g.,

– *send* and *receive* capabilities on communication channels

– *encrypt*/*decrypt* capabilities of cryptographic keys

– ...

Base Types

For language with a rich set of base types, it's better to introduce primitive subtype relations among them

- e.g., Bool <: Nat
- Bool， 0/1

Intersection and Union Types

Intersection Types

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

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

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

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

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

Intersection Types

Intersection types permit a very *flexible form* of *finitary overloading*.

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

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 (a term is typable iff its evaluation terminates)

type reconstruction problem is undecidable (cf. ch22)

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.

Note well: 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)

HW for Chap15

- 15.2.2
- 15.3.2
- 15.5.2