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

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

Is the term

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

right?

It is *not* well typed.
Motivation

With the usual typing rule for applications

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

the term

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

is not well typed.

This is silly: what we’re doing is passing the function a better argument than it needs.
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 \( T \), i.e.,

\[
\Gamma \vdash t : S \quad S <: T \\
\overline{\Gamma \vdash t : T} \quad (T\text{-SUB})
\]
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 $T$, i.e.,

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

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

Back to the example, we will define subtyping between record types so that, for example

\[
\{ x : \text{Nat}, y : \text{Nat} \} \ <: \ { x : \text{Nat} } \\
\]

by subsumption,

\[
\vdash \{ x = 0, y = 1 \} : \{ x : \text{Nat} \}
\]
Example

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

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

by subsumption,

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

and hence

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

is well typed.
The Subtype Relation: Records

“Width subtyping” (forgetting fields on the right):

\[ \{ l_i : T_i^{1..n+k} \} <: \{ l_i : T_i^{1..n} \} \] (S-RcdWidth)

Intuition:

\{ x : Nat \} is the type of all records with \textit{at least} a numeric \( x \) field.
The Subtype Relation: Records

“Width subtyping” (forgetting fields on the right):

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

Intuition:

\{x: Nat\} is the type of all records with \textit{at least} a numeric \(x\) field.

Note that the record type with \textit{more} fields is a \textit{subtype} of the record type with \textit{fewer} fields.

Reason: the type with more fields places \textit{stronger constraints} on values, so it describes \textit{fewer values}. 
The Subtype Relation: Records

“Depth subtyping” within fields:

\[
\text{for each } i \quad S_i \ll T_i \\
\{1_i:S_i \mid i \in 1..n\} \ll \{1_i:T_i \mid i \in 1..n\} \quad (S-RcdDepth)
\]

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.
Examples

\[
\{a:\text{Nat}, b:\text{Nat}\} <: \{a:\text{Nat}\} \\
\{m:\text{Nat}\} <: \{\}\ \\
\{x:\{a:\text{Nat}, b:\text{Nat}\}, y:\{m:\text{Nat}\}\} <: \{x:\{a:\text{Nat}\}, y:\{\}\}\}
\]
Examples

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.

\[
\begin{align*}
\{a: \text{Nat}, b: \text{Nat}\} &\leq \{a: \text{Nat}\} & \text{S - RCDWIDTH} \\
\{x: \{a: \text{Nat}, b: \text{Nat}\}, y: \{m: \text{Nat}\}\} &\leq \{x: \{a: \text{Nat}\}, y: \{m: \text{Nat}\}\} & \text{S - RcdDepth}
\end{align*}
\]
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:

\[
\{k_j : S_j \mid j \in 1 \ldots n\} \text{ is a permutation of } \{l_i : T_i \mid i \in 1 \ldots n\}
\]

\[
\{k_j : S_j \mid j \in 1 \ldots n\} \prec: \{l_i : T_i \mid i \in 1 \ldots n\} \quad (S-RcdPerm)
\]

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

A high-order language, functions can be passed as arguments to other functions

\[
\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad \text{(S-ARROW)}
\]
The Subtype Relation: Arrow types

\[
\begin{align*}
T_1 &<: S_1 & S_2 &<: T_2 \\
S_1 \rightarrow S_2 &<: T_1 \rightarrow T_2
\end{align*}
\]

\[\text{(S-Arrow)}\]

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

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

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 \( S_1 \rightarrow S_2 \), then we know

1. \( f \) accepts elements of type \( S_1 \); clearly, \( f \) will also accept elements of any subtype \( T_1 \) of \( S_1 \).

2. 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 \( S_2 \).

i.e., any function \( f \) of type \( S_1 \rightarrow S_2 \) can also be viewed as having type \( T_1 \rightarrow T_2 \).
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.

\[ S <: \text{Top} \quad \text{(S-Top)} \]
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.

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

Cf. **Object** in Java.
Subtype Relation: General rules

\[ S \subseteq S \] 
\[ S \subseteq U \quad U \subseteq T \]
\[ S \subseteq T \] 
\[(S\text{-Refl})\] 
\[(S\text{-Trans})\]
A subtyping is a binary relation between types that is closed under the rules:
Subtype Relation

\[
S \triangleleft S \quad \text{(S-REFL)}
\]

\[
\frac{S \triangleleft U \quad U \triangleleft T}{S \triangleleft T} \quad \text{(S-TRANS)}
\]

\[
\{l_i:T_i \mid i \in 1..n+k\} \triangleleft \{l_i:T_i \mid i \in 1..n\} \quad \text{(S-RCDWIDTH)}
\]

for each \(i\)

\[
\frac{S_i \triangleleft T_i}{\{l_i:S_i \mid i \in 1..n\} \triangleleft \{l_i:T_i \mid i \in 1..n\}} \quad \text{(S-RCDDEPTH)}
\]

\[
\{k_j:S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i:T_i \mid i \in 1..n\}
\]

\[
\{k_j:S_j \mid j \in 1..n\} \triangleleft \{l_i:T_i \mid i \in 1..n\} \quad \text{(S-RCDPERM)}
\]

\[
T_1 \triangleleft S_1 \quad S_2 \triangleleft T_2 \quad \frac{T_1 \rightarrow S_2 \triangleleft T_1 \rightarrow T_2}{S_1 \rightarrow S_2} \quad \text{(S-ARROW)}
\]

\[
S \triangleleft \text{Top} \quad \text{(S-TOP)}
\]
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 $\lambda \rightarrow$.*

*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 $T$-$\text{SUB}$ could appear anywhere.

$$
\Gamma \vdash t : S \quad S <: T \\
\frac{}{\Gamma \vdash t : T} \quad (T$-$\text{SUB})
$$
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 $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**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, $\Gamma \vdash t' : T$.

Not hard!
Application case

Case T-App:

\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

By the inversion lemma for evaluation, there are three rules by which \( t \rightarrow t' \) can be derived: E-App1, E-App2, and E-AppAbs.

Proceed by cases.
Application case

Case T-App:
\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

By the inversion lemma for evaluation, there are three rules by which \( t \rightarrow t' \) can be derived: E-App1, E-App2, and E-AppAbs.

Proceed by cases.

Subcase E-App1: \( t_1 \rightarrow t'_1 \quad t' = t'_1 \ t_2 \)

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

\[
\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \\
\Gamma \vdash t_1 \ t_2 : T_{12} \\
(T-\text{App})
\]
Application case

**Case T-App:**

\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

**Subcase E-App2:**

\[ t_1 = v_1 \quad t_2 \rightarrow t'_2 \quad t' = v_1 \ t'_2 \]

Similar.

\[
\begin{align*}
\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11} \\
\phantom{\Gamma \vdash} \\
\Gamma \vdash t_1 \ t_2 : T_{12}
\end{align*}
\]

**T-App**

\[
\begin{align*}
&t_2 \rightarrow t'_2 \\
v_1 \ t_2 \rightarrow v_1 \ t'_2
\end{align*}
\]

**E-App2**
Application case

Case T-App:

\[ t = t_1 \ t_2 \quad \Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11} \quad T = T_{12} \]

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...

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

By using T-Sub, \( \Gamma \vdash t_2 : S_{11} \).

by the substitution lemma, \( \Gamma \vdash t' : T_{12} \).

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

\[
(\lambda x : T_{11} . t_{12}) \ v_2 \to [x \mapsto v_2] t_{12} \quad \text{(E-AppAbs)}
\]
Inversion Lemma for Typing

Lemma: if \( \Gamma \vdash \lambda x : S_1 . s_2 : T_1 \rightarrow T_2 \), then \( T_1 <: S_1 \) and \( \Gamma, x : S_1 \vdash s_2 : T_2 \).
Inversion Lemma for Typing

**Lemma:** if $\Gamma \vdash \lambda x : S_1. s_2 : T_1 \rightarrow T_2$, 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 $U : T_1 \rightarrow T_2$

We want to say “By the induction hypothesis...”, but the IH does not apply (since we do not know that $U$ is an arrow type).

Need another lemma...

**Lemma:** If $U <: T_1 \rightarrow T_2$, then $U$ has the form of $U_1 \rightarrow U_2$, 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
\[ U = U_1 \rightarrow U_2, \text{with } T_1 <: U_1 \text{ and } U_2 <: T_2. \]

The IH now applies, yielding
\[ U_1 <: S_1 \text{ and } \Gamma, x:S_1 \vdash s_2:U_2. \]

From \( U_1 <: S_1 \) and \( T_1 <: U_1 \), rule \textit{S-Trans} gives
\[ T_1 <: S_1. \]

From \( \Gamma, x:S_1 \vdash s_2:U_2 \) and \( U_2 <: T_2 \), rule \textit{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:

\[
\begin{align*}
\Gamma \vdash t_1 : T \\
\Gamma \vdash t_1 \text{ as } T : T \\
v_1 \text{ as } T \rightarrow v_1
\end{align*}
\]

(T-Ascribe) 

(E-Ascribe)
Ascription and Casting

Ordinary ascription:

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

\[
v_1 \text{ as } T \rightarrow v_1 \quad (E\text{-Ascribe})
\]

Casting (cf. Java):

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

\[
\Gamma \vdash v_1 : T \\
\frac{v_1 \text{ as } T \rightarrow v_1}{v_1 \text{ as } T : T} \quad (E\text{-Cast})
\]
Subtyping and Variants

\[ \langle l_i : T_i \rangle_{i \in 1..n} \triangleleft \langle l_i : T_i \rangle_{i \in 1..n+k} \]

(S-VARIANT WIDTH)

for each \( i \)

\[ S_i \triangleleft T_i \]

(S-VARIANT DEPTH)

\[ \langle l_i : S_i \rangle_{i \in 1..n} \triangleleft \langle l_i : T_i \rangle_{i \in 1..n} \]

(S-VARIANT PERM)

\[ \langle k_j : S_j \rangle_{j \in 1..n} \triangleleft \langle l_i : T_i \rangle_{i \in 1..n} \]

\[ \Gamma \vdash t_1 : T_1 \]

\[ \Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle \]
Subtyping and Lists

\[
\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1} \quad (S\text{-List})
\]

i.e., List is a *covariant type* constructor.
Subtyping and References

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

i.e., \textbf{Ref} is \textit{not a covariant} (nor a contravariant) type constructor, but an \textit{invariant}.
Subtyping and References

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

i.e., \textbf{Ref} is \textit{not a covariant} (nor a \textit{contravariant}) type constructor.

Why?

– When a reference is \textit{read}, the context expects a $T_1$, so if $S_1 <: T_1$ then an $S_1$ is ok.
Subtyping and References

\[
\begin{align*}
S_1 &: T_1 \\
T_1 &: S_1 \\
\text{Ref } S_1 &: \text{Ref } T_1
\end{align*}
\]

(S-Ref)

i.e., \textbf{Ref} is not a \textit{covariant} (nor a \textit{contravariant}) type constructor.

Why?

– When a reference is \textit{read}, the context expects a \textit{T}_1, so if \textit{S}_1 \textless: \textit{T}_1 then an \textit{S}_1 is ok.

– When a reference is \textit{written}, the context provides a \textit{T}_1 and if the actual type of the reference is \textit{Ref S}_1, someone else may use the \textit{T}_1 as an \textit{S}_1. So we need \textit{T}_1 \textless: \textit{S}_1.
Subtyping and Arrays

Similarly...

\[
\begin{align*}
S_1 & <: T_1 & T_1 & <: S_1 \\
\text{Array } S_1 & <: \text{ Array } T_1
\end{align*}
\]  
\[\text{(S-Array)}\]

\[
\begin{align*}
S_1 & <: T_1 \\
\text{Array } S_1 & <: \text{ Array } T_1
\end{align*}
\]  
\[\text{(S-ArrayJava)}\]

This is regarded (even by the Java designers) as a mistake in the design.
Observation: a value of type $\text{Ref } T$ can be used in two different ways:

- as a *source* for values of type $T$, and
- as a *sink* for values of type $T$. 
Observation: a value of type $\text{Ref } T$ 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 $\text{Ref } T$ 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

\[ \Gamma, \Sigma \vdash t_1 : \text{Source} \ T_{11} \]
\[ \Gamma, \Sigma \vdash !t_1 : T_{11} \]

(T-DEREF)

\[ \Gamma, \Sigma \vdash t_1 : \text{Sink} \ T_{11} \quad \Gamma, \Sigma \vdash t_2 : T_{11} \]
\[ \Gamma, \Sigma \vdash t_1 := t_2 : \text{Unit} \]

(T-ASSIGN)
Subtyping rules

\[ S_1 <: T_1 \]
\[ \text{Source } S_1 <: \text{Source } T_1 \]

\[ T_1 <: S_1 \]
\[ \text{Sink } S_1 <: \text{Sink } T_1 \]

\[ \text{Ref } T_1 <: \text{Source } T_1 \]
\[ \text{Ref } T_1 <: \text{Sink } T_1 \]

\[ (S-\text{SOURCE}) \]
\[ (S-\text{SINK}) \]
\[ (S-\text{RefSource}) \]
\[ (S-\text{RefSink}) \]
Capabilities

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

– *send* and *receive* capabilities on communication channels,
– *encrypt/decrypt* capabilities of cryptographic keys,
– ...
Intersection and Union Types
Intersection Types

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

\[
T_1 \land T_2 \preceq T_1 \quad (S-INTER1)
\]

\[
T_1 \land T_2 \preceq T_2 \quad (S-INTER2)
\]

\[
S \preceq T_1 \quad S \preceq T_2 \\
\hline
S \preceq T_1 \land T_2 \quad (S-INTER3)
\]

\[
S \rightarrow T_1 \land T_2 \preceq S \rightarrow (T_1 \land T_2) \quad (S-INTER4)
\]
Intersection Types

Intersection types permit a very flexible form of finitary overloading.

\[ + : (\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}) \land (\text{Float} \rightarrow \text{Float} \rightarrow \text{Float}) \]

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.

\( T_1 \lor T_2 \) 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 \( T_1 \lor T_2 \) 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 \( T_1 \lor T_2 \) 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 & 16

- 15.2.5
- 15.3.2