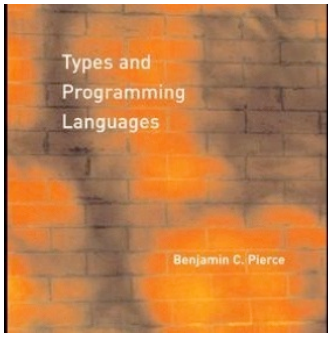
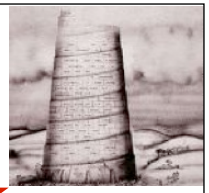
Introduction to Subtyping

Lecture 13

CS 565



Polymorphism



Different varieties of polymorphism:

- Parametric (ML)

  type variables are abstract, and used to encode the fact that the same term can be used in many different contexts.

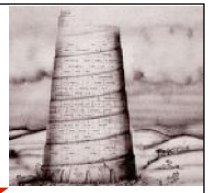
- Subtype (OO)

  a single term may have many types using the rule of subsumption that allows the type system to selectively "forget" information about the term's behavior (and type)

- Ad-hoc (overloading)

  A polymorphic value exhibits different behaviors when "viewed" at different types. Overloading permits a single function symbol to be associated with many implementations.
  Intensional polymorphism allows computation over types at runtime. (e.g., `typecase` or `instanceof`)

Motivation



Consider the rule for typing applications:

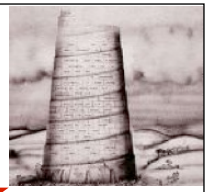
$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}} \quad \text{T\_App}$$

This rule is very rigid. For example, the term:

```
(λr:{x:Nat}.r.x) {x=0, y=1}
```

is not well-typed. The type rule is too restrictive since we're simply passing an argument more precise than necessary.

Subsumption

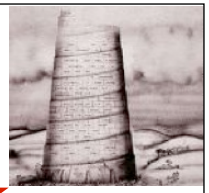


Some types are “better” than others in the sense that the value of one can always be safely used where a value of the other is expected.

Formalize this notion:

- a *subtyping* relation between types: σ `<:` τ
- a *subsumption* relation that allows us to “forget” unnecessary information recorded in a type at the context where it is used.

Subsumption

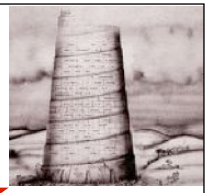


$$\frac{\Gamma \vdash t : \sigma \qquad \sigma <: \tau}{\Gamma \vdash t : \tau}$$

This rule tells us that if $\sigma <: \tau$, every element $v \in \sigma$ is also an element of τ.

Thus, if our subtype relation defined `{x:Nat,y:Bool} <: {x:Nat}`, then the subsumption rule would allow us to derive `{x=0,y=true}:{x:Nat}`

A Subtype Relation



Intuition: σ <: τ if an element of σ may be safely used wherever an element of τ is expected.

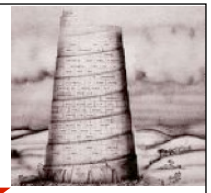
- σ is "better" than τ
- σ is a subset of τ
- σ is more informative/richer than τ.

$$\sigma <: \sigma \qquad \text{(Reflexive)}$$

$$\frac{\sigma <: \tau' \quad \tau' <: \tau}{\sigma <: \tau} \qquad \text{(Transitive)}$$

We now consider subtyping for each form of type (records, functions, etc.)

Records



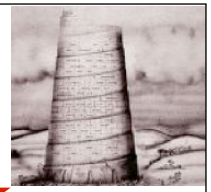
"Width" subtyping:

- forget fields "on the right"

  $\{l_i : \tau_i \mid i \in 1...n+k\} <: \{l_i : \tau_i \mid i \in 1...n\}$

- Intuition: `{x:Nat}` is the type of all records that have at least one field numeric field `x`.
- A record type with more fields is a subtype of a record with fewer fields.
- The more fields a record has, the more constraints it imposes on its uses, and thus describes fewer values.

Records

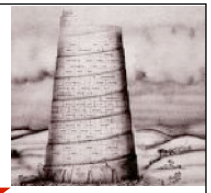


Depth subtyping:

$$\frac{\sigma_i <: \tau_i}{\{l_i:\sigma_i \mid^{i \in 1...n}\} <: \{l_i:\tau_i \mid^{i \in 1...n}\}}$$

- types of individual fields can vary provided all types in the record satisfy the subtype relation.

Example



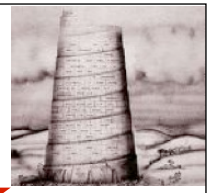
To show that

```
{x:{a:I,b:I}, y:{m:I}} <: {x:{a:I},y:{}}
```

$$\frac{\texttt{\{a:I,b:I\} <: \{a:I\}} \qquad \texttt{\{m:I\} <: \{\}}}{\texttt{\{x:\{a:I,b:I\},y:\{m:I\}\} <: \{x:\{a:I\},y:\{\}\}}}$$

Where is width subtyping used? Where is depth subtyping used?

Example



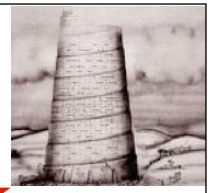
To show that

```
{x:{a:I,b:I},y:{m:I}} <: {x:{a:I},y:{m:I}}
```

$$\frac{\texttt{\{a:I,b:I\} <: \{a:I\}} \qquad \texttt{\{m:I\} <: \{m:I\}}}{\texttt{\{x:\{a:I,b:I\},y:\{m:I\}\} <: \{x:\{a:I\},y:\{m:I\}\}}}$$

Uses reflexive property of subtype relation.

Example



To show that

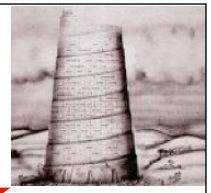
$$\{x: \{a:I,b:I\}, y:\{m:I\}\} <: \{x:\{a:I\}\}$$

$$\frac{\{a:I,b:I\} <: \{a:I\}}{\{x:\{a:I,b:I\},y:\{m:I\}\}<:\{x:\{a:I,b:I\}\}\quad \{x:\{a:I,b:I\}\}<:\{x:\{a:I\}\}}$$

$$\{x:\{a:I,b:I\},y:\{m:I\}\} <: \{x:\{a:I\}\}$$

Uses transitive property of subtype relation.

Records



Permutation subtyping allows us to ignore the order of fields in records

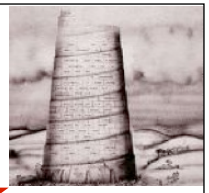
$$\frac{\{k_j{:}\sigma_j \mid j \in 1\ldots n\} \text{ a permutation of } \{l_i{:}\tau_i \mid i \in 1\ldots n\}}{\{k_j{:}\sigma_j \mid j \in 1\ldots n\} <: \{l_i{:}\tau_i \mid i \in 1\ldots n\}}$$

Why is this safe?

Example: `{c: Bool, d: Nat} <: {d: Nat, c: Bool}`

- Is the subtype relation anti-symmetric in the presence of permutations?

Variations

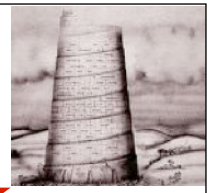


Not all languages adopt all these features.

Example: Java

- A subclass may not change the argument or result types of a method of its superclass
  - no depth subtyping on methods
- Each class has just one superclass and each class member can be assigned a single index, adding new indices on the right
  - no permutation rule
  - permutations allowed only when considering interfaces

Arrow Types



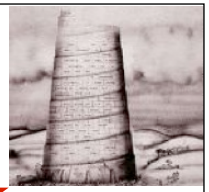
$$\frac{\tau 1 <: \sigma 1 \qquad \sigma 2 <: \tau 2}{\sigma 1 \to \sigma 2 <: \tau 1 \to \tau 2}$$

The subtype relation is contravariant in the type of the argument, and covariant in the type of the result.

Intuition:

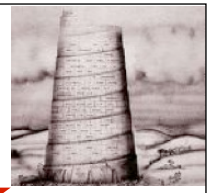
if we have a function `f` of type $\sigma 1 \to \sigma 2$, then we know `f` accepts elements of any subtype $\tau 1$ `<:` $\sigma 1$. Since `f` returns elements of type $\sigma 2$, these results belong to any supertype $\tau 2$ of $\sigma 2$.

Arrow Types



It is safe to allow a function of type $\sigma 1 \to \sigma 2$ to be used in a context $\tau 1 \to \tau 2$ provided none of the arguments supplied will "surprise" it ($\tau 1 <: \sigma 1$) and none of its results will "surprise" the context in which it is used ($\sigma 2 <: \tau 2$)

Examples

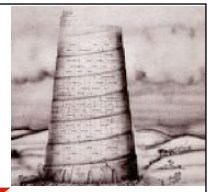


$b \to b <: a \to b$

$a \to a <: a \to b$

$(a \to b) \to a <: (b \to a) \to b$ assuming $a <: b$

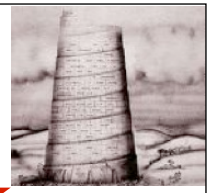
Top



It is convenient to have a type that is the supertype of every type. Introduce a new type constant `Top` plus a rule that makes Top a maximal element in the subtype relation:

$$\sigma \mathrel{<:} \texttt{Top}$$

Example revisited



Recall the term:

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

$f \equiv \lambda\ r{:}\ \{x{:}Nat\}.\ r.x$ $\qquad$ $xy \equiv \{x = 0,\ y = 1\}$

$Rx \equiv \{x{:}Nat\}$ $\qquad$ $Rxy \equiv \{x{:}Nat,\ y{:}Nat\}$

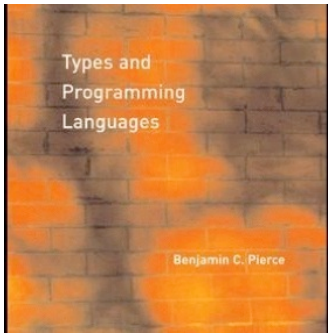
$$\dfrac{\vdash f{:}\ Rx \to Nat \qquad \dfrac{\dfrac{\vdash 0{:}Nat \quad \vdash 1{:}Nat}{\vdash xy{:}\ Rxy} \qquad Rxy <: Rx}{\vdash xy : Rx}}{\vdash f\ xy : Nat}$$

(… : $\vdash f{:}\ Rx \to Nat$)

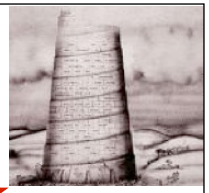
Subtyping (cont)

Lecture 15

CS 565



Formalization of Subtyping



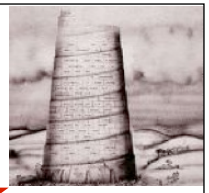
Inversion of the subtype relation:

- If $\sigma <: \tau_1 \to \tau_2$ then σ has the form $\sigma_1 \to \sigma_2$, and $\tau_1 <: \sigma_1$ and $\sigma_2 <: \tau_2$
- If $\sigma <: \{l_i : \tau_i \mid i \in 1\ldots n\}$, then σ has the form $\{k_j : \sigma_j \mid j \in 1\ldots m\}$ with at least the labels $\{l_i \mid i \in 1\ldots n\} \subseteq \{k_j \mid j \in 1\ldots m\}$ and with $\sigma_j <: \tau_i$ for each common label $l_i = k_j$

  This lemma says that a subtype of an arrow type must be an arrow type that is contravariant in the argument and covariant in the result, and a subtype of a record type must be a record type that either has more fields (width) in some order (permute) and that the types of common fields obey the subtype relation (depth)

Proof of preservation and progress follow as before using induction on typing derivations.

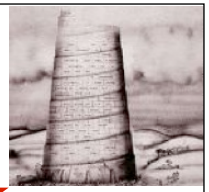
Casts and Ascription



Ascription `(t as T)` is a form of checked documentation:

- it allows the programmer to record an assertion that the subterm of a complex expression has some specific type.
- Ascription in the presence of subtyping raises interesting issues.

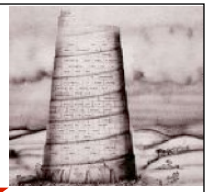
Up Casts



Upcasts: a term is ascribed a supertype of the type that would normally be assigned.

- Benign since it effectively “hides” some parts of a value so they cannot be used in some surrounding context.

Down Casts



Downcasts: assign a type to a term that would not be assigned by the typechecker.

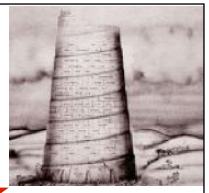
No specific relation between the term expected and the term desired.

- `List` objects in Java expect elements belonging to `Object`.
- When an element of type `T` is stored, it is promoted to `Object`.
- When an element is extracted, it must be downcast to a type suitable for the context in which it is to be used.

  Requires runtime checks

  Complicates implementations since type information must be presented at runtime to validate the cast.

Subtyping References



Try covariance:

$$\frac{\sigma <: \tau}{\sigma\ \texttt{ref} <: \tau\ \texttt{ref}}$$

Assume $\sigma <: \tau$

- The following holds:

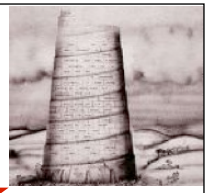
  $\texttt{x:}\tau\texttt{,y:}\sigma\ \texttt{ref,f:}\sigma \to \texttt{int} \vdash \texttt{y := x ; f (!y)}$

  Unsound: `f` is called on a τ but is defined only on σ

If we want covariance of references we can recover type safety with a runtime check for each `y:=x` assignment

- The actual type of `x` matches the actual type of `y`
- This is not a particularly good design.
- Note: Java has covariant arrays

Subtyping References



Another approach (contravariance):

$$\frac{\sigma <: \tau}{\tau \text{ ref} <: \sigma \text{ ref}}$$

- Assume $\sigma <: \tau$

The following holds:

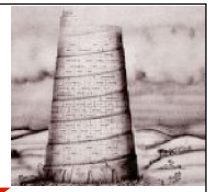
$$x{:}\tau,\ y{:}\tau \text{ ref},\ f{:}\sigma \to \text{int} \vdash y := x;\ f(!y)$$

- Unsound: `f` is called on a τ but is defined only on σ

references can be used in two ways:

- for reading, context expects a value of type σ but the reference yields a value of type τ then we need $\tau <: \sigma$.
- for writing, the new value provided by the context will have type σ. If the actual type of the reference is τ ref, then this value may be read and used as a τ. This will only be safe if $\sigma <: \tau$.

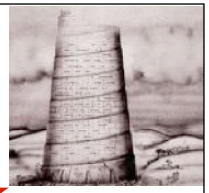
Solution



References should be typed invariantly:

- No subtyping for references (unless there are runtime checks)
- Arrays (which are implemented using updates) should be typed invariantly.
- Similarly, mutable records should also be invariant.

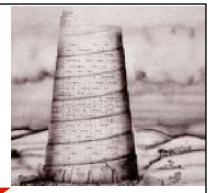
Refinements



A value of type `ref T` can be used in two different ways: as a source for reading and a sink for writing:

- 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

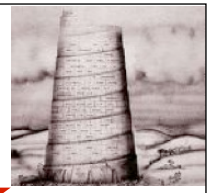
Modified Typing Rules



$$\frac{\Gamma, \Sigma \vdash \texttt{e}_1\texttt{:source } \tau_1}{\Gamma, \Sigma \vdash \texttt{!e}_1 \texttt{ : } \tau_1}$$

$$\frac{\Gamma, \Sigma \vdash \texttt{e}_1\texttt{:sink } \tau \quad \Gamma, \Sigma \vdash \texttt{e}_2\texttt{:}\tau}{\Gamma, \Sigma \vdash \texttt{e}_1\texttt{:=e}_2\texttt{:Unit}}$$

Subtyping Rules



$$\frac{\sigma <: \tau}{\texttt{source}\ \sigma <: \texttt{source}\ \tau}$$

covariant on reads

$$\frac{\tau <: \sigma}{\texttt{sink}\ \sigma <: \texttt{sink}\ \tau}$$

contravariant on writes

$$\texttt{ref}\ \tau <: \texttt{source}\ \tau$$

$$\texttt{ref}\ \tau <: \texttt{sink}\ \tau$$

refs can be “downgraded” to source or sink