Subtyping

Motivation

With our usual typing rule for applications

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

the term

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

is *not* well typed.

Motivation

With our usual typing rule for applications

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

the term

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

is *not* well typed.

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

Polymorphism

A *polymorphic* function may be applied to many different types of data.

Varieties of polymorphism:

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

Our topic for the next few lectures is *subtype* polymorphism, which is based on the idea of *subsumption*.

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

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

The Subtype Relation: General rules

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

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

Example

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

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

So, by subsumption,

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

and hence

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

is well typed.

The Subtype Relation: Records

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

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

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

Note that the record type with *more* fields is a *sub*type 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:

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

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:

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

The types of individual fields may change.

Example

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

Another example

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

(board)

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 super*class* ("single inheritance" of classes)

  $\longrightarrow$ *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

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

Note the order of $\mathtt{T}_1$ and $\mathtt{S}_1$ 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 $\mathtt{f}$ of type $\mathtt{S}_1 \to \mathtt{S}_2$, then we know that $\mathtt{f}$ accepts elements of type $\mathtt{S}_1$; clearly, $\mathtt{f}$ will also accept elements of any subtype $\mathtt{T}_1$ of $\mathtt{S}_1$. The type of $\mathtt{f}$ also tells us that it returns elements of type $\mathtt{S}_2$; we can also view these results belonging to any supertype $\mathtt{T}_2$ of $\mathtt{S}_2$. That is, any function $\mathtt{f}$ of type $\mathtt{S}_1 \to \mathtt{S}_2$ can also be viewed as having type $\mathtt{T}_1 \to \mathtt{T}_2$.

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.

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

Cf. `Object` in Java.

Subtype relation

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

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

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

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

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

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

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

Aside: Structural vs. declared subtyping

The subtype relation we have defined is *structural*: We decide whether `S` is a subtype of `T` by examining the structure of `S` and `T`.

By contrast, the subtype relation in most OO languages (e.g., Java) is *explicitly declared*: `S` is a subtype of `T` only if the programmer has stated that it should be.

There are pragmatic arguments for both.

For the moment, we'll concentrate on structural subtyping, which is the more fundamental of the two. (It is sound to *declare* `S` to be a subtype of `T` only when `S` is structurally a subtype of `T`.)

We'll come back to declared subtyping when we talk about Featherweight Java.