particolare
valenza nella
programmazione
ad oggetti

Subtyping and Objects

Massimo Merro

20 November 2017

Polymorphism

So far, our type systems are very rigid: there is little support to code reuse.

Polymorphism

Ability to use expressions at different places with different types.

- Ad-hoc polymorphism (overloading).
  / e div stessa operazione ma per reali e interi
  e.g. in Moscow ML (but also in Java) the built in '$+$' can be used to add two integers or to add two reals.
- Parametric polymorphism - as in ML.
  Can write, for instance, a function that can take as an argument of type list α and computes its lenght (parmametric - uniformly in whatever α is).
  linguaggi funzionali
- Subtype polymorphism - as in most Object-Oriented Languages.
  Dating back to 1960s (Simula etc.) formalized in the 1980s. We will focus on this kind of subtyping!
  primi linguaggi ad oggetti

Subtyping - Motivation

Let us recall the typing rules for the functional extension:

$$(\text{fun}) \ \frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash (\text{fn } x : T \Rightarrow e) : T \rightarrow T'}$$

$$(\text{app}) \ \frac{\Gamma \vdash e_1 : T \rightarrow T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \, e_2 : T'}$$

agreement stretto

According to our type system:

$$\Gamma \vdash (\text{fn } \mathrm{x} : \{\mathit{left} : \text{int}\} \Rightarrow \#\mathit{left}\ \mathrm{x}) : \{\mathit{left} : \text{int}\} \rightarrow \text{int}$$

in genere un sottotipo ha più informazioni

Thus, we cannot type the following:

$$\Gamma \nvdash (\text{fn } \mathrm{x} : \{\mathit{left} : \text{int}\} \Rightarrow \#\mathit{left}\ \mathrm{x})\{\mathit{left} = 3, \mathit{right} = 5\}$$

Even if we are giving the function a better argument, with more structure, than it is required by the function itself!

Subsumption →Upcast

- In which sense we are passing a better argument?

  Any value of type $\{left : \text{int}, right : \text{int}\}$ can be safely used whenever a value of type $\{left : \text{int}\}$ is expected!

- Introduce a subtyping relation between types, written $T <: T'$, read as T is a subtype of T': an object of type T can always be used in a context where an object of type T' is expected!

  ***

- So, for instance:

  similar object in Java

$$\{left : \text{int}, right : \text{int}\} <: \{left : \text{int}\} <: \{\}$$

- The subtype relation $<:$ is then used by introducing a *subsumption rule*

$$(\text{sub}) \quad \frac{\Gamma \vdash e : T \qquad T <: T'}{\Gamma \vdash e : T'}$$

Example

By an application of rule (sub) we can deduce that sottotipo di

$$\text{(sub)}\ \frac{\Gamma \vdash \{\mathit{left} = 3, \mathit{right} = 5\} : \{\mathit{left} : \text{int}, \mathit{right} : \text{int}\} \qquad \{\mathit{left} : \text{int}, \mathit{right} : \text{int}\} <: \{\mathit{left} : \text{int}\}}{\Gamma \vdash \{\mathit{left} = 3, \mathit{right} = 5\} : \{\mathit{left} : \text{int}\}}$$

and type the previous expression:

$$\text{(app)}\ \frac{\{\} \vdash (\text{fn } \mathrm{x} : \{\mathit{left} : \text{int}\} \Rightarrow \#\mathit{left}\ \mathrm{x}) : \{\mathit{left} : \text{int}\} \rightarrow \text{int} \qquad \{\} \vdash \{\mathit{left} = 3, \mathit{right} = 5\} : \{\mathit{left} : \text{int}\}}{\{\} \vdash (\text{fn } \mathrm{x} : \{\mathit{left} : \text{int}\} \Rightarrow \#\mathit{left}\ \mathrm{x})\{\mathit{left} = 3, \mathit{right} = 5\} : \text{int}}$$

The Subtype relation $T <: T'$

It is a reflexive and transitive relation:

$$\text{(s-refl)} \quad \frac{-}{T <: T}$$

$$\text{(s-trans)} \quad \frac{T <: T' \qquad T' <: T''}{T <: T''}$$

Let us define subtyping for the different data structures of our language.

Subtyping - Records

Allowing reordering of fields: → l'ordine non è più importante

$$(\text{rec-perm}) \; \frac{\pi \text{ a permutation of } 1, 2, \ldots k}{\{p_1{:}T_1, \ldots, p_k{:}T_k\} <: \{p_{\pi(1)}{:}T_{\pi(1)}, \ldots, p_{\pi(k)}{:}T_{\pi(k)}\}}$$

permutazione

The *subtype relation* is not anti-symmetric: a preorder, not a partial order.

Forgetting about fields on the right:

$$(\text{rec-width}) \; \frac{-}{\{p_1{:}T_1, \ldots, p_k{:}T_k, p_{k+1}{:}T_{k+1}, \ldots p_z{:}T_z\} <: \{p_1{:}T_1, \ldots, p_k{:}T_k\}}$$

tagliare

If we do reordering first, we can forget about any field.

Allowing subtype within fields:

$$(\text{rec-depth}) \; \frac{T_1 <: T'_1 \quad \ldots \quad T_k <: T'_k}{\{p_1{:}T_1, \ldots, p_k{:}T_k\} <: \{p_1{:}T'_1, \ldots, p_k{:}T'_k\}}$$

andare in profondità

Subtyping is said to be covariant on record types!

Example: Combining rules

For instance, we can derive:

$$(\underline{\text{rec-d}})\ \dfrac{(\text{rec-w})\ \dfrac{-}{\{p : \text{int}, q : \text{int}\} <: \{p : \text{int}\}} \qquad (\text{rec-w})\ \dfrac{-}{\{r : \text{int}\} <: \{\}}}{\{x : \{p{:}\text{int}, q{:}\text{int}\}\ ,\ y : \{r{:}\text{int}\}\}\ <:\ \{x : \{p{:}\text{int}\}\ ,\ y : \{\}\}}$$

Another possibility is:

$$(\underline{\text{trans}})\ \dfrac{(\text{w})\ \dfrac{-}{\{x : \{p{:}\text{int}, q{:}\text{int}\}\ ,\ \overbrace{y : \{r{:}\text{int}\}\}}^{\text{tagliato}} <: \{x : \{p : \text{int}, q : \text{int}\}\}} \qquad \Delta}{\{x : \{p{:}\text{int}, q{:}\text{int}\}\ ,\ y : \{r{:}\text{int}\}\}\ <:\ \{x : \{p{:}\text{int}\}\}}$$

where Δ is:

$$(\underline{\text{rec-depth}})\ \dfrac{(\text{rec-width})\ \dfrac{-}{\{p : \text{int}, q : \text{int}\} <: \{p : \text{int}\}}}{\{x : \{p{:}\text{int}, q{:}\text{int}\}\} <: \{x : \{p{:}\text{int}\}\}}$$

Subtyping - Functions

The subtyping rule is the following:

$$(\text{fun-sub}) \ \frac{T_1 :> T_1' \qquad T_2 <: T_2'}{T_1 \rightarrow T_2 <: T_1' \rightarrow T_2'}$$

We say that subtyping on functions is:
argomento

- contravariant on the left of $\rightarrow$
- covariant on the right of $\rightarrow$ (like the rule (rec-depth)).

risultato

Thus, if $f : T_1 \rightarrow T_2$ then we can use f in any context where:

- we give f any argument of type T_1', with $T_1' <: T_1$
- we use the result of f as it was of type T_2', with of $T_2 <: T_2'$.

For instance, if we define f in a let construct:

$$\text{let } f : T = \text{fn x} : \{p : \text{int}\} \Rightarrow \{a = \#p\,\text{x}\,,\ b = 28\} \text{ in } e$$

then when typing e we must use a type environment Γ such that $\Gamma(f) = T = \{p : \text{int}\} \rightarrow \{a : \text{int}\,,\ b : \text{int}\}$.

By subtyping we can use f in e as it had one of the following types:

$$\{p : \text{int}\} \rightarrow \{a : \text{int}\}$$

$$\{p : \text{int}\,,\ q : \text{int}\} \rightarrow \{a : \text{int}\,,\ b : \text{int}\}$$

$$\{p : \text{int}\,,\ q : \text{int}\} \rightarrow \{a : \text{int}\}$$

basically, because

$$\begin{array}{rcl} \overset{T_1}{\{p : \text{int}\}} & :> & \overset{T_1'}{\{p : \text{int}\,,\ q : \text{int}\}} \\ \underset{T_2}{\{a : \text{int}\,,\ b : \text{int}\}} & <: & \underset{T_2'}{\{a : \text{int}\}} \end{array}$$

On the other hand, in the program

$$\text{let } f : \hat{T} = \text{fn } \mathrm{x} : \{p : \text{int}, q : \text{int}\} \Rightarrow \{a = (\#p\ \mathrm{x}) + (\#q\ \mathrm{x})\} \text{ in } e$$

when typing e we must use a type environment Γ such that:
$\Gamma(f) = \hat{T} = \{p : \text{int}, q : \text{int}\} \to \{a : \text{int}\}$. "tipo di fabbrica"
By subtyping, we can use f in e as it had, for instance, type:

$$\{p : \text{int}, q : \text{int}, r : \text{int}\} \to \{a : \text{int}\}$$

However, by no means we can use f in e as it had one of the following types:

- $\{p : \text{int}\} \to T$, for any Γ and T
- $T \to \{a : \text{int}, b : \text{int}\}$, for any Γ and T

Subtyping - Products and Sums

Subtyping is covariant on both components of products:

$$(\underline{\text{prod-sub}})\ \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 * T_2 <: T_1' * T_2'}$$

Again, covariant on both components of summations

$$(\underline{\text{sum-sub}})\ \frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 + T_2 <: T_1' + T_2'}$$

Subtyping - References

We don't introduce subtyping rules for references to avoid inconsistencies while typing. See exercises.

or $T_1 :> T_2$

$$\cancel{\frac{T_1 <: T_2}{ref\ T_1 <: ref\ T_2}}$$

covariante funziona

tipiamo problemi che
a round time danno problemi

Perdo la soundness

dice che se un programma e' ben tipato non va in deadlock

What else does it change?

Semantics

No change (note that we have not changed the grammar for expressions)

Properties

Of course, we still have Type Preservation and Progress.

Implementation

Type inference is now more subtle, as the typing rules are not syntax-directed. Getting a good runtime implementation is also tricky, especially with field reordering.

Subtyping - Down-casts

The subsumption rule (sub) permits up-casting at any moment: If $T <: T'$, any expression e of type T can be used in any context where an expression of type T' is expected!

How about down-casting? Suppose to add in the grammar a construct:

$$e \quad ::= \quad \ldots \quad | \; (T)e$$

Subcast

with the typing rule

$$(\text{down-cast}) \; \frac{\Gamma \vdash e : T' \quad T <: T'}{\Gamma \vdash (T)e : T}$$

Can we statically type-check an expression $(T)e$?

No! This can be done only dynamically because the correctness of the down-casting depends on the "real type", at runtime, of e.

Example on down-casting

Recall that

$$\{left : \mathsf{int}, right : \mathsf{int}\} <: \{left : \mathsf{int}\} <: \{\}$$

Let us suppose to have a fragment of code of the form

$$\mathrm{l} := !\mathrm{m}; e$$

where, at runtime, at location m, there will be in the store an expression of one of the three types above.

Now, can we statically type-check in e the following expression

$$(\{left : \mathsf{int}\})!\mathrm{l} \ ?$$

No! Because at runtime $!\mathrm{l}$ could return an object of type $\{left : \mathsf{int}, right : \mathsf{int}\}$ but $\{left : \mathsf{int}\} \not<: \{left : \mathsf{int}, right : \mathsf{int}\}$.

Thus, down-casting can only be dynamically type-checked.

subtiping é una sorta di isomorfismo

(Very simple) Objects

$$\text{let } \mathrm{cnt} : \{\mathit{get} : \text{unit} \rightarrow \text{int}\,,\ \mathit{inc} : \text{unit} \rightarrow \text{unit}\} =$$
$$\quad \text{let } \mathrm{val} : \text{ref int} = \text{ref } 0$$
$$\quad \text{in}$$
$$\qquad \{\mathit{get} = \text{fn } \mathrm{y}{:}\text{unit} \Rightarrow !\mathrm{val},$$
$$\qquad\ \ \mathit{inc} = \text{fn } \mathrm{y}{:}\text{unit} \Rightarrow \mathrm{val} := !\mathrm{val} + 1\}$$
$$\text{in}$$
$$\quad (\#\mathit{inc}\ \mathrm{cnt})(); (\#\mathit{get}\ \mathrm{cnt})()$$

- cnt models a simple object of type

  ritorna il valore del contatore

  ho un oggetto cnt con due funzioni che lavorano sullo stato ← $\text{Counter} = \{\mathit{get} : \text{unit} \rightarrow \text{int}\,,\ \mathit{inc} : \text{unit} \rightarrow \text{unit}\}$

  incrementa il contatore

  with two methods: *get*() and *inc*()[1];
- val records the state (ie the value) of the counter which can be accessed only be means of the two methods.

[1] For simplicity we write $e()$ instead of $e(\mathit{skip})$.

Using Subtyping

$$
\begin{aligned}
&\mathsf{let}\ \mathrm{cnt} : \{\mathit{get} : \mathsf{unit} \to \mathsf{int}\,,\ \mathit{inc} : \mathsf{unit} \to \mathsf{unit}\,,\ \mathit{reset} : \mathsf{unit} \to \mathsf{unit}\} = \\
&\quad \mathsf{let}\ \mathrm{val} : \mathsf{ref\ int} = \mathsf{ref}\ 0 \\
&\quad \mathsf{in} \\
&\qquad \{\mathit{get} = \mathsf{fn}\ \mathrm{y}{:}\mathsf{unit} \Rightarrow !\mathrm{val}, \\
&\qquad\ \ \mathit{inc} = \mathsf{fn}\ \mathrm{y}{:}\mathsf{unit} \Rightarrow \mathrm{val} := !\mathrm{val} + 1\} \\
&\qquad\ \ \mathit{reset} = \mathrm{y}{:}\mathsf{unit} \Rightarrow \mathrm{val} := 0\} \\
&\mathsf{in} \\
&\quad (\#\mathit{inc}\ \mathrm{cnt})(); (\#\mathit{get}\ \mathrm{cnt})()
\end{aligned}
$$

The use of the new variable cnt is perfectly safe because now it has type

$$\mathsf{ResetCounter} = \{\mathit{get} : \mathsf{unit} \to \mathsf{int}\,,\ \mathit{inc} : \mathsf{unit} \to \mathsf{unit}\,,\ \mathit{reset} : \mathsf{unit} \to \mathsf{unit}\}$$

resetCounter è sottotipo di counter

with $\mathsf{ResetCounter} <: \mathsf{Counter}$.

Object Generators

What about a function to generate new objects each time we wish so?

$$
\begin{array}{l}
\textsf{let}\ \mathrm{newCnt} : \textsf{unit} \rightarrow \underbrace{\{\textit{get} : \textsf{unit} \rightarrow \textsf{int}\,,\ \textit{inc} : \textsf{unit} \rightarrow \textsf{unit}\}}_{\text{costruttore di contatori}} = \\
\quad \textsf{fn}\ \mathrm{z} : \textsf{unit} \Rightarrow \\
\qquad \textsf{let}\ \mathrm{val} : \textsf{ref int} = \textsf{ref}\ 0 \\
\qquad \textsf{in} \\
\qquad\quad \{\textit{get} = \textsf{fn}\ \mathrm{y} : \textsf{unit} \Rightarrow !\mathrm{val}, \\
\qquad\quad\ \ \textit{inc} = \textsf{fn}\ \mathrm{y} : \textsf{unit} \Rightarrow \mathrm{val} := !\mathrm{val} + 1\} \\
\textsf{in} \\
\quad (\#\textit{inc}\ (\underline{\textit{newCnt}()}))()
\end{array}
$$

With our simple data structures we can start programming in a object-oriented style!

By the way, what about classes? Can we represent them?

Classes in Java (small example)

Consider the following Java class:

```
class Counter
  { protected int p;
     Counter()  { this.p=0; }
     int get()  { return this.p; }
     void inc() { this.p++; }
      };
```

Can we model something similar?

Reusing Method Code (Simple Classes)

Recall the type $\text{Counter} = \{get : \text{unit} \rightarrow \text{int}\,,\ inc : \text{unit} \rightarrow \text{unit}\}$.
First, make the internal state into a record:

$$\text{CounterRep} = \{\text{p} : \text{ref int}\}$$

$$
\begin{array}{l}
\text{let } \mathrm{cntClass} : \text{CounterRep} \rightarrow \text{Counter} = \\
\quad \text{fn } \mathrm{val} : \text{CounterRep} \Rightarrow \\
\qquad \{get = \text{fn } \mathrm{y}{:}\text{unit} \Rightarrow !(\#p\ \mathrm{val}), \\
\qquad\ inc = \text{fn } \mathrm{y}{:}\text{unit} \Rightarrow (\#p\ \mathrm{val}) := !(\#p\ \mathrm{val}) + 1\} \\
\text{in} \\
\quad \text{let } \mathrm{newCnt} : \text{unit} \rightarrow \text{Counter} = \\
\qquad \text{fn } \mathrm{z} : \text{unit} \Rightarrow \\
\qquad\quad \text{let } \mathrm{x} : \text{CounterRep} = \{p = \text{ref } 0\} \text{ in} \\
\qquad\qquad \underline{\mathrm{cntClass}\ \mathrm{x}} \\
\quad \text{in } \ldots
\end{array}
$$

per costruire un contatore
uso la classe

Reusing Method Code (Simple Classes)

Can we represent the following subclass in Java?

```
class ResetCounter extends Counter
  { void reset() { this.p=0; }
      };
```

$$
\begin{array}{l}
\textsf{let}\ \mathrm{resetCntClass} : \mathsf{CounterRep} \to \mathsf{ResetCounter} = \\
\quad \textsf{fn}\ \mathrm{val} : \mathsf{CounterRep} \Rightarrow \\
\quad\quad \textsf{let}\ \mathrm{super} : \mathsf{Counter} = \mathrm{cntClass\ val}\ \textsf{in} \\
\quad\quad\quad \{\mathit{get} = \#\mathit{get}\ \mathrm{super} \\
\quad\quad\quad\ \ \mathit{inc} = \#\mathit{inc}\ \mathrm{super} \\
\quad\quad\quad\ \ \mathit{reset} = \textsf{fn}\ \mathrm{y}{:}\mathsf{unit} \Rightarrow (\#\mathit{p}\ \mathrm{val}) := 0\} \\
\textsf{in}\quad \ldots
\end{array}
$$

and ResetCounter <: Counter entails
CounterRep → ResetCounter <: CounterRep → Counter!