Overview

Covered so far:

- ➜ Syntax of programming languages
- ➜ Semantics of programming languages:
  - static semantics: scopes, types
  - dynamic semantics: big step/small step semantics
- Abstract machines:
  - substitution
  - environments
  - control stacks
- ➜ Exceptions, Continuations
- ➜ Polymorphism
- ➜ Aggregate data types
- ➜ Type inference
- ➜ Overloading and some aspects of type classes

What's next?

- ➜ Subtyping
- ➜ Data abstraction
- ➜ Featherweight Java
- ➜ Inheritance

Subtyping

Why subtyping?

- ➜ eliminates the need to explicitely convert between elements of different types
- ➜ can be used to express program properties
- ➜ essential in OO-languages: closely related to subclass-relationship

Subtype relation

We write $\tau \leq \sigma$ to express that τ is a subtype of σ.
If $\tau \leq \sigma$ then wherever a value of type σ is required, we can use a value of type τ instead.

Two different forms of subtyping

Subset Interpretation *If $\tau \leq \sigma$ then every value of τ is also a value of σ*

- ➜ *even integers and integers*
- ➜ *empty lists and lists*

Coercion Interpretation *If $\tau \leq \sigma$ then every value of τ can be coerced to a value of type σ in a unique way.*

- ➜ *integer and floating point values (almost...)*
  - `3` to `3.0`
- ➜ *characters and strings*
  - `'w'` to `"w"`

We include the type `Float` into (type class-less) MinML

➜ operations on `Float`: $+_{Float}$, $*_{Float}$, and so on

We want to be able to write[1]

➜ $3 +_{Float} 4.0$

➜ $3.0 +_{Float} 4$

This requires that either

➜ the floating point operation detects at run time presence of integer arguments and converts them to floating point values, or

➜ the static semantic analysis detect the integer value and inserts coercion operations.

[1] Note that this is different to overloading as discussed in the previous lecture

Properties of Subtyping

As coercion interpretation is more expressive than subset interpretation, we will discuss the coercion interpretation in more detail.
The subtype relation should be reflexive and transitive:

➜ Reflexivity:

$$\frac{}{\tau \leq \tau}$$

➜ Transitivity:

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

➜ obvious for subset interpretation (reflexivity and transitivity of subset relation)

➜ holds also for coercion interpretation (identity and composition of functions)

Coherence

We have to be careful the subtyping relation is coherent

➜ coerced value has to be unique

Counter example

If we define `Int` and `Float` to be subtypes of `String`, with the coercion yielding the string representation, then $3 : \texttt{Int}$ can be coerced to

➜ `"3"` (by converting it directly), but also to

➜ `"3.0"` (first coercing to `3.0:Float`)

Subsumption

An important property of subtyping is expressed by the rule of subsumption

➜ implicit subsumption:

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \sigma}{\Gamma \vdash e : \sigma}$$

- rule not syntax directed

➜ explicit subtyping (we add a cast expression (σ)):

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \sigma}{\Gamma \vdash (\sigma)e : \sigma}$$

- syntax directed

Safety

Explicit Subtyping if $\sigma \leq \tau$ then casting a value of type σ to τ must yield a value of type τ

Implicit Subtyping the dynamic semantics must ensure that the value of each primitive operation is defined for the values of any subtype of the expected argument types

Subtyping

What is the relation between the types

- `(Int, Int)`
- `(Float, Int)`
- `(Int, Float)`
- `(Float, Float)`?

Subtyping rule for products (depth subtyping)

$$\frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{(\tau_1, \tau_2) \leq (\sigma_1, \sigma_2)}$$

Subtyping rule for sums

$$\frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{(\tau_1 + \tau_2) \leq (\sigma_1 + \sigma_2)}$$

Subtyping

What is the relation between the types

- `Int` $\rightarrow$ `Int`
- `Float` $\rightarrow$ `Int`
- `Int` $\rightarrow$ `Float`
- `Float` $\rightarrow$ `Float`

Given the coercion function:

- `intToFloat:: Int -> Float`

Can we use them to define conversion functions of type

- `(Int -> Int) -> (Int -> Float)`
- `(Int ->Int) -> (Float -> Float)`
- `(Int ->Int) -> (Float -> Int)`
- `(Int -> Float) -> (Float -> Int)`
- ...

Subtyping rules for functions

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{(\tau_1 \to \tau_2) \leq (\sigma_1 \to \sigma_2)}$$

Variance

- ➜ Rules which specify how a type constructor interacts with subtyping are called variance principles
- ➜ if a constructor preserves subtyping in a given argument position, it is called covariant
  - product type constructor is covariant in both arguments
  - sum type constructor is covariant in both arguments
- ➜ if a constructor inverts subtyping in a given argument position, it is called contravariant
  - function type constructor is contravariant in first argument, and
  - covariant in second argument
- ➜ some construnctors are invariant