CSCI 740 - Programming Language Theory

Lecture 15

Subtyping

Instructor: Hossein Hojjat

October 4, 2017

Subtyping in Scala: Primitive Types

- System F style polymorphism is **parametric** polymorphism:
  - Program can be typed generically using variables in place of actual types
  - Instantiate with concrete types as needed
- Another popular form is **subtype** polymorphism (as in object-oriented programming)
- First introduced in SIMULA (first OO programming language)
- Example:
  When a function takes an argument of T it also works correctly if passed an argument of a type that is a subtype of T

Subtyping in Scala: Primitive Types

```scala
def func(n :Int) {
  val f: Float = n
  // ...
}
```

- The $\leq$ symbol is typically used to denote the subtype relationship

$$\texttt{Int} \leq \texttt{Float}$$

Subtyping in Scala: `traits`

```
trait List {
  def append(l: List): List
}
class Nil extends List {
  def append(ls: List): List = ls
}
class Cons(data: Int, tail: List) extends List {
  def append(ls: List): List =
           new Cons(data, tail.append(ls))
}
```

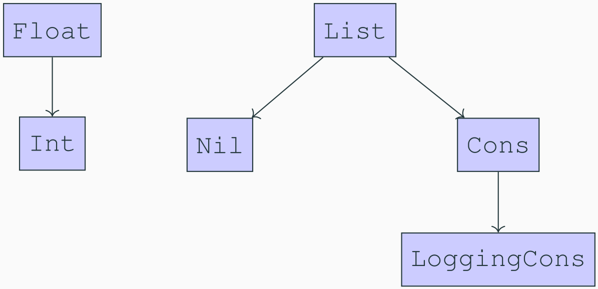
$$\texttt{Nil} \leq \texttt{List}$$
$$\texttt{Cons} \leq \texttt{List}$$

Subtyping in Scala: Inheritance

```scala
class Cons(data: Int, tail: List) extends List {
  /*...*/
}
class LoggingCons(d:Int,t:List) extends Cons(d,t) {
  private var numAppends: Int = 0
  override def append(ls: List): List = {
    numAppends = numAppends + 1
    super.append(ls)
  }
  def howManyAppends: Int = numAppends
}
```

$\boxed{\texttt{LoggingCons} \leq \texttt{Cons}}$

Subtyping as Graph Search



- Arrow from A to B to indicates that B is a direct subtype of A
- We can decide if B is a subtype of A using graph reachability

Subtyping Properties

Primitive rule:

$$\frac{}{\texttt{Int} \leq \texttt{Float}}$$

Reflexivity:

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

Transitivity:

$$\frac{\tau \leq \tau' \qquad \tau' \leq \tau''}{\tau \leq \tau''}$$

Subtyping Properties

$$\boxed{\tau_1 \leq \tau_2}$$

Subtyping preserves behavior

- **Liskov substitution principle:** values of type τ_2 may be replaced with values of type τ_1 without altering any of the desirable properties of τ_2

Subtyping preserves type safety

- Any expression of type τ_1 can be used in any context that expects an expression of type τ_2 and no type error will occur

Subtyping as Inclusion

- $\tau_1 \leq \tau_2$ means that a τ_1 can be used wherever a τ_2 is expected
- Any value of type τ_1 must also be a value of type τ_2
- Type interpretation $[\![\tau]\!]$ gives the set of elements of τ
    - $[\![\mathsf{Int}]\!] = \mathbb{Z}$
    - $[\![\mathsf{Float}]\!] = \mathbb{R}$
- $\tau_1 \leq \tau_2$ means $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$

Sanity-Checking the Principle

Primitive rule:

$$\frac{}{\texttt{Int} \leq \texttt{Float}}$$

✓ Any integer N can be treated as $N.0$ with no loss of meaning

Primitive rule:

$$\cancel{\frac{}{\texttt{Float} \leq \texttt{Int}}}$$

✗ e.g., the bitwise AND operator `&` defined for `Int` but not `Float`

Primitive rule:

$$\cancel{\frac{}{\texttt{Int} \leq \texttt{Int} \to \texttt{Int}}}$$

✗ Can't call an `Int`!

From Nominal to Structural

Inheritance: $$\frac{\texttt{class A extends B}}{\texttt{A} \leq \texttt{B}}$$

- This style of subtyping is called **nominal**:
  the edges between user-defined types are all declared explicitly, via the names of those types
- A **structural** subtyping system includes rules that analyze the structure of types, rather than just using graph edges declared by the user explicitly

Pair Types

- Consider types $\tau_1 \times \tau_2$ consisting of (immutable)pairs of a τ_1 and a τ_2
- What is a good subtyping rule for this feature?
- Ask ourselves: What operations does it support?
    1. Pull out a τ_1
    2. Pull out a τ_2

$$\frac{\tau_1 \leq \tau_1' \qquad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

- Jargon: The pair type constructor is **covariant**

Covariant, Contravariant, Invariant

If τ_1 is a subtype of τ_2 ($\tau_1 \leq \tau_2$) and f is a type transformation

- f is **covariant** if $\tau_1 \leq \tau_2$ implies that $f(\tau_1) \leq f(\tau_2)$
- f is **contravariant** if $\tau_1 \leq \tau_2$ implies that $f(\tau_2) \leq f(\tau_1)$
- f is **invariant** if neither of the above holds

Record Types

- A record consists of a set of labeled fields
- A record type includes the types of the fields in the record
- Define the type **Point** to be the record type

$$\textbf{Point} = \{x : \mathsf{Int}, y : \mathsf{Int}\}$$

- Define **Point3D** as the type of a record with three integer fields

$$\textbf{Point3D} = \{x : \mathsf{Int}, y : \mathsf{Int}, z : \mathsf{Int}\}$$

- Point3D contains all of the fields of Point,
  and those have the same type as in Point
- It makes sense to say that Point3D is a subtype of Point

$$\text{Point3D} \leq \text{Point}$$

Record Types

- Consider types like $\{a_1 : \tau_1, \cdots, a_n : \tau_n\}$, consisting of, for each i, a field a_i of type τ_i

Depth subtyping:

$$\frac{\forall i.\tau_i \leq \tau_i'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_i : \tau_i'\}}}$$

Same field names, possibly with different types

Width subtyping:

$$\frac{\forall j.\exists i.a_i = a_j' \land \tau_i = \tau_j'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_j' : \tau_j'\}}}$$

Field names may be different

Width subtyping

Width Subtyping

$$\frac{\forall j.\exists i.a_i = a'_j \land \tau_i = \tau'_j}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a'_j : \tau'_j\}}}$$

- Width subtyping allows forgetting fields on the right
- Intuition:
  x: `Int` is the type of all records with at least a numeric x field
- Record type with more fields is a subtype of
  record type with fewer fields
- Reason: the type with more fields places a stronger constraint on values, so it describes fewer values

Record Type Examples

$$\texttt{\{A: Int, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float, B : Float\}}$$

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Int, B : Float\}}$$

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float\}}$$

Depth subtyping: $$\frac{\forall i.\tau_i \leq \tau_i'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_i : \tau_i'\}}}$$

Width subtyping: $$\frac{\forall j.\exists i.a_i = a_j' \wedge \tau_i = \tau_j'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_j' : \tau_j'\}}}$$

Record Type Examples

$$\texttt{\{A: Int, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float, B : Float\}}$$

Yes

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Int, B : Float\}}$$

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float\}}$$

Depth subtyping: $$\frac{\forall i. \tau_i \leq \tau_i'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_i : \tau_i'\}}}$$

Width subtyping: $$\frac{\forall j. \exists i. a_i = a_j' \wedge \tau_i = \tau_j'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_j' : \tau_j'\}}}$$

Record Type Examples

$$\texttt{\{A: Int, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float, B : Float\}}$$

Yes

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Int, B : Float\}}$$

No

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float\}}$$

Depth subtyping: $$\frac{\forall i. \tau_i \leq \tau_i'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_i : \tau_i'\}}}$$

Width subtyping: $$\frac{\forall j. \exists i. a_i = a_j' \land \tau_i = \tau_j'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_j' : \tau_j'\}}}$$

Record Type Examples

$$\texttt{\{A: Int, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float, B : Float\}}$$

Yes

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Int, B : Float\}}$$

No

$$\texttt{\{A: Float, B: Float\}} \overset{?}{\leq} \texttt{\{A: Float\}}$$

Yes

Depth subtyping:

$$\frac{\forall i. \tau_i \leq \tau_i'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_i : \tau_i'\}}}$$

Width subtyping:

$$\frac{\forall j. \exists i. a_i = a_j' \wedge \tau_i = \tau_j'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_j' : \tau_j'\}}}$$

Exercise

Does the following subtyping relation hold?

`{x:{a:Int, b:Int}, y:{m:Int}}` $\stackrel{?}{\leq}$ `{x:{a:Int}}`

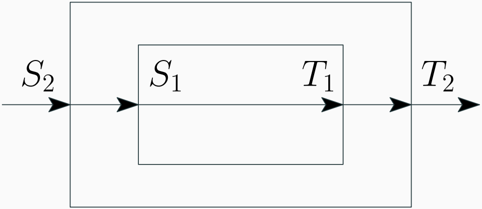
Depth subtyping: $$\frac{\forall i. \tau_i \leq \tau_i'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_i : \tau_i'\}}}$$

Width subtyping: $$\frac{\forall j. \exists i. a_i = a_j' \wedge \tau_i = \tau_j'}{\overrightarrow{\{a_i : \tau_i\}} \leq \overrightarrow{\{a_j' : \tau_j'\}}}$$

Subtyping for Function Types

- Function type $\to$ is covariant in the result type and contravariant in the argument type

$$\frac{\begin{array}{c} S_2 \leq S_1 \\ T_1 \leq T_2 \end{array}}{S_1 \to T_1 \;\leq\; S_2 \to T_2}$$



Arrays

Consider types $\tau[\,]$

What operations must we support?

1. Read a τ from some index
2. Write a τ to some index

Covariant rule:

$$\frac{\tau_1 \leq \tau_2}{\tau_1[\,] \leq \tau_2[\,]}$$

Counterexample:

```
String[] strings = new String[1];
Object[] objects = strings;
objects[0] = new Integer(0);
```

Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer

Arrays

Consider types $\tau[\,]$

What operations must we support?

1. Read a τ from some index
2. Write a τ to some index

Contravariant rule:

$$\frac{\tau_2 \leq \tau_1}{\tau_1[\,] \leq \tau_2[\,]}$$

Counterexample:

```
float[] x = new float[1];
int[] y = x; // Use subtyping here.
x[0] = 1.23;
int z = y[0]; // Not an int!
```

Arrays

Consider types $\tau[\,]$

What operations must we support?

1. Read a τ from some index
2. Write a τ to some index

- Correct rule: Array type constructor is invariant
- Only reflexivity applies to array types
- Java and many other "practical" languages use the covariant rule
- Run-time type errors (exceptions) are possible!