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

```
Int \leq \text{Float}
```
```
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))
}

Nil ⊆ List
Cons ⊆ List
```
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
}

LoggingCons ≤ 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:

Int ≤ Float

Reflexivity:

τ ≤ τ

Transitivity:

\( \tau \leq \tau' \quad \tau' \leq \tau'' \)

\( \tau \leq \tau'' \)
Subtyping preserves behavior

- **Liskov substitution principle**: values of type $\tau_2$ may be replaced with values of type $\tau_1$ without altering any of the desirable properties of $\tau_2$

Subtyping preserves type safety

- Any expression of type $\tau_1$ can be used in any context that expects an expression of type $\tau_2$ and no type error will occur
• $\tau_1 \leq \tau_2$ means that a $\tau_1$ can be used wherever a $\tau_2$ is expected.
• Any value of type $\tau_1$ must also be a value of type $\tau_2$.
• Type interpretation $\llbracket \tau \rrbracket$ gives the set of elements of $\tau$.
  • $\llbracket \text{Int} \rrbracket = \mathbb{Z}$
  • $\llbracket \text{Float} \rrbracket = \mathbb{R}$
• $\tau_1 \leq \tau_2$ means $\llbracket \tau_1 \rrbracket \subseteq \llbracket \tau_2 \rrbracket$. 
Sanity-Checking the Principle

Primitive rule: $\underline{\text{Int} \leq \text{Float}}$

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

Primitive rule: $\underline{\text{Float} \leq \text{Int}}$

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

Primitive rule: $\underline{\text{Int} \leq \text{Int} \rightarrow \text{Int}}$

✗ Can’t call an Int!
Inheritance: \[
\text{class } A \text{ extends } B \\
A \leq B
\]

- 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.
• Consider types $\tau_1 \times \tau_2$ consisting of (immutable)pairs of a $\tau_1$ and a $\tau_2$

• What is a good subtyping rule for this feature?

• Ask ourselves: What operations does it support?
   1. Pull out a $\tau_1$
   2. Pull out a $\tau_2$

$$
\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \times \tau_2 \leq \tau'_1 \times \tau'_2}
$$

• Jargon: The pair type constructor is **covariant**
Covariant, Contravariant, Invariant

If $\tau_1$ is a subtype of $\tau_2$ ($\tau_1 \leq \tau_2$) 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
  \[
  \text{Point} = \{x : \text{Int}, y : \text{Int}\}
  \]
- Define `Point3D` as the type of a record with three integer fields
  \[
  \text{Point3D} = \{x : \text{Int}, y : \text{Int}, z : \text{Int}\}
  \]
- `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 \( \tau_i \)

Depth subtyping:

\[
\forall i. \tau_i \leq \tau'_i \quad \Rightarrow \quad \{ a_i : \tau_i \} \leq \{ a_i : \tau'_i \}
\]

Same field names, possibly with different types

Width subtyping:

\[
\forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \quad \Rightarrow \quad \{ a_i : \tau_i \} \leq \{ a'_j : \tau'_j \}
\]

Field names may be different
Width subtyping

\[ \forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \]

\[ \{ a_i : \tau_i \} \leq \{ a'_j : \tau'_j \} \]

- Width subtyping allows forgetting fields on the right
- Intuition:
  \( x : \text{Int} \) 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

\{A: \text{Int}, B: \text{Float}\} \leq \{A: \text{Float}, B: \text{Float}\}

\{A: \text{Float}, B: \text{Float}\} \leq \{A: \text{Int}, B: \text{Float}\}

\{A: \text{Float}, B: \text{Float}\} \leq \{A: \text{Float}\}

Depth subtyping:

$$\forall i. \tau_i \leq \tau'_i \quad \Rightarrow \quad \{a_i : \tau_i\} \leq \{a_i : \tau'_i\}$$

Width subtyping:

$$\forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \quad \Rightarrow \quad \{a_i : \tau_i\} \leq \{a'_j : \tau'_j\}$$
Record Type Examples

\{A: \text{Int}, B: \text{Float}\} \leq \{A: \text{Float}, B: \text{Float}\}

\{A: \text{Float}, B: \text{Float}\} \leq \{A: \text{Int}, B: \text{Float}\}

\{A: \text{Float}, B: \text{Float}\} \leq \{A: \text{Float}\}

Depth subtyping: \[ \forall i. \tau_i \leq \tau'_i \]
\[ \{a_i : \tau_i\} \leq \{a_i : \tau'_i\} \]

Width subtyping: \[ \forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \]
\[ \{a_i : \tau_i\} \leq \{a'_j : \tau'_j\} \]
Record Type Examples

\{A: \text{Int}, B: \text{Float}\} \not\leq \{A: \text{Float}, B: \text{Float}\}

\{A: \text{Float}, B: \text{Float}\} \not\leq \{A: \text{Int}, B: \text{Float}\}

\{A: \text{Float}, B: \text{Float}\} \not\leq \{A: \text{Float}\}

Depth subtyping:

\forall i. \tau_i \leq \tau'_i \\
\{a_i : \tau_i\} \leq \{a_i : \tau'_i\}

Width subtyping:

\forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \\
\{a_i : \tau_i\} \leq \{a'_j : \tau'_j\}
Record Type Examples

{A: Int, B: Float} \leq \{A: Float, B: Float\}

{A: Float, B: Float} \leq \{A: Int, B: Float\}

{A: Float, B: Float} \leq \{A: Float\}

Depth subtyping:

\( \forall i. \tau_i \leq \tau'_i \)
\( \{a_i : \tau_i\} \leq \{a_i : \tau'_i\} \)

Width subtyping:

\( \forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \)
\( \{a_i : \tau_i\} \leq \{a'_j : \tau'_j\} \)
Exercise

Does the following subtyping relation hold?

\[
\{x: \{a: \text{Int}, b: \text{Int}\}, y: \{m: \text{Int}\}\} \leq \{x: \{a: \text{Int}\}\}
\]

Depth subtyping:

\[
\forall i. \tau_i \leq \tau'_i \quad \Rightarrow \quad \{a_i : \tau_i\} \leq \{a_i : \tau'_i\}
\]

Width subtyping:

\[
\forall j. \exists i. a_i = a'_j \land \tau_i = \tau'_j \quad \Rightarrow \quad \{a_i : \tau_i\} \leq \{a'_j : \tau'_j\}
\]
Function type $\rightarrow$ is covariant in the result type and contravariant in the argument type

\[
\begin{align*}
S_2 &\leq S_1 \\
T_1 &\leq T_2 \\
\implies S_1 \rightarrow T_1 &\leq S_2 \rightarrow T_2
\end{align*}
\]
Consider types $\tau[\ ]$
What operations must we support?

1. Read a $\tau$ from some index
2. Write a $\tau$ 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 $\tau$ from some index
2. Write a $\tau$ to some index

Contravariant rule: 

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

Counterexample:

```java
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 $\tau$ from some index
2. Write a $\tau$ 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!