Arrays

Using array as an expression, on the right-hand side

$$\frac{\Gamma \vdash \mathrm{a}:\ \mathrm{Array(T)} \qquad \Gamma \vdash \mathrm{i}:\ \mathrm{Int}}{\Gamma \vdash \mathrm{a[i]:T}}$$

Assigning to an array

$$\frac{\Gamma \vdash \mathrm{a}:\ \mathrm{Array(T)} \qquad \Gamma \vdash \mathrm{i}:\ \mathrm{Int} \qquad \Gamma \vdash \mathrm{e}:\ \mathrm{T}}{\Gamma \vdash (\mathrm{a[i]}\ = \mathrm{e}):\ \mathrm{void}}$$

Example with Arrays

```
def next(a : Array[Int], k : Int) : Int = {
    a[k] = a[a[k]]
}
```

Given Γ = {(a, Array(Int)), (k, Int)}, check $\Gamma \vdash$ a[k] = a[a[k]]: void

$$\dfrac{\Gamma \vdash \mathrm{a{:}\ Array(Int)} \qquad \dfrac{\Gamma \vdash \mathrm{a{:}\ Array(Int)} \qquad \Gamma \vdash \mathrm{k{:}\ Int}}{\Gamma \vdash \mathrm{a[k]{:}\ Int}}}{\Gamma \vdash \mathrm{a[a[k]]{:}Int}} \qquad \Gamma \vdash \mathrm{a{:}\ Array(Int)} \qquad \Gamma \vdash \mathrm{k{:}Int}$$
$$\overline{\Gamma \vdash \mathrm{a[k]} = \mathrm{a[a[k]]}{:}\ \text{void}}$$

Type Rules (1)

$$\frac{(\mathrm{x}\colon\ \mathrm{T}) \in \Gamma}{\Gamma \vdash \mathrm{x}\colon\ \mathrm{T}}$$ variable

$$\frac{}{\mathrm{IntConst(k)}\colon\ \mathrm{Int}}$$ constant

$$\frac{\Gamma \vdash e_1 : T_1 \ \ldots\ \Gamma \vdash e_n : T_n \qquad \Gamma \vdash f : (T_1 \times \cdots \times T_n \to T)}{\Gamma \vdash f(e_1, \ldots, e_n) : T}$$ function application

$$\frac{\Gamma \vdash e_1\colon\ \mathrm{Int} \qquad \Gamma \vdash e_2\colon\ \mathrm{Int}}{\Gamma \vdash (e_1 + e_2)\colon\ \mathrm{Int}}$$ plus $$\frac{\Gamma \vdash e_1\colon\ \mathrm{String} \qquad \Gamma \vdash e_2\colon\ \mathrm{String}}{\Gamma \vdash (e_1 + e_2)\colon\ \mathrm{String}}$$

$$\frac{\Gamma \vdash \mathrm{b}\colon\ \mathrm{Boolean} \qquad \Gamma \vdash e_1 : \mathrm{T} \qquad \Gamma \vdash e_2 : \mathrm{T}}{\Gamma \vdash (\mathrm{if(b)}\ e_1\ \mathrm{else}\ e_2) : \mathrm{T}}$$ if

$$\frac{\Gamma \vdash \mathrm{b}\colon\ \mathrm{Boolean} \qquad \Gamma \vdash \mathrm{s}\colon\ \mathrm{void}}{\Gamma \vdash (\mathrm{while(b)\ s})\colon\ \mathrm{void}}$$ $$\frac{(\mathrm{x},\ \mathrm{T}) \in \Gamma \qquad \Gamma \vdash \mathrm{e}\colon\ \mathrm{T}}{\Gamma \vdash (\mathrm{x{=}e})\colon\ \mathrm{void}}$$

while assignment

Type Rules (2)

$$\frac{\Gamma \vdash \text{e: T}}{\Gamma \vdash \{\text{e}\}\text{: T}} \qquad \frac{}{\Gamma \vdash \{\}\text{: void}}$$

$$\frac{\Gamma \oplus \{(x, T_1)\} \vdash \{t_2; \ \ldots \ ; t_n\}\text{: T}}{\Gamma \vdash \{\text{var } x : T_1; t_2; \ \ldots \ ; t_n\}\text{: T}}$$

$$\frac{\Gamma \vdash s_1\text{: void} \qquad \Gamma \vdash \{t_2; \ \ldots \ ; t_n\}\text{: T}}{\Gamma \vdash \{s_1; t_2; \ \ldots \ ; t_n\}\text{: T}}$$

block

$$\frac{\Gamma \vdash \text{a: Array(T)} \qquad \Gamma \vdash \text{i: Int}}{\Gamma \vdash \text{a[i]: T}}$$

array use

$$\frac{\Gamma \vdash \text{a: Array(T)} \qquad \Gamma \vdash \text{i: Int} \qquad \Gamma \vdash \text{e: T}}{\Gamma \vdash \text{a[i]} = \text{e}}$$

array assignment

Type Rules (3)

Γ^c - top-level environment of class C

```
class C {
  var x: Int;
  def m(p: Int): Boolean = {…}
}
```

Γ^c = { (x, Int), (m, C x Int $\rightarrow$ Boolean)}

$$\frac{\Gamma \vdash e : C \quad \Gamma^C \vdash m : \mathsf{C} \times T_1 \times \ \ldots \ \times T_n \rightarrow T_{n+1} \qquad \Gamma \vdash e_i : T_i \quad 1 \leq i \leq n}{\Gamma \vdash e.m(e_1, \ \ldots \ , e_n) : T_{n+1}}$$

method invocation

$$\frac{\Gamma \vdash \mathrm{e{:}}\ \mathrm{C} \qquad \Gamma^C \vdash \mathrm{f{:}}\ \mathrm{T}}{\Gamma \vdash \mathrm{e.f{:}}\ \mathrm{T}}$$

field use

$$\frac{\Gamma \vdash \mathrm{e{:}}\ \mathrm{C} \qquad \Gamma^C \vdash \mathrm{f{:}}\ \mathrm{T} \qquad \Gamma \vdash \mathrm{x{:}}\ \mathrm{T}}{\Gamma \vdash (\mathrm{e.f} = \mathrm{x}){:}\ \mathrm{void}}$$

field assignment

Does this program type check?

```
class Rectangle {
  var width: Int
  var height: Int
  var xPos: Int
  var yPos: Int
  def area(): Int = {
    if (width > 0 && height > 0)
     width * height
   else 0
  }
  def resize(maxSize: Int) {
    while (area > maxSize) {
      width = width / 2
      height = height / 2
    }
  }
}
```

$$\Gamma_0 = \left\{ \begin{array}{c} \text{w: Int, h: Int,} \\ \text{x: Int, y: Int,} \\ area : \text{Unit} \rightarrow \text{Int,} \\ resize : \text{Int} \rightarrow \text{Unit} \end{array} \right\}$$

Type check: area

Type check: resize

Semantics of Types

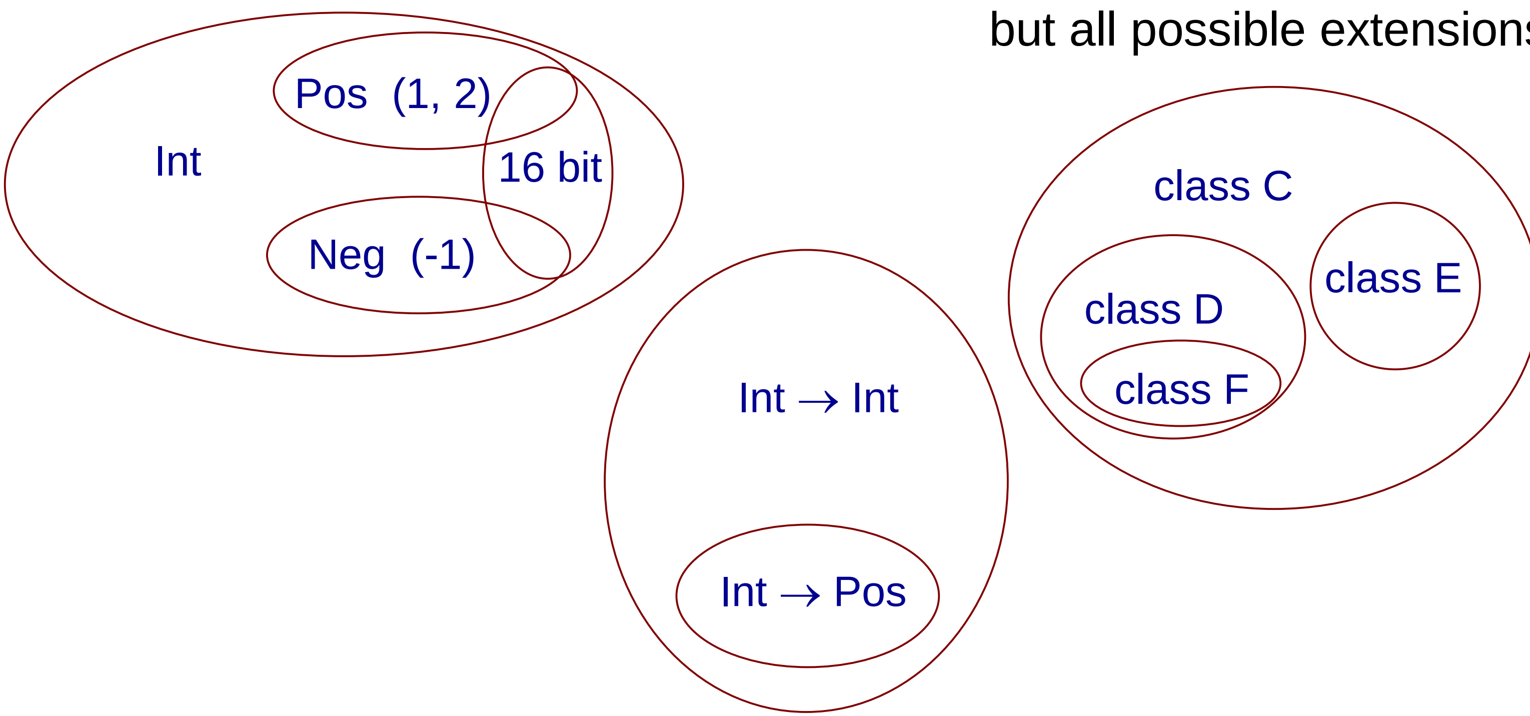
- Operational view: Types are named entities
  - such as the primitive types (Int, Bool etc.) and explicitly declared classes, traits …
  - their meaning is given by methods they have
  - constructs such as inheritance establish relationships between classes
- Mathematically, Types are sets of values
  - Int = { ..., -2, -1, 0, 1, 2, ... }
  - Boolean = { false, true }
  - Int → Int = { f : Int -> Int | f is computable }

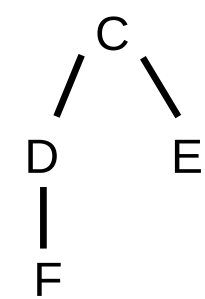
Types as Sets

- Sets so far were disjoint

- Sets can overlap

C represents not only declared C, but all possible extensions as well





F extends D,
D extends C

SUBTYPING

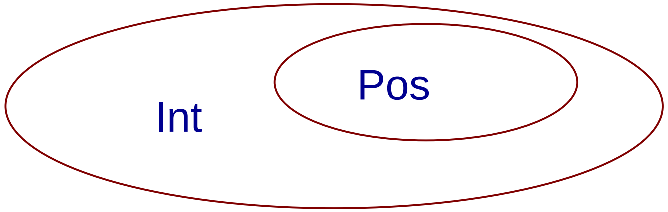
Subtyping

- Subtyping corresponds to subset
- Systems with subtyping have non-disjoint sets
- T_1 <: T_2 means T_1 is a subtype of T_2
  - corresponds to $T_1 \subseteq T_2$ in sets of values
- Rule for subtyping: analogous to set reasoning

In terms of sets

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

$$\frac{e \in T_1 \qquad T_1 \subseteq T_2}{e \in T_2}$$



Types for Positive and Negative Ints

Int = { ... , -2, -1, 0, 1, 2, ... }

Pos = { 1, 2, ... } (not including zero)

Neg = { ..., -2, -1 } (not including zero)

types:

Pos <: Int

Neg <: Int

$$\frac{\Gamma \vdash \mathrm{x}\text{: Pos} \qquad \Gamma \vdash \mathrm{y}\text{: Pos}}{\Gamma \vdash \mathrm{x} + \mathrm{y}\text{: Pos}}$$

$$\frac{\Gamma \vdash \mathrm{x}\text{: Pos} \qquad \Gamma \vdash \mathrm{y}\text{: Neg}}{\Gamma \vdash \mathrm{x} * \mathrm{y}\text{: Neg}}$$

$$\frac{\Gamma \vdash \mathrm{x}\text{: Pos} \qquad \Gamma \vdash \mathrm{y}\text{: Pos}}{\Gamma \vdash \mathrm{x} / \mathrm{y}\text{: Pos}}$$

sets:

Pos $\subseteq$ Int

Neg $\subseteq$ Int

$$\frac{\mathrm{x} \in \mathrm{Pos} \qquad \mathrm{y} \in \mathrm{Pos}}{\mathrm{x} + \mathrm{y} \in \mathrm{Pos}}$$

$$\frac{\mathrm{x} \in \mathrm{Pos} \qquad \mathrm{y} \in \mathrm{Neg}}{\mathrm{x} * \mathrm{y} \in \mathrm{Neg}}$$

$$\frac{\mathrm{x} \in \mathrm{Pos} \qquad \mathrm{y} \in \mathrm{Pos}}{\mathrm{x} / \mathrm{y} \in \mathrm{Pos}}$$

(y not zero)

(x/y well defined)

Rules for Neg, Pos, Int

$$\frac{\Gamma \vdash x : \mathrm{Pos} \quad \Gamma \vdash y : \mathrm{Neg}}{\Gamma \vdash x + y : ???}$$

$$\frac{\Gamma \vdash x : \mathrm{Pos} \quad \Gamma \vdash y : \mathrm{Neg}}{\Gamma \vdash x * y : ???}$$

$$\frac{\Gamma \vdash x : \mathrm{Pos} \quad \Gamma \vdash y : \mathrm{Int}}{\Gamma \vdash x + y : ???}$$

$$\frac{\Gamma \vdash x : \mathrm{Pos} \quad \Gamma \vdash y : \mathrm{Int}}{\Gamma \vdash x * y : ???}$$

More Rules

$$\frac{\Gamma \vdash \text{x: Neg} \qquad \Gamma \vdash \text{y: Neg}}{\Gamma \vdash \text{x * y: Pos}}$$

$$\frac{\Gamma \vdash \text{x: Neg} \qquad \Gamma \vdash \text{y: Neg}}{\Gamma \vdash \text{x + y: Neg}}$$

More rules for division?

$$\frac{\Gamma \vdash \text{x: Neg} \qquad \Gamma \vdash \text{y: Neg}}{\Gamma \vdash \text{x / y: Pos}}$$

$$\frac{\Gamma \vdash \text{x: Pos} \qquad \Gamma \vdash \text{y: Neg}}{\Gamma \vdash \text{x / y: Neg}}$$

$$\frac{\Gamma \vdash \text{x: Int} \qquad \Gamma \vdash \text{y: Neg}}{\Gamma \vdash \text{x / y: Int}}$$

Making Rules Useful

- Let x be a variable

$$\frac{\Gamma \vdash \mathrm{x}\colon\ \mathrm{Int} \qquad \Gamma \oplus \{(x, Pos)\} \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (\mathrm{if}\ (\mathrm{x} > 0)\ e_1\ \mathrm{else}\ e_2)\colon\ \mathrm{T}}$$

$$\frac{\Gamma \vdash \mathrm{x}\colon\ \mathrm{Int} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \oplus \{(x, Neg)\} \vdash e_2 : T}{\Gamma \vdash (\mathrm{if}\ (\mathrm{x} >= 0)\ e_1\ \mathrm{else}\ e_2)\colon\ \mathrm{T}}$$

```
var x : Int
var y : Int
if (y > 0) {
  if (x > 0) {
    var z : Pos = x * y
    res = 10 / z
} }
```

type system proves: no division by zero

Subtyping Example

```
def f(x:Int) : Pos = {
  if (x < 0) -x else x+1
}
var p : Pos
var q : Int
q = f(p)
```

Does this statement type check?

Given:

$$\text{Pos} <: \text{Int}$$

$$\Gamma \vdash f: \text{Int} \to \text{Pos}$$

$$\dfrac{(q, \text{Int}) \in \Gamma \qquad \dfrac{\dfrac{\dfrac{p: \text{Pos} \qquad \text{Pos} <: \text{Int}}{p: \text{Int}} \qquad f: \text{Int} \to \text{Pos}}{f(p): \text{Pos}} \qquad \text{Pos} <: \text{Int}}{f(p): \text{Int}}}{q{=}f(p): \text{void}}$$

Subtyping Example

```
def f(x:Pos) : Pos = {
  if (x < 0) -x else x+1
}
var p : Int
var q : Int
q = f(p)
```

Does this statement type check?

does not type check

What Pos/Neg Types Can Do

```
def multiplyFractions(p1 : Int, q1 : Pos, p2 : Int, q2 : Pos) : (Int,Pos) {
  (p1*q1, q1*q2)
}
def addFractions(p1 : Int, q1 : Pos, p2 : Int, q2 : Pos) : (Int,Pos) {
  (p1*q2 + p2*q1, q1*q2)
}
def printApproxValue(p : Int, q : Pos) = {
  print(p/q)   // no division by zero
}
```

More sophisticated types can track intervals of numbers and ensure that a program does not crash with an array out of bounds error.

Subtyping and Product Types

Subtyping for Products

T_1 <: T_2 implies for all e:

$$\frac{\Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

Type for a tuple:

$$\frac{x : T_1 \qquad y : T_2}{(x, y) : T_1 \times T_2}$$

$$\dfrac{\dfrac{x : T_1 \qquad T_1 <: T_1'}{x : T_1'} \qquad \dfrac{y : T_2 \qquad T_2 <: T_2'}{y : T_2'}}{(x, y) : T_1' \times T_2'}$$

So, we might as well add:

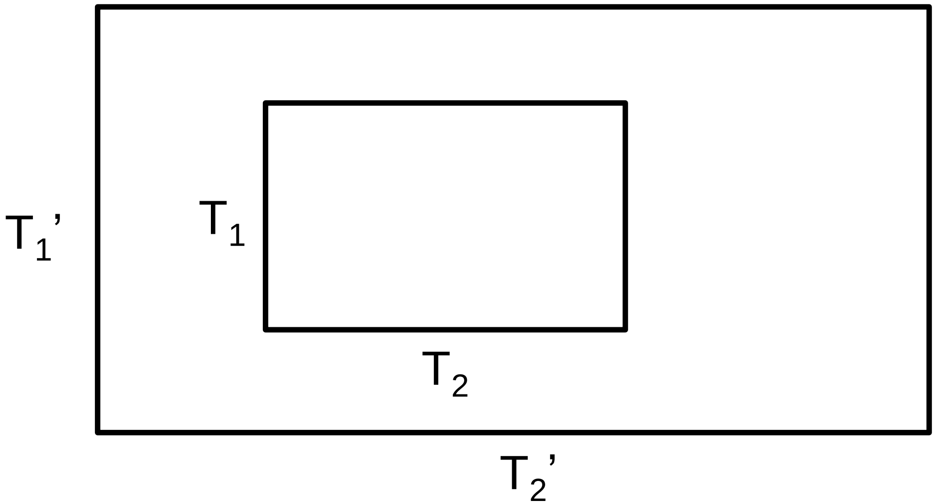
$$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 \times T_2 <: T_1' \times T_2'}$$

covariant subtyping for pair types denoted (T_1, T_2) or Pair[T_1, T_2]

Analogy with Cartesian Product

$$\frac{T_1 <: T_1' \qquad T_2 <: T_2'}{T_1 \times T_2 <: T_1' \times T_2'}$$

$$\frac{T_1 \subseteq T_1' \qquad T_2 \subseteq T_2'}{T_1 \times T_2 \subseteq T_1' \times T_2'}$$



A x B = { (a, b) | a $\in$ A, b $\in$ B}

Subtyping and Function Types

Subtyping for Function Types

$T_1 <: T_2$ implies for all e:

$$\frac{\Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

$$\frac{\overbrace{T_1' <: T_1 \quad \ldots \quad T_n' <: T_n}^{\text{contravariance}} \qquad \overbrace{T <: T'}^{\text{covariance}}}{(T_1 \times \ \ldots \ \times T_n \to T) <: (T_1' \times \ \ldots \ \times T_n' \to T')}$$

Consequence:

$$\dfrac{\dfrac{\Gamma \vdash m : T_1 \times \ \ldots \ \times T_n \to T \qquad \dfrac{\Gamma \vdash e_1 : T_1' \qquad T_1' <: T_1}{\Gamma \vdash e_1 : T_1} \qquad \dfrac{\Gamma \vdash e_n : T_n' \qquad T_n' <: T_n}{\Gamma \vdash e_n : T_n}}{\Gamma \vdash m(e_1, \ \ldots \ , e_n) : T} \qquad T <: T'}{\Gamma \vdash m(e_1, \ \ldots \ , e_n) : T'}$$

as if Γ |- m: $T'_1 \times \ldots \times T_n' \to T'$

Function Space as Set

A function type is a set of functions (function space) defined as follows:

$$T_1 \to T_2 = \{ f \mid \forall x. (x \in T_1 \to f(x) \in T_2)\}$$

contravariance because
$x \in T_1$ is left of implication

We can prove

$$\frac{T_1' \subseteq T_1 \qquad T_2 \subseteq T_2'}{T_1 \to T_2 \subseteq T_1' \to T_2'}$$

Proof

$T_1 \rightarrow T_2 = \{ f \mid \forall x. (x \in T_1 \rightarrow f(x) \in T_2)\}$

$$\frac{T_1' \subseteq T_1 \qquad T_2 \subseteq T_2'}{T_1 \rightarrow T_2 \subseteq T_1' \rightarrow T_2'}$$

Subtyping for Classes

- Class C contains a collection of methods

- For class sub-typing, we require that methods named the same are subtypes

Example

class C {
  def m(x : T_1) : T_2 = {...}
}
class D **extends** C {
  override def m(x : T'_1) : T'_2 = {...}
}

D <: C so need to have $(T'_1 \to T'_2) <: (T_1 \to T_2)$

Therefore, we need to have:

$T'_2 <: T_2$ (result behaves like the class)

$T_1 <: T'_1$ (**argument behaves opposite**)

Mutable and Immutable Fields

- We view field var f: T as two methods
  - getF : T   T
  - setF(x:T): void   T → void
- For val f: T (immutable): we have only getF

Could we allow this?

class A {}   **class** B **extends** A {...}   B <: A

```
class C {
  val x : A = ...
}
class D extends C {
  override val x : B = ...
}
```

Because B <: A, this is a valid way for D to extend C (D <: C)

Substitution principle:

If someone uses z:D thinking it is z:C, the fact that they read z.x and obtain B as a specific kind of A is not a problem.

What if x is a **var** ?

class A {} **class** B **extends** A {...} B <: A

class C {
  var x : A = ...
}
class D **extends** C {
  override var x : B = ... **?!?**
}

If we now imagine the setter method (i.e. field assignment), in the first case the setter has type, for D <: C

- B <: A, because of setter (reading values)
- (B -> void) <: (A -> void), so by contravariance A <: B
- Thus A=B

Soundness of Types

ensuring that a type system
is not broken
For every program and every input,
if it type checks, it does not break.