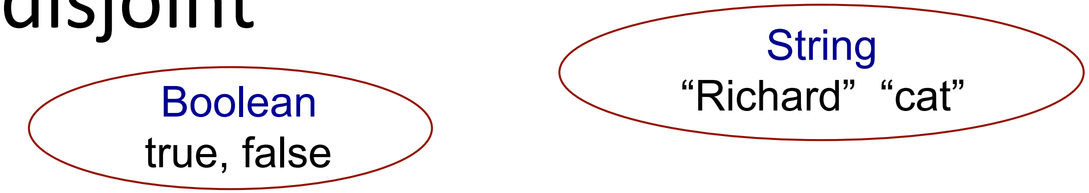
Meaning of Types

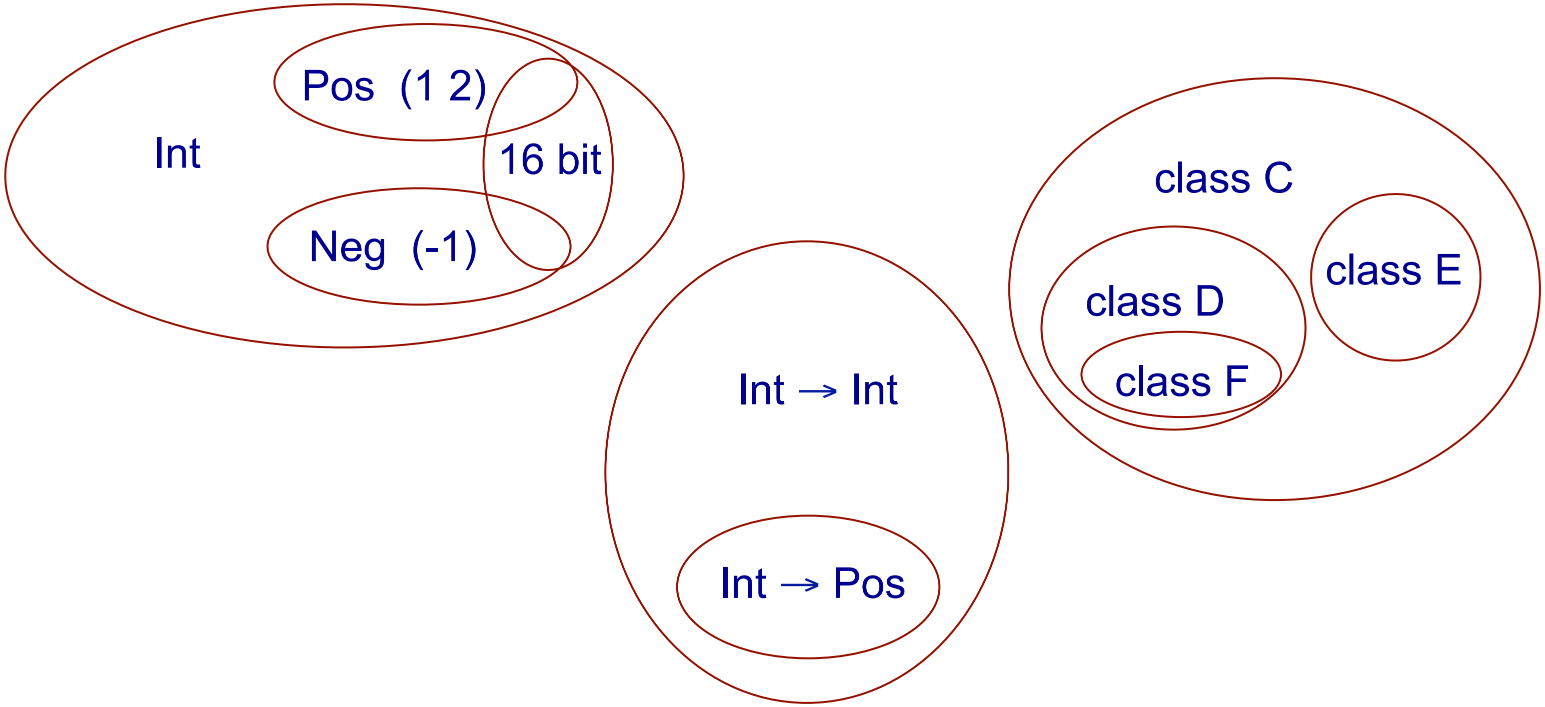
- Types can be viewed as named entities
  - explicitly declared classes, traits
  - their meaning is given by methods they have
  - constructs such as inheritance establish relationships between classes
- Types can be viewed as sets of values
  - Int = { ..., -2, -1, 0, 1, 2, ... }
  - Boolean = { false, true }
  - Int → Int = { f : Int -> Int | f is computable }

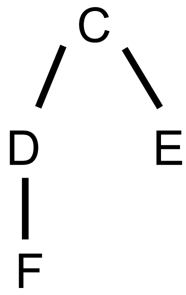
Types as Sets

- Sets so far were disjoint



- Sets can overlap





F extends D,
D extends C

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
- Main rule for subtyping ≈ corresponds to

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

Types for Positive and Negative Ints

Int = { … , -2, -1, 0, 1, 2, … }

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

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

Pos <: Int

Neg <: Int

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

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

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

Pos $\subseteq$ Int

Neg $\subseteq$ Int

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

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

(y not zero)

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

(x/y well defined)

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

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

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

```
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?

Pos <: Int

Γ :   f: Int → Pos

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

Using Subtyping

```
def f(x:Pos) : Pos = {
  if (x < 0) –x else x+1
}

var p : Int
var q : Int

q = f(p)
```

Pos <: Int

Γ :   f: Pos → Pos

- 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

Using Subtyping

```
def f(x:Pos) : Pos = {
  if (x < 0) —x else x+1
}

var p : Int
var q : Int

q = f(p)
```

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

$\Gamma: \quad \text{f: Pos} \rightarrow \text{Pos}$

- does not type check

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

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

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

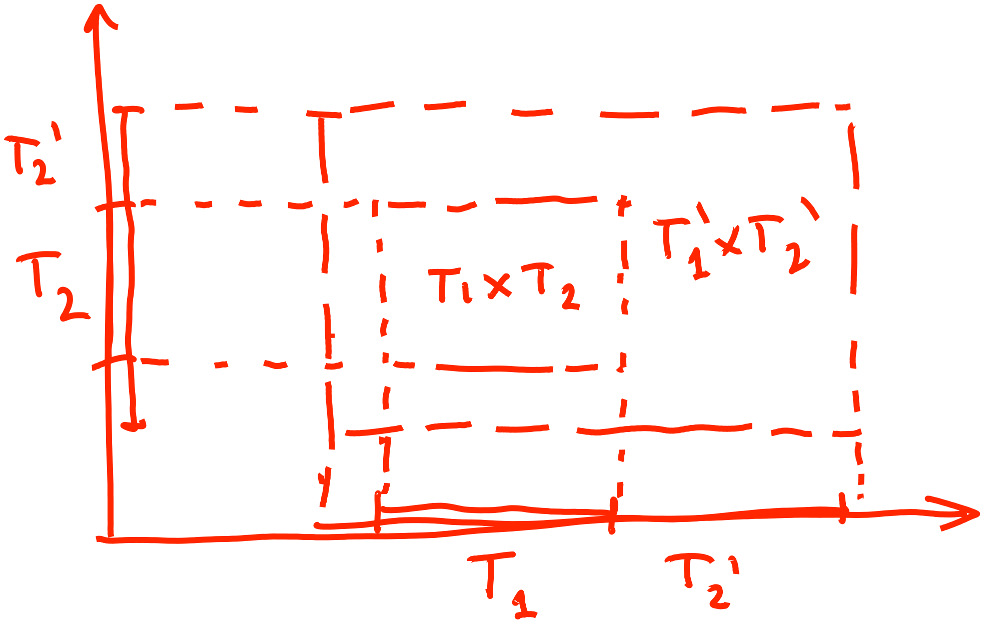
So, we might as well add:

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

covariant subtyping for pairs
Pair [T_1, T_2]

Analogy with Cartesian Product

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



A × B = { (a, b) | a ∈ A, b ∈ B}

Subtyping and Function Types

Subtyping for Function Types

when: $T_0 \to T_R \quad <: \quad T_0' \to T_R'$?

$$T <: T' \quad \underset{\text{ideally}}{\overset{\text{implies}}{\rightleftarrows}} \quad \text{for all e:} \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash e : T'}$$

Suppose: $T_R <: T_R' \qquad T_0' <: T_0$

then:

$$\dfrac{\dfrac{\Gamma \vdash f : T_0 \to T_R \qquad \dfrac{\Gamma \vdash x : T_0'}{\Gamma \vdash x : T_0}}{\Gamma \vdash f(x) : T_R}}{\Gamma \vdash f(x) : T_R'}$$

Subtyping for Function Types

when: $T_0 \to T_R$ <: $T_0' \to T_R'$?

T <: T' $\underset{\text{ideally}}{\overset{\text{implies}}{\rightleftarrows}}$ for all e: $\dfrac{\Gamma \vdash e : T}{\Gamma \vdash e : T'}$

Suppose: $T_R <: T_R'$ $\qquad T_0' <: T_0$

then:

$$\dfrac{\dfrac{\Gamma \vdash f : T_0 \to T_R \qquad \dfrac{\Gamma \vdash x : T_0'}{\Gamma \vdash x : T_0}}{\Gamma \vdash f(x) : T_R}}{\Gamma \vdash f(x) : T_R'}$$

as if
$\Gamma \vdash$ f: $T_0' \to T_R'$

Subtyping for Function Types

when: $T_0 \to T_R \;<:\; T_0' \to T_R'$?

$$T <: T' \;\underset{\text{ideally}}{\overset{\text{implies}}{\rightleftarrows}}\; \text{for all } e\text{:} \quad \frac{\Gamma \vdash e : T}{\Gamma \vdash e : T'}$$

Suppose:

$$\boxed{\frac{T_R <: T_R' \qquad T_0' <: T_0}{T_0 \to T_R <: T_0' \to T_R'}}$$

then:

→

$$\cfrac{\cfrac{\Gamma \vdash f : T_0 \to T_R \qquad \cfrac{\Gamma \vdash x : T_0'}{\Gamma \vdash x : T_0}}{\Gamma \vdash f(x) : T_R}}{\Gamma \vdash f(x) : T_R'}$$

as if

$\Gamma \vdash f\colon T_0' \to T_R'$

Function Space as Set

To get the appropriate behavior we need to assign sets to function types like this:

$(\neg\, x \in T_1) \vee f(x) \in T_2$

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

$\neq T_1 \times T_2$

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

We can prove

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

Proof

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

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

- Let $T_1' \subseteq T_1$ and $T_2 \subseteq T_2'$ and $f \in T_1 \rightarrow T_2$
- $\forall$ x. $x \in T_1 \rightarrow f(x) \in T_2$
- Let $x \in T_1'$. From $T_1' \subseteq T_2$, also $x \in T_1$
- $f(x) \in T_2$. By $T_2 \subseteq T_2'$, also $f(x) \in T_s'$
- $\forall$ x. $x \in T_1' \rightarrow f(x) \in T_2'$
- Therefore, $f \in t_1' \rightarrow T_2'$
- Thus, $T_1 \rightarrow T_2 \subseteq T_1' \rightarrow T_2'$

Subtyping for Classes

- Class C contains a collection of methods
- We view field var f: T as two methods
  - getF(this:C): T   C → T
  - setF(this:C, x:T): void   C x T → void
- For val f: T (immutable): we have only getF
- Class has all functionality of a pair of method
- We must require (at least) that methods named the same are subtypes
- If type T is generic, it must be invariant
  - as for mutable arrays

Example

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

$D <: C$

Therefore, we need to have:

- $T_1 <: T'_1$ (argument behaves opposite)
- $T'_2 <: T_2$ (result behaves like class)

Today

- More Subtyping Rules
  - product types (pairs) ✓
  - function types ✓
  - classes ✓
- Soundness ←
  - motivating example
  - idea of proving soundness
  - operational semantics
  - a soundness proof
- Subtyping and generics

Example: *Tootool 0.1* Language

Tootool is a rural community in the central east part of the Riverina [New South Wales, Australia]. It is situated by road, about 4 kilometres east from French Park and 16 kilometres west from The Rock.
Tootool Post Office opened on 1 August 1901 and closed in 1966. [Wikipedia]

unsound

Type System for *Tootool 0.1*

Pos <: Int

Neg <: Int

$$\frac{\Gamma \vdash x:\ T \qquad \Gamma \vdash e:\ T}{\Gamma \vdash (x = e):\ \text{void}}$$

assignment

$$\frac{\Gamma \vdash e:\ T \qquad \Gamma \vdash T <:\ T'}{\Gamma \vdash e:\ T'}$$

subtyping

does it type check?

```
def intSqrt(x:Pos) : Pos = { ...}
var p : Pos
var q : Neg
var r : Pos
q = -5
p = q
r = intSqrt(p)
```

Γ = {(p, Pos), (q, Neg), (r, Pos), (intSqrt, Pos → Pos)}

Runtime error: intSqrt invoked with a negative argument!

$$\frac{\dfrac{p:\ \text{Pos} \qquad \text{Pos} <:\ \text{Int}}{p:\ \text{Int}} \qquad \dfrac{q:\ \text{Neg} \qquad \text{Neg} <:\ \text{Int}}{q:\ \text{Int}}}{(p{=}q):\ \text{void}}$$

What went wrong in *Tootool 0.1* ?

Pos <: Int

Neg <: Int

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

assignment

$$\frac{\Gamma \vdash \mathrm{e}:\ \mathrm{T} \qquad \Gamma \vdash \mathrm{T} <:\ \mathrm{T'}}{\Gamma \vdash \mathrm{e}:\ \mathrm{T'}}$$

subtyping

does it type check? – yes

```
def intSqrt(x:Pos) : Pos = { ...}
var p : Pos
var q : Neg
var r : Pos
q = -5
p = q
r = intSqrt(p)
```

Γ = {(p, Pos), (q, Neg), (r, Pos), (intSqrt, Pos $\rightarrow$ Pos)}

Runtime error: intSqrt invoked with a negative argument!

x must be able to store any value from T

e can have any value from T

$$\frac{? \qquad \Gamma \vdash \mathrm{e}:\ \mathrm{T}}{\Gamma \vdash (\mathrm{x} = \mathrm{e}):\ \mathrm{void}}$$

Cannot use $\Gamma \vdash$ to mean "x promises it can store any e $\in$ T"

Recall Our Type Derivation

Pos <: Int
Neg <: Int

$$\frac{\Gamma \vdash x:\ T \qquad \Gamma \vdash e:\ T}{\Gamma \vdash (x = e):\ \text{void}} \quad \text{assignment}$$

$$\frac{\Gamma \vdash e:\ T \qquad \Gamma \vdash T <:\ T'}{\Gamma \vdash e:\ T'} \quad \text{subtyping}$$

does it type check? – yes

```
def intSqrt(x:Pos) : Pos = { ...}
var p : Pos
var q : Neg
var r : Pos
q = -5
p = q
r = intSqrt(p)
```

Γ = {(p, Pos), (q, Neg), (r, Pos), (intSqrt, Pos → Pos)}

Runtime error: intSqrt invoked with a negative argument!

Values from p are integers. But p did not promise to store all kinds of integers/ Only positive ones!

$$\dfrac{\dfrac{p:\ \text{Pos} \qquad \text{Pos} <:\ \text{Int}}{\boxed{p:\ \text{Int}}} \qquad \dfrac{q:\ \text{Neg} \qquad \text{Neg} <:\ \text{Int}}{q:\ \text{Int}}}{(p{=}q):\ \text{void}}$$

Corrected Type Rule for Assignment

Pos <: Int

Neg <: Int

$$\frac{\Gamma \vdash x\colon\ T \qquad \Gamma \vdash e\colon\ T}{\Gamma \vdash (x = e)\colon\ \text{void}}$$ assignment

$$\frac{\Gamma \vdash e\colon\ T \qquad \Gamma \vdash T <\colon\ T'}{\Gamma \vdash e\colon\ T'}$$ subtyping

does it type check? – yes

```
def intSqrt(x:Pos) : Pos = { ...}
var p : Pos
var q : Neg
var r : Pos
q = -5
p = q
r = intSqrt(p)
```

Γ = {(p, Pos), (q, Neg), (r, Pos), (intSqrt, Pos → Pos)}

does not type check

x must be able to store any value from T

e can have any value from T

$$\frac{(x, T) \in \Gamma \qquad \Gamma \vdash e\colon\ T}{\Gamma \vdash (x = e)\colon\ \text{void}}$$

Γ stores declarations (promises)

How could we ensure that some other programs will not break?

Type System Soundness

Today

- More Subtyping Rules ✓
  - product types (pairs)
  - function types
  - classes
- Soundness
  - motivating example ✓
  - idea of proving soundness ←
  - operational semantics
  - a soundness proof
- Subtyping and generics

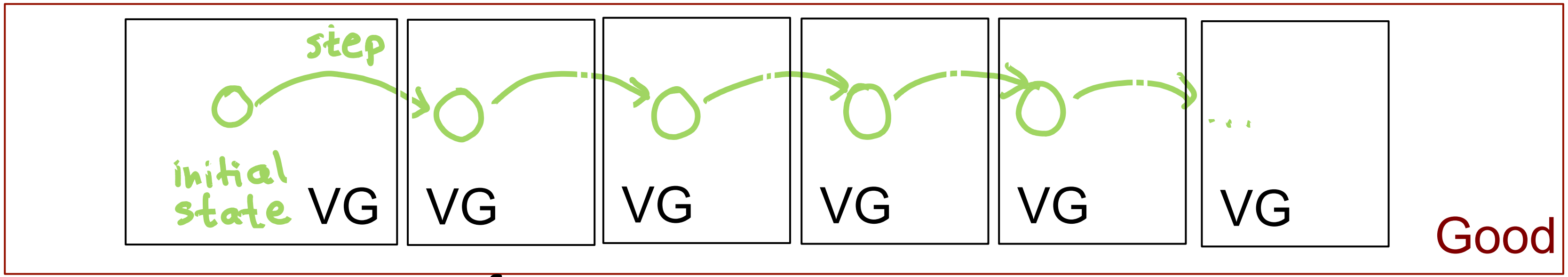
Proving Soundness of Type Systems

- Goal of a sound type system:
  - if the program type checks, then it never “crashes”
  - crash = some precisely specified bad behavior

    e.g. invoking an operation with a wrong type
    - dividing one string by another string “cat” / “frog
    - trying to multiply a Window object by a File object

    e.g. not dividing an integer by zero
- Never crashes: no matter how long it executes
  - proof is done by induction on program execution

Proving Soundness by Induction



- Program moves from state to state
- Bad state = state where program is about to exhibit a bad operation (“cat” / “frog”)
- Good state = state that is not bad
- To prove:
  program type checks → states in all executions are good
- Usually need a *stronger inductive hypothesis*; some notion of very good (VG) state such that:
  program type checks → program’s initial state is very good
  state is very good → next state is also very good
  state is very good → state is good (not about to crash)

A Simple Programming Language

Program State

```
var x : Pos
var y : Int
var z : Pos
x = 3
y = -5
z = 4
x = x + z
y = x / z
z = z + x
```

position in source

Initially, all variables
have value 1

values of variables:

x = 1

y = 1

z = 1

Program State

```
var x : Pos
var y : Int
var z : Pos
x = 3
y = -5
z = 4
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

```
  x = 3
  y = 1
  z = 1
```

Program State

```
var x : Pos
var y : Int
var z : Pos
x = 3
y = -5
z = 4
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

```
  x = 3
  y = -5
  z = 1
```

Program State

```
var x : Pos
var y : Int
var z : Pos
x = 3
y = -5
z = 4
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

x = 3

y = -5

z = 4

Program State

```
var x : Pos
var y : Int
var z : Pos
x = 3
y = -5
z = 4
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

x = 7

y = -5

z = 4

Program State

```
var x : Pos
var y : Int
var z : Pos
x = 3
y = -5
z = 4
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

x = 7

y = 1

z = 4

formal description of such program execution
is called operational semantics

Definition of Simple Language

Programs:

var x_1 : Pos
var x_2 : Int
…
var x_n : Pos

variable declarations
var x: Pos
or
var x: Int

followed by

$x_i = x_j$
$x_p = x_q + x_r$
$x_a = x_b / x_c$
…
$x_p = x_q + x_r$

statements of one of 3 forms
1) $x_i = x_j$
2) $x_i = x_j / x_k$
3) $x_i = x_j + x_k$

(No complex expressions)

Type rules:
Γ = { (x_1, Pos),
(x_2, Int),
…
(x_n, Pos)}
Pos <: int

$$\frac{(x,T)\in\Gamma \qquad \Gamma\vdash e:T}{\Gamma\vdash(x=e):void}$$

$$\frac{\Gamma\vdash x:T \qquad T<:T'}{\Gamma\vdash x:T'}$$

$$\frac{(x,T)\in\Gamma}{\Gamma\vdash x:T} \qquad \frac{e_1:Int \qquad e_2:Int}{e_1+e_2:Int}$$

$$\frac{e_1:Int \qquad e_2:Pos}{e_1/e_2:Int} \qquad \frac{e_1:Pos \qquad e_2:Pos}{e_1+e_2:Pos}$$

$$\frac{}{\text{k: Pos}} \qquad \frac{}{\text{-k: Int}}$$

Bad State: About to Divide by Zero (Crash)

```
var x : Pos
var y : Int
var z : Pos
x = 1
y = -1
z = x + y
x = x + z
y = x / z
z = z +
```

position in source

values of variables:

x = 1

y = -1

z = 0

Definition: state is *bad* if the next instruction is of the form $x_i = x_j / x_k$ and x_k has value 0 in the current state.

Good State: Not (Yet) About to Divide by Zero

```
var x : Pos
var y : Int
var z : Pos
x = 1
y = -1
z = x + y
x = x + z
y = x / z
z = z + x
```

← position in source

values of variables:

$x = 1$

$y = -1$

$z = 1$

Good

Definition: state is *good* if it is not *bad.*

Definition: state is *bad* if the next instruction is of the form $x_i = x_j / x_k$ and x_k has value 0 in the current state.

Good State: Not (Yet) About to Divide by Zero

```
var x : Pos
var y : Int
var z : Pos
x = 1
y = -1
z = x + y
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

x = 1

y = -1

z = 0

Good

Definition: state is *good* if it is not *bad.*

Definition: state is *bad* if the next instruction is of the form $x_i = x_j / x_k$ and x_k has value 0 in the current state.

Moved from Good to Bad in One Step!

Being good is not preserved by one step, not inductive!
It is very local property, does not take future into account.

```
var x : Pos
var y : Int
var z : Pos
x = 1
y = -1
z = x + y
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:
x = 1
y = -1
z = 0

Bad

Definition: state is *good* if it is not *bad.*

Definition: state is *bad* if the next instruction is of the form $x_i = x_j / x_k$ and x_k has value 0 in the current state.

Being Very Good: A Stronger Inductive Property

Pos = { 1, 2, 3, ... }

```
var x : Pos
var y : Int
var z : Pos
x = 1
y = -1
z = x + y
x = x + z
y = x / z
z = z + x
```

position in source

This state is already not *very good.*
We took future into account.

values of variables:
x = 1
y = -1
z = 0 ∉ Pos

Definition: state is *good* if it is not about to divide by zero.

Definition: state is *very good* if each variable belongs to the domain determined by its type (if z:Pos, then z is strictly positive).

If you are a little typed program, what will your parents teach you?

- If you *type check* and succeed:
  - you will be *very good* from the start.
  - if you are *very good*, then you will remain *very good* in the next step
  - If you are *very good*, you will not *crash.*

Hence, type check and you will never crash!

Soundnes proof = defining “very good” and checking the properties above.

Definition of Simple Language

Programs:

var x_1 : Pos
var x_2 : Int
…
var x_n : Pos

variable declarations
var x: Pos
or
var x: Int

followed by

$x_i = x_j$
$x_p = x_q + x_r$
$x_a = x_b / x_c$
…
$x_p = x_q + x_r$

statements of one of 3 forms

1) $x_i = x_j$
2) $x_i = x_j / x_k$
3) $x_i = x_j + x_k$

(No complex expressions)

Type rules:

Γ = { (x_1, Pos),
(x_2, Int),
…
(x_n, Pos)}

Pos <: int

$$\frac{(x,T) \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash (x = e) : void}$$

$$\frac{\Gamma \vdash x : T \qquad T <: T'}{\Gamma \vdash x : T'}$$

$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{e_1 : Int \qquad e_2 : Int}{e_1 + e_2 : Int}$$

$$\frac{e_1 : Int \qquad e_2 : Pos}{e_1 / e_2 : Int} \qquad \frac{e_1 : Pos \qquad e_2 : Pos}{e_1 + e_2 : Pos}$$

$$\frac{}{\text{k: Pos}} \qquad \frac{}{\text{-k: Int}}$$

Checking Properties in Our Case

Holds: in initial state, variables are =1

- If you *type check* and succeed:
  - ✓ – you will be *very good* from the start.

$1 \in$ Pos
$1 \in$ Int

  - – if you are *very good*, then you will remain *very good* in the next step
  - ✓ – If you are *very good*, you will not *crash.*

If next state is x / z, type rule ensures z has type Pos
Because state is very good, it means z $\in$ Pos
so z is not 0, and there will be no crash.

Definition: state is *very good* if each variable belongs to the domain determined by its type (if z:Pos, then z is strictly positive).

Example Case 1

Assume each variable belongs to its type.

```
var x : Pos
var y : Pos
var z : Pos
y = 3
z = 2
z = x + y
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

x = 1

y = 3

z = 2

the next statement is: z=x+y
where x,y,z are declared Pos.

Goal: prove that again each variable belongs to its type.

- variables other than z did not change, so belong to their type
- z is sum of two positive values, so it will have positive value

Example Case 2

Assume each variable belongs to its type.

```
var x : Pos
var y : Int
var z : Pos
y = -5
z = 2
z = x + y
x = x + z
y = x / z
z = z + x
```

position in source

values of variables:

```
x = 1
y = -5
z = 2
```

the next statement is: z=x+y
where x,z declared Pos, y declared Int

Goal: prove that again each variable belongs to its type.
this case is impossible, because z=x+y would not type check

How do we know it could not type check?

Must Carefully Check Our Type Rules

var x : Pos
var y : Int
var z : Pos
y = -5
z = 2
z = x + y
x = x + z
y = x / z
z = z + x

Conclude that the only types we can derive are:

x : Pos, x : Int
y : Int
x + y : Int

Cannot type check
z = x + y in this environment.

Type rules:
Γ = { (x_1, Pos),
(x_2, Int),
…
(x_n, Pos)}
Pos <: int

$$\frac{(x,T) \in \Gamma \qquad \Gamma \vdash e : T}{\Gamma \vdash (x = e) : void}$$

$$\frac{\Gamma \vdash x : T \qquad T <: T'}{\Gamma \vdash x : T'}$$

$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{e_1 : Int \qquad e_2 : Int}{e_1 + e_2 : Int}$$

$$\frac{e_1 : Int \qquad e_2 : Pos}{e_1 / e_2 : Int} \qquad \frac{e_1 : Pos \qquad e_2 : Pos}{e_1 + e_2 : Pos}$$

$$\frac{}{\text{k: Pos}} \qquad \frac{}{\text{-k: Int}}$$

We would need to check all cases (there are many, but they are easy)

Remark

- We used in examples Pos <: Int

- Same examples work if we have

class Int { ... }
class Pos extends Int { ... }

and is therefore relevant for OO languages

Today

- More Subtyping Rules ✓
  - product types (pairs)
  - function types
  - classes
- Soundness ✓
  - motivating example
  - idea of proving soundness
  - operational semantics
  - a soundness proof
- Subtyping and generics ←

Simple Parametric Class

class Ref[T](var content : T)

Can we use the subtyping rule

$$\frac{\mathrm{T} <: \mathrm{T'}}{\mathrm{Ref[T]} <: \mathrm{Ref[T']}} \qquad \frac{\mathrm{Pos} <: \mathrm{Int}}{\mathrm{Ref[Pos]} <: \mathrm{Ref[Int]}}$$

$$\left.\begin{array}{l}\text{var x : Ref[Pos]}\\ \text{var y : Ref[Int]}\\ \text{var z : Int}\end{array}\right\} \Gamma$$

```
x.content = 1
y.content = -1
y = x
y.content = 0
z = z / x.content
```

$$\frac{\dfrac{\Gamma \vdash x : Ref[Pos]}{(x, Ref[Int]) \in \Gamma} \qquad \Gamma \vdash y : Ref[Int]}{\mathrm{(y{=}x){:}void}}$$

type checks

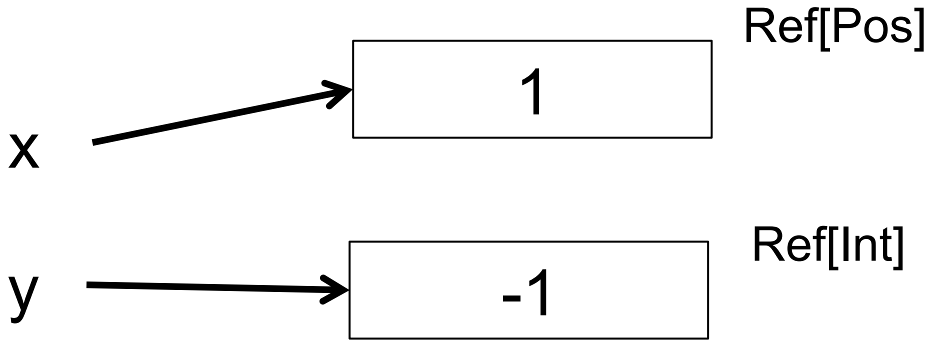
Simple Parametric Class

class Ref[T](var content : T)

Can we use the subtyping rule

$$\frac{T <: T'}{Ref[T] <: Ref[T']}$$

```
var x : Ref[Pos]
var y : Ref[Int]
var z : Int
x.content = 1
y.content = -1
y = x
y.content = 0
z = z / x.content
```



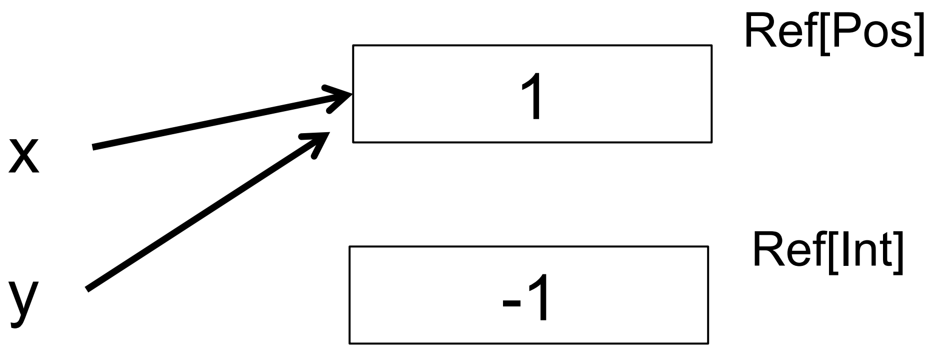
Simple Parametric Class

```
class Ref[T](var content : T)
```

Can we use the subtyping rule

$$\frac{\mathrm{T} <: \mathrm{T'}}{\mathrm{Ref[T]} <: \mathrm{Ref[T']}}$$

```
var x : Ref[Pos]
var y : Ref[Int]
var z : Int
x.content = 1
y.content = -1
y = x
y.content = 0
z = z / x.content
```



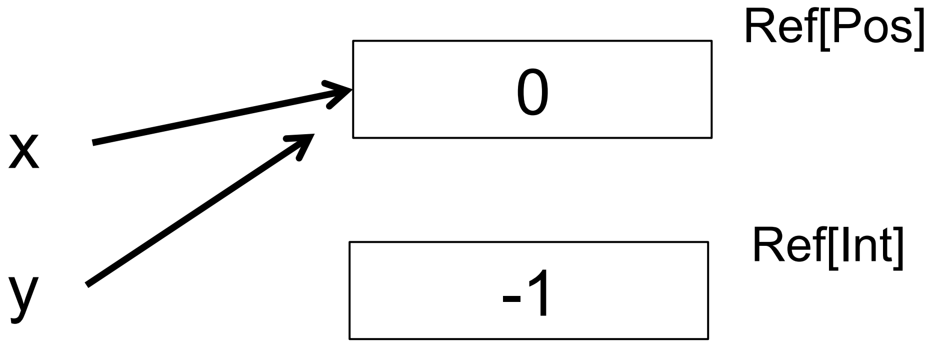
Simple Parametric Class

```
class Ref[T](var content : T)
```

Can we use the subtyping rule

$$\frac{T <: T'}{\mathrm{Ref}[T] <: \mathrm{Ref}[T']}$$

```
var x : Ref[Pos]
var y : Ref[Int]
var z : Int
x.content = 1
y.content = -1
y = x
y.content = 0
z = z / x.content
```



CRASHES

Analogously

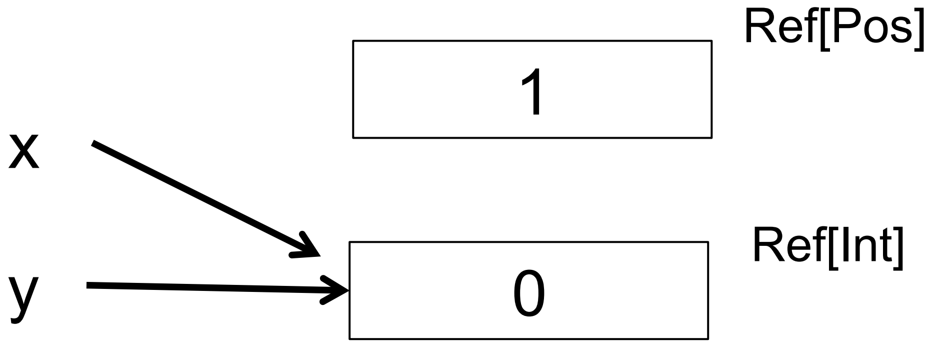
class Ref[T](var content : T)

Can we use the converse subtyping rule

$$\frac{\mathrm{T} <: \mathrm{T'}}{\mathrm{Ref[T']} <: \mathrm{Ref[T]}}$$

```
var x : Ref[Pos]
var y : Ref[Int]
var z : Int
x.content = 1
y.content = -1
x = y
y.content = 0
z = z / x.content
```

CRASHES



Mutable Classes do not Preserve Subtyping

```
class Ref[T](var content : T)
```

Even if T <: T’,

Ref[T] and Ref[T’] are unrelated types

```
var x : Ref[T]
var y : Ref[T’]
...
x = y  ←  type checks only if T=T’
...
```

Same Holds for Arrays, Vectors, all mutable containers

Even if T <: T’,

Array[T] and Array[T’] are unrelated types

```
var x : Array[Pos](1)
var y : Array[Int](1)
var z : Int
x[0] = 1
y[0] = -1
y = x
y[0] = 0
z = z / x[0]
```

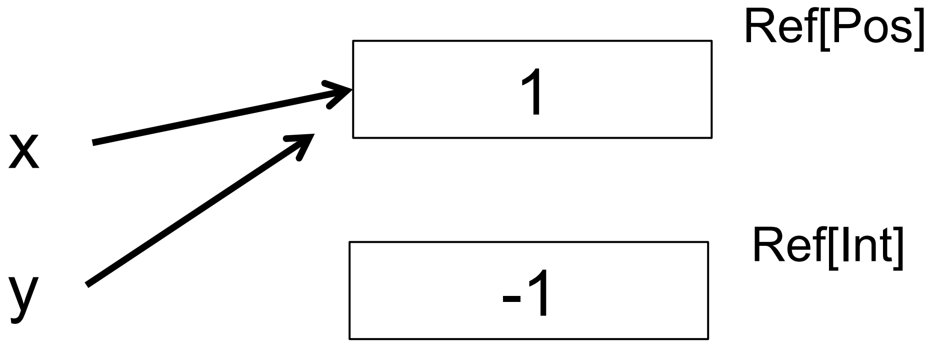
Case in Soundness Proof Attempt

class Ref[T](var content : T)

Can we use the subtyping rule

$$\frac{T <: T'}{\mathrm{Ref}[T] <: \mathrm{Ref}[T']}$$

```
var x : Ref[Pos]
var y : Ref[Int]
var z : Int
x.content = 1
y.content = -1
y = x
y.content = 0
z = z / x.content
```



prove each variable belongs to its type: variables other than y did not change.. (?!)

Mutable vs Immutable Containers

- Immutable container, Coll[T]
  - has methods of form e.g. get(x:A) : T
  - if T <: T’, then Coll[T’] has get(x:A) : T’
  - we have (A → T) <: (A→ T’)
    covariant rule for functions, so Coll[T] <: Coll[T’]
- Write-only data structure have
  - setter-like methods, set(v:T) : B
  - if T <: T’, then Container[T’] has set(v:T) : B
  - would need (T → B) <: (T’ → B)
    contravariance for arguments, so Coll[T’] <: Coll[T]
- Read-Write data structure need both,
  so they are invariant, no subtype on Coll if T <: T’