

Exercise

Determine the output of the following program assuming static and dynamic scoping. Explain the difference, if there is any.

```
object MyClass {  
  val x = 5  
  def foo(z: Int): Int = { x + z }  
  def bar(y: Int): Int = {  
    val x = 1; val z = 2  
    foo(y)  
  }  
  def main() {  
    val x = 7  
    println(foo(bar(3)))  
  }  
}
```

type judgement relation

$$\Gamma \vdash e : T$$

if the (free) variables of e have types given by Γ ,
then e (correctly) type checks and has type T

$$\Gamma \vdash e_1 : T_1 \quad \dots \quad \Gamma \vdash e_n : T_n$$

$$\Gamma \vdash e : T$$

If e_1 type checks in Γ and has type T_1 and ...
and e_n type checks in Γ and has type T_n
then e type checks in Γ and has type T

type rule

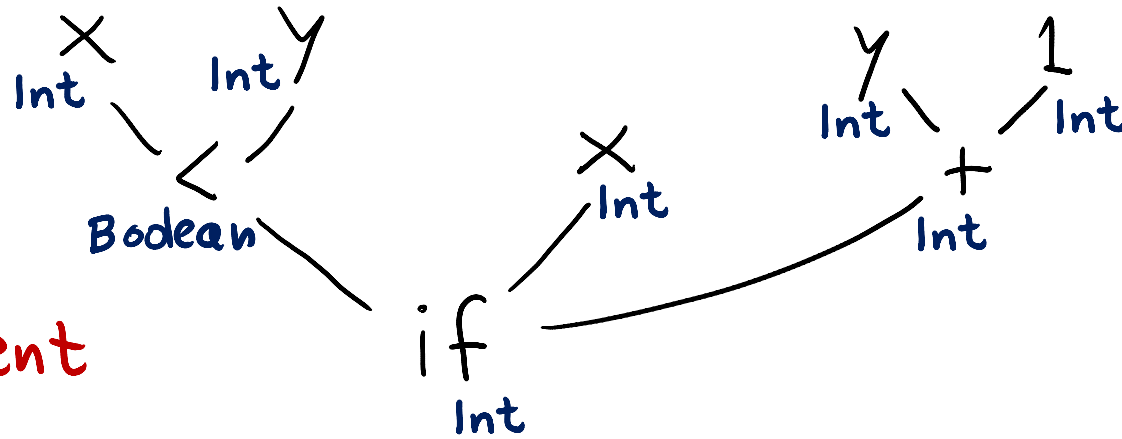
Type Checker Implementation Sketch

```
def typeCheck( $\Gamma$  : Map[ID, Type], e : ExprTree) : TypeTree = {  
  e match {  
    case Var(id) => {  $\Gamma$ (id) match  
      case Some(t) => t  
      case None => error(UnknownIdentifier(id,id.pos))  
    }  
    case If(c,e1,e2) => {  
      val tc = typeCheck( $\Gamma$ ,c)  
      if (tc != BooleanType) error(IfExpectsBooleanCondition(e.pos))  
      val t1 = typeCheck( $\Gamma$ , e1); val t2 = typeCheck( $\Gamma$ , e2)  
      if (t1 != t2) error(IfBranchesShouldHaveSameType(e.pos))  
      t1  
    }  
    ...  
  }  
}
```

Derivation Using Type Rules

$x : \text{Int}$
 $y : \text{Int}$

type environment
 Γ



Let $\Gamma = \{(x, \text{Int}), (y, \text{Int})\}$

$$\begin{array}{c}
 \frac{(x, \text{Int}) \in \Gamma}{\Gamma \vdash x : \text{Int}} \quad \frac{(y, \text{Int}) \in \Gamma}{\Gamma \vdash y : \text{Int}} \\
 \hline
 \Gamma \vdash (x < y) : \text{Boolean} \\
 \hline
 \frac{(x, \text{Int}) \in \Gamma}{\Gamma \vdash x : \text{Int}} \quad \frac{(y, \text{Int}) \in \Gamma}{\Gamma \vdash y : \text{Int}} \quad \frac{}{\Gamma \vdash 1 : \text{Int}} \\
 \hline
 \Gamma \vdash (y + 1) : \text{Int} \\
 \hline
 \Gamma \vdash (\text{if}(x < y) x \text{ else } y + 1) : \text{Int}
 \end{array}$$

Type Rule for Function Application

We can treat operators as variables that have function type

$$+ : \text{Int} \times \text{Int} \rightarrow \text{Int}$$

$$< : \text{Int} \times \text{Int} \rightarrow \text{Boolean}$$

$$\&\& : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean}$$

We can replace many previous rules with application rule:

$$\Gamma \vdash e_1 : T_1 \quad \dots \quad \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : ((T_1 \times \dots \times T_n) \rightarrow T)$$

$$\Gamma \vdash f(e_1, \dots, e_n) : T$$

Computing the Environment of a Class

$\Gamma_0 = \{$

```
object World {  
  var data : Int  
  var name : String  
  def m(x : Int, y : Int) : Boolean { ... }  
  def n(x : Int) : Int {  
    if (x > 0) p(x - 1) else 3  
  }  
  def p(r : Int) : Int = {  
    var k = r + 2  
    m(k, n(k))  
  }  
}
```

$(data, Int),$
 $(name, String),$
 $(m, Int \times Int \rightarrow Boolean),$
 $(n, Int \rightarrow Int),$

 $(p, Int \rightarrow Int)$
 $\}$

Type check each function m,n,p in this global environment

Extending the Environment

$\Gamma_0 = \{$

$(data, Int),$
 $(name, String),$
 $(m, Int \times Int \rightarrow Boolean),$
 $(n, Int \rightarrow Int),$
 $(p, Int \rightarrow Int) \}$

```
class World {  
  var data : Int  
  var name : String  
  def m(x : Int, y : Int) : Boolean { ... }  
  def n(x : Int) : Int {  
    if (x > 0) p(x - 1) else 3  
  }  
  def p(r : Int) : Int = {  
    var k: Int = r + 2  
    m(k, n(k))  
  }  
}
```

$\leftarrow \Gamma_0$

$\leftarrow \Gamma_1 = \Gamma_0 \oplus \{(r, Int)\}$

$\leftarrow \Gamma_2 = \Gamma_1 \oplus \{(k, Int)\} = \Gamma_0 \cup \{(r, Int), (k, Int)\}$

Type Checking Expression in a Body

$\Gamma_0 = \{$

(data, Int),
(name, String),
(m, Int x Int \rightarrow Boolean),
(n, Int \rightarrow Int),
(p, Int \rightarrow Int) }

```
class World {  
  var data : Int  
  var name : String  
  def m(x : Int, y : Int) : Boolean { ... }  
  def n(x : Int) : Int {  
    if (x > 0) p(x - 1) else 3  
  }  
  def p(r : Int) : Int = {  
    var k: Int = r + 2  
    m(k, n(k))  
  }  
}
```

$\leftarrow \Gamma_0$

$\leftarrow \Gamma_1 = \Gamma_0 \oplus \{(r, \text{Int})\}$

$\leftarrow \Gamma_2 = \Gamma_1 \oplus \{(k, \text{Int})\}$

Remember Function Updates

$$\{(x, T_1), (y, T_2)\} \oplus \{(x, T_3)\} = \{(x, T_3), (y, T_2)\}$$

Type Rule for Method Bodies

$$\Gamma \vdash (\text{def } m(x_1:T_1, \dots, x_n:T_n):T = e) : \text{OK}$$

Type Rule for Assignments

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

Type Rules for Block: $\{ \text{var } x_1:T_1 \dots \text{var } x_n:T_n; s_1; \dots s_m; e \}$

$$\Gamma \vdash \{ \text{var } x_1:T_1; \dots; \text{var } x_n:T_n; s_1; \dots; s_n; e \} : T$$

Blocks with Declarations in the Middle

$\frac{}{\Gamma \vdash \{e\} : T}$ just
expression

$\frac{}{\Gamma \vdash \{\} : \text{void}}$ empty

$\frac{}{\Gamma \vdash \{\text{var } x : T_1; t_2; \dots; t_n\} : T}$

declaration is first

$\frac{}{\Gamma \vdash \{s_1; t_2; \dots; t_n\} : T}$

statement is first

Rule for While Statement

$$\Gamma \vdash (\text{while}(b) s) : \text{void}$$

Rule for Method Call

```
class T0 {  
  ...  
  def m(x1:T1, ..., xn:Tn):T = {  
    ...  
  }  
  ...  
}
```

$$\frac{\Gamma \vdash x : T_0 \quad \Gamma \vdash T_0.m : T_0 \times T_1 \times \dots \times T_n \rightarrow T \quad \forall i \in \{1, 2, \dots, n\} \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash x.m(e_1, \dots, e_n) : T}$$

Example to Type Check

```

object World {
  var z : Boolean
  var u : Int
  def f(y : Boolean) : Int {
    z = y
    if (u > 0) {
      u = u - 1
      var z : Int
      z = f(!y) + 3
      z+z
    } else { 0 }
  }
}

```

$$\Gamma_0 = \{$$

(z, Boolean),
 (u, Int),
 (f, Boolean \rightarrow Int) }

$$\Gamma_1 = \Gamma_0 \oplus \{(y, \text{Boolean})\}$$

$$\frac{\Gamma_1 \vdash z: \text{Boolean} \quad \Gamma_1 \vdash y: \text{Boolean}}{\Gamma_1 \vdash (z=y): \text{void}}$$

Exercise:

$$\frac{\text{???}}{\Gamma_1 \vdash \text{if } (n > 0) \{ \text{body} \} \text{ else } \{ 0 \} : \text{Int}}$$

Overloading of Operators

Int x Int \rightarrow Int

$$\frac{\Gamma \vdash e_1: \text{Int} \quad \Gamma \vdash e_2: \text{Int}}{\Gamma \vdash (e_1 + e_2): \text{Int}}$$

Not a problem for type checking from leaves to root

String x String \rightarrow String

$$\frac{\Gamma \vdash e_1: \text{String} \quad \Gamma \vdash e_2: \text{String}}{\Gamma \vdash (e_1 + e_2): \text{String}}$$

Arrays

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

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

Assigning to an array

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

Example with Arrays

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

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

$$\frac{\Gamma \vdash a : \text{Array}(\text{Int}) \quad \frac{\Gamma \vdash a : \text{Array}(\text{Int}) \quad \Gamma \vdash k : \text{Int}}{\Gamma \vdash a[k] : \text{Int}}}{\Gamma \vdash a[a[k]] : \text{Int}} \quad \frac{\Gamma \vdash a : \text{Array}(\text{Int}) \quad \Gamma \vdash k : \text{Int}}{\Gamma \vdash a[k] = a[a[k]] : \text{Int}}$$

Type Rules (1)

$$\frac{(x: T) \in \Gamma}{\Gamma \vdash x: T} \quad \text{variable}$$

$$\frac{}{\text{IntConst}(k): \text{Int}} \quad \text{constant}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \dots \quad \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : (T_1 \times \dots \times T_n \rightarrow T)}{\Gamma \vdash f(e_1, \dots, e_n) : T} \quad \text{function application}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash (e_1 + e_2) : \text{Int}} \quad \text{plus} \quad \frac{\Gamma \vdash e_1 : \text{String} \quad \Gamma \vdash e_2 : \text{String}}{\Gamma \vdash (e_1 + e_2) : \text{String}}$$

$$\frac{\Gamma \vdash b : \text{Boolean} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash (\text{if}(b) e_1 \text{ else } e_2) : T} \quad \text{if}$$

$$\frac{\Gamma \vdash b : \text{Boolean} \quad \Gamma \vdash s : \text{void}}{\Gamma \vdash (\text{while}(b) s) : \text{void}} \quad \text{while}$$

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

Type Rules (2)

$$\frac{\Gamma \vdash e: T}{\Gamma \vdash \{e\}: T}$$

$$\frac{}{\Gamma \vdash \{\}: \text{void}}$$

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

block

$$\frac{\Gamma \vdash s_1: \text{void} \quad \Gamma \vdash \{t_2; \dots; t_n\}: T}{\Gamma \vdash \{s_1; t_2; \dots; t_n\}: T}$$

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

array use

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

array
assignment

Type Rules (3)

Γ^C - top-level environment of class C

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



$\Gamma^C = \{ (x, \text{Int}), (m, C \times \text{Int} \rightarrow \text{Boolean}) \}$

$$\frac{\Gamma \vdash e : C \quad \Gamma^C \vdash m : T \times T_1 \times \dots \times T_n \rightarrow T_{n+1} \quad \Gamma \vdash e_i : T_i \quad 1 \leq i \leq n}{\Gamma \vdash e.m(e_1, \dots, e_n) : T_{n+1}} \quad \text{method invocation}$$

$$\frac{\Gamma \vdash e : C \quad \Gamma^C \vdash f : T}{\Gamma \vdash e.f : T} \quad \text{field use}$$

$$\frac{\Gamma \vdash e : C \quad \Gamma^C \vdash f : T \quad \Gamma \vdash x : T}{\Gamma \vdash (e.f = x) : \text{void}} \quad \text{field assignment}$$

Does this program type check?

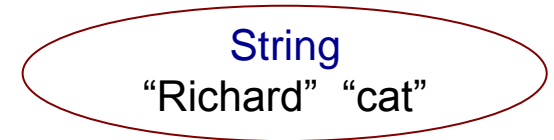
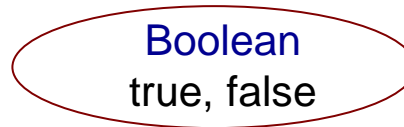
```
class Rectangle {  
  var width: Int  
  var height: Int  
  var xPos: Int  
  var yPos: Int  
  def area(): Int = {  
    if (width > 0 && height > 0)  
      width * height  
    else 0  
  }  
  def resize(maxSize: Int) {  
    while (area > maxSize) {  
      width = width / 2  
      height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

Meaning of Types

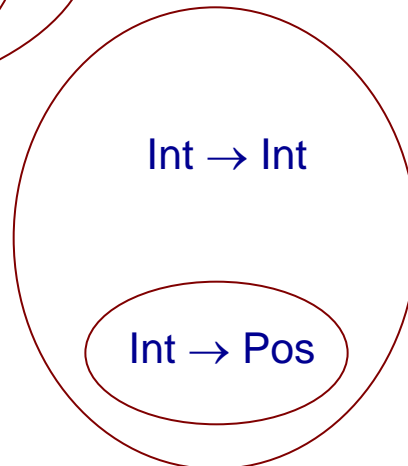
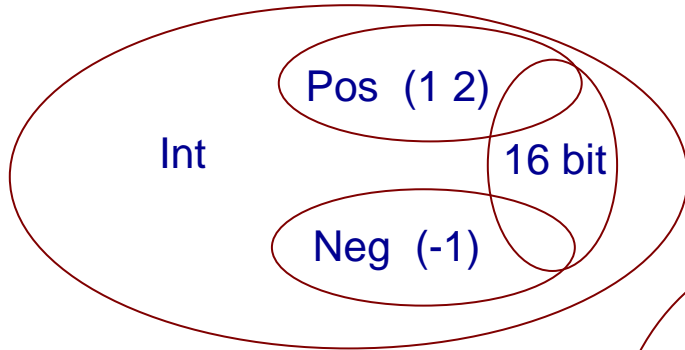
- Types can be viewed as named entities
 - explicitly declared classes, traits
 - their meaning is given by methods they have
 - constructs such as inheritance establish relationships between classes
- Types can be viewed as sets of values
 - $\text{Int} = \{ \dots, -2, -1, 0, 1, 2, \dots \}$
 - $\text{Boolean}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

Types as Sets

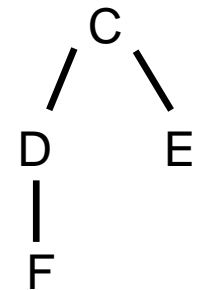
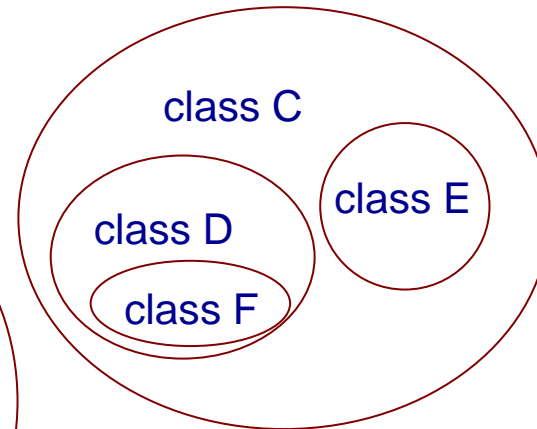
- Sets so far were disjoint



- 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

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

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

Types for Positive and Negative Ints

$\text{Int} = \{ \dots, -2, -1, 0, 1, 2, \dots \}$

$\text{Pos} = \{ 1, 2, \dots \}$ (not including zero)

$\text{Neg} = \{ \dots, -2, -1 \}$ (not including zero)

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

$\text{Pos} \mu \text{Int}$
 $\text{Neg} \mu \text{Int}$

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

$$\frac{x \in \text{Pos} \quad y \in \text{Pos}}{x + y \in \text{Pos}}$$

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

$$\frac{x \in \text{Pos} \quad y \in \text{Neg}}{x * y \in \text{Neg}}$$

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

$$\frac{x \in \text{Pos} \quad y \in \text{Pos} \text{ (y not zero)}}{x / y \in \text{Pos} \text{ (x/y well defined)}}$$

More Rules

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

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

More rules for division?

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

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

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

Making Rules Useful

- Let x be a variable

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

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

```
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
}
```

Pos <: Int

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

```
var p : Pos
var q : Int
```

```
q = f(p)
```

← Does this statement type check?

$$\frac{
 \frac{
 \frac{
 p: \text{Pos} \quad \text{Pos} <: \text{Int}
 }{
 p: \text{Int}
 }
 \quad
 f: \text{Int} \rightarrow \text{Pos}
 }{
 f(p): \text{Pos}
 }
 \quad
 \text{Pos} <: \text{Int}
 }{
 f(p): \text{Int}
 }
 \quad
 (q, \text{Int}) \in \Gamma
 }{
 q=f(p): \text{void}
 }$$

Using Subtyping

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

Pos <: Int

Γ f: Pos \rightarrow Pos

```
var p : Int  
var q : Int
```

```
q = f(p)
```

- 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  
}
```

Pos <: Int

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

```
var p : Int  
var q : Int
```

```
q = f(p)
```

- does not type check

Subtyping for Products

$$T_1 <: T_2 \text{ implies for all } e: \frac{\Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

Type for
a tuple:

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

$$\frac{\frac{x : T_1 \quad T_1 <: T'_1}{x : T'_1} \quad \frac{y : T_2 \quad T_2 <: T'_2}{y : T'_2}}{(x, y) : T'_1 \times T'_2}$$

So, we might as well add:

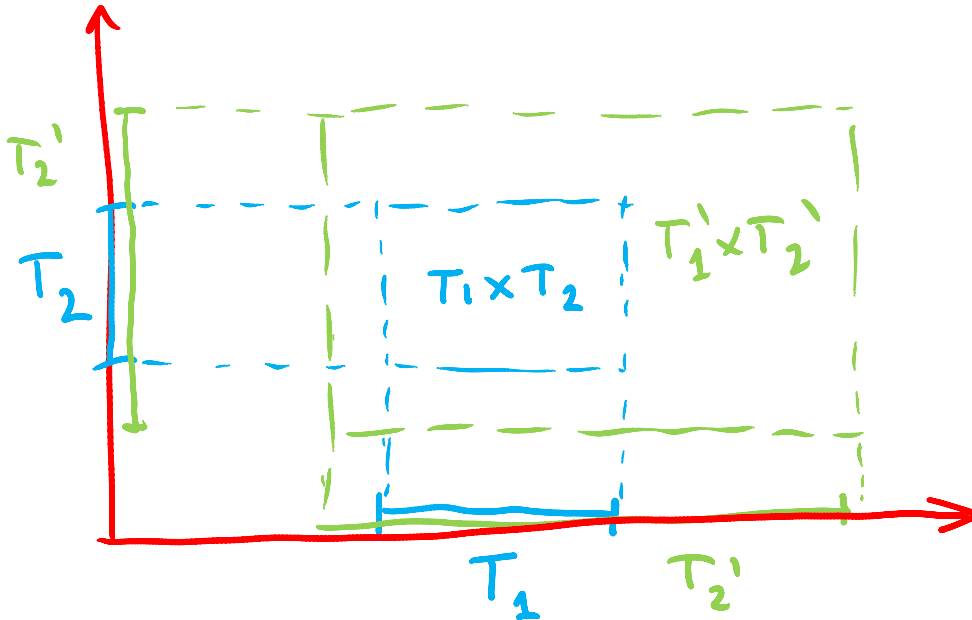
$$\frac{T_1 <: T'_1 \quad T_2 <: T'_2}{T_1 \times T_2 <: T'_1 \times T'_2}$$

covariant subtyping for pairs
Pair[T_1, T_2] also denoted (T_1, T_2)

Analogy with Cartesian Product

$$\frac{T_1 \subset T'_1 \quad T_2 \subset T'_2}{T_1 \times T_2 \subset T'_1 \times T'_2}$$

$$\frac{T_1 \subseteq T'_1 \quad T_2 \subseteq T'_2}{T_1 \times T_2 \subseteq T'_1 \times T'_2}$$



$$A \times B = \{ (a, b) \mid a \in A, b \in B \}$$

Subtyping and Function Types

Subtyping for Function Types

$$T_1 <: T_2 \text{ implies for all } e: \frac{\Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

$\xrightarrow{\hspace{1cm}}$
 $\xleftarrow{\hspace{1cm}}$

Function $[T_1, T_2]$

\nearrow contra-
 \nwarrow co-

$$\frac{\overbrace{T'_1 <: T_1 \dots T'_n <: T_n}^{\text{contravariance}} \quad \overbrace{T <: T'}^{\text{covariance}}}{(T_1 \times \dots \times T_n \rightarrow T) <: (T'_1 \times \dots \times T'_n \rightarrow T')}$$

Consequence:

$$\frac{\Gamma \vdash m : T_1 \times \dots \times T_n \rightarrow T \quad \frac{\Gamma \vdash e_1 : T'_1 \quad T'_1 <: T_1}{\Gamma \vdash e_1 : T_1} \quad \frac{\Gamma \vdash e_n : T'_n \quad T'_n <: T_n}{\Gamma \vdash e_n : T_n}}{\Gamma \vdash m(e_1, \dots, e_n) : T} \quad T <: T'}{\Gamma \vdash m(e_1, \dots, e_n) : T'}$$

as if $\Gamma \vdash m : T'_1 \times \dots \times T'_n \rightarrow T'$

Function Space as Set

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

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

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

We can prove

$$\frac{T'_1 \subseteq T_1 \quad T_2 \subseteq T'_2}{T_1 \rightarrow T_2 \subseteq T'_1 \rightarrow T'_2}$$

Proof

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

$$\frac{T'_1 \subseteq T_1 \quad T_2 \subseteq T'_2}{T_1 \rightarrow T_2 \subseteq T'_1 \rightarrow T'_2}$$

Subtyping for Classes

- Class C contains a collection of methods
- We view field `var f: T` as two methods
 - `getF(this:C): T` $C \rightarrow T$
 - `setF(this:C, x:T): void` $C \times T \rightarrow \text{void}$
- 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 : T1) : T2 = {...}  
}  
class D extends C {  
  override def m(x : T'1) : T'2 = {...}  
}
```

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

Therefore, we need to have:

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

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

Soundness of Types

ensuring that a type system
is not broken

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

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)
```

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

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

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

Runtime error: intSqrt invoked with a negative argument!

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

What went wrong in *Tootool 0.1* ?

Pos <: Int
Neg <: Int

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

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

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)
```

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

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

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

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

Recall Our Type Derivation

Pos <: Int
Neg <: Int

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)
```

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

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

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

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!

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

Corrected Type Rule for Assignment

Pos <: Int
Neg <: Int

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

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

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

does not type check

x must be able to store any value from T

e can have any value from T

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

Γ stores declarations (promises)

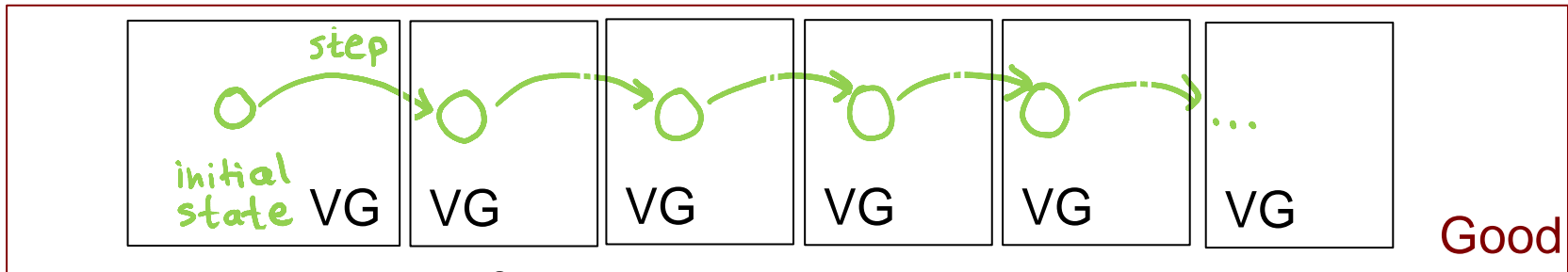
How could we ensure that some other programs will not break?

Type System Soundness

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 \rightarrow states in all executions are good
- Usually need a *stronger inductive hypothesis*;
some notion of very good (VG) state such that:
 - program type checks \rightarrow program’s initial state is very good
 - state is very good \rightarrow next state is also very good
 - state is very good \rightarrow state is good (not about to crash)

A Simple Programming Language

Program State

```
var x : Pos
```

```
var y : Int
```

```
var z : Pos
```

```
x = 3
```

 position in source

```
y = -5
```

```
z = 4
```

```
x = x + z
```

```
y = x / z
```

```
z = z + x
```

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
```

values of variables:

```
x = 7
y = 1
z = 4
```

 position in source

formal description of such program execution
is called operational semantics

Operational semantics

Operational semantics gives meaning to programs by describing how the program state changes as a sequence of steps.

- big-step semantics: consider the effect of entire blocks
- small-step semantics: consider individual steps (e.g. $z = x + y$)

V: set of variables in the program

pc: integer variable denoting the program counter

$g: V \rightarrow \text{Int}$ function giving the values of program variables

(g, pc) program state

Then, for each possible statement in the program we define how it changes the program state.

Example: $z = x+y$

$(g, \text{pc}) \rightarrow (g', \text{pc} + 1)$ s. t. $g' = g(z := g(x)+g(y))$

Type Rules of Simple Language

Programs:

var x_1 : Pos
 var x_2 : Int
 ...
 var x_n : Pos

variable declarations
 var x : Pos (strictly positive)
 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 the forms

- 1) $x_i = k$
- 2) $x_i = x_j$
- 3) $x_i = x_j / x_k$
- 4) $x_i = x_j + x_k$

(No complex expressions)

Type rules:

$\Gamma = \{ (x_1, \text{Pos}),$
 $(x_2, \text{Int}),$
 ...
 $(x_n, \text{Pos}) \}$

Pos <: Int

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

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

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

$$\frac{e_1 : \text{Int} \quad e_2 : \text{Pos}}{e_1 / e_2 : \text{Int}}$$

$$\frac{e_1 : \text{Pos} \quad e_2 : \text{Pos}}{e_1 + e_2 : \text{Pos}}$$

k : Pos

$-k$: 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 +
```

values of variables:

x = 1

y = -1

z = 0

 position in source

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

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

← position in source

values of variables:

x = 1

y = -1

z = 0 \notin 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, please type check, and you will never crash!

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

Definition of Simple Language

Programs:

$\text{var } x_1 : \text{Pos}$
 $\text{var } x_2 : \text{Int}$
 \dots
 $\text{var } x_n : \text{Pos}$

variable declarations
 $\text{var } x : \text{Pos}$
 or
 $\text{var } x : \text{Int}$

followed by

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

statements of one of the forms

- 1) $x_i = k$
- 2) $x_i = x_j$
- 3) $x_i = x_j / x_k$
- 4) $x_i = x_j + x_k$

(No complex expressions)

Type rules:

$\Gamma = \{ (x_1, \text{Pos}),$
 $(x_2, \text{Int}),$
 \dots
 $(x_n, \text{Pos}) \}$

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

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

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

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

$$\frac{e_1 : \text{Int} \quad e_2 : \text{Pos}}{e_1 / e_2 : \text{Int}}$$

$$\frac{e_1 : \text{Pos} \quad e_2 : \text{Pos}}{e_1 + e_2 : \text{Pos}}$$

$k : \text{Pos}$

$-k : \text{Int}$

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 ∈ Pos
1 ∈ 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 $z \in \text{Pos}$
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 $z:\text{Pos}$, 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
```

values of variables:

x = 1

y = 3

z = 2

 position in source

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

the next statement is: $z=x+y$

where x,z declared Pos, y declared Int

values of variables:

$x = 1$

$y = -5$

$z = 2$

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:

$\Gamma = \{ (x_1, \text{Pos}), (x_2, \text{Int}), \dots, (x_n, \text{Pos}) \}$

Pos <: int

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

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

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

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

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

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

Back to the start

$\overline{k: Pos}$ $\overline{-k: Int}$

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

$$\frac{\Gamma \vdash x : T \quad T <: T'}{\Gamma \vdash x : T'}$$
$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T}$$
$$\frac{e_1 : Int \quad e_2 : Int}{e_1 + e_2 : Int}$$
$$\frac{e_1 : Int \quad e_2 : Pos}{e_1 / e_2 : Int}$$
$$\frac{e_1 : Pos \quad e_2 : Pos}{e_1 + e_2 : Pos}$$

Does the proof still work?

If not, where does it break?

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

What if we want more complex types?

```
class A { }
```

```
class B extends A {  
    void foo() { }  
}
```

- Should it type check?
- Does this type check in Java?
- Does this type check in Scala?

```
class Test {
```

```
    public static void main(String[] args) {
```

```
        B[] b = new B[5];
```

```
        A[] a;
```

```
        a = b;
```

```
        System.out.println("Hello,");
```

```
        a[0] = new A();
```

```
        System.out.println("world!");
```

```
        b[0].foo();
```

```
    }
```

```
}
```

What if we want more complex types?

Suppose we add to our language a reference type:

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

Programs:

```
var x1 : Pos  
var x2 : Int  
var x3 : Ref[Int]  
var x4 : Ref[Pos]
```

```
x = y  
x = y + z  
x = y / z  
x = y + z.content  
x.content = y
```

Exercise 1:

Extend the type rules to use with Ref[T] types.

Show your new type system is sound.

Exercise 2:

Can we use the subtyping rule?

If not, where does the proof break?

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

Simple Parametric Class

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

Can we use the subtyping rule

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

$$\frac{\text{Pos} <: \text{Int}}{\text{Ref}[\text{Pos}] <: \text{Ref}[\text{Int}]}$$

```
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
```

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

type checks

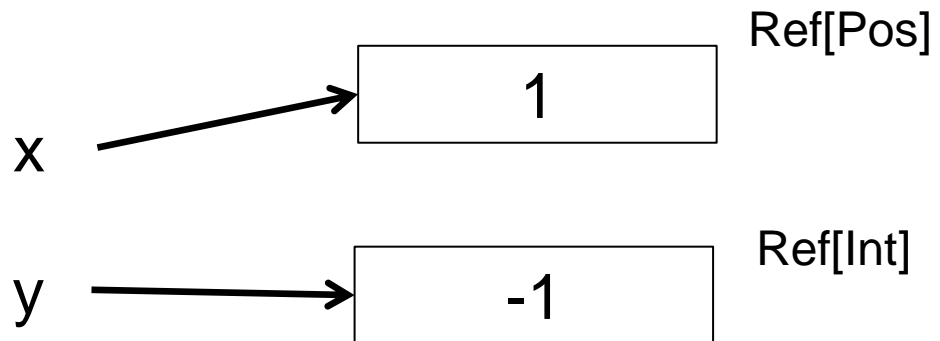
Simple Parametric Class

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

Can we use the subtyping rule

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

```
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
```



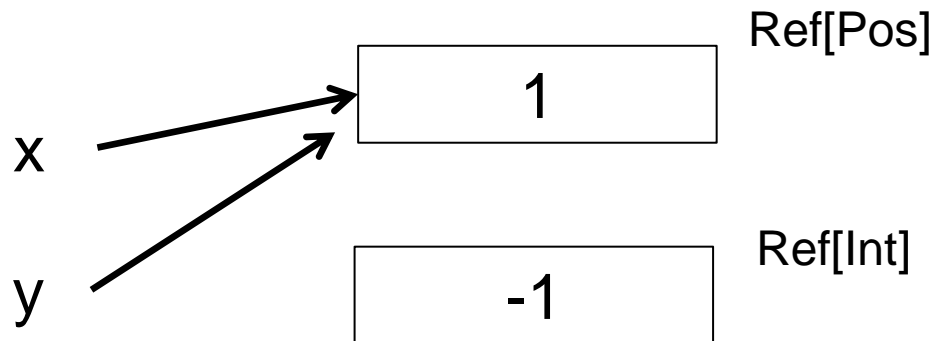
Simple Parametric Class

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

Can we use the subtyping rule

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

```
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
```



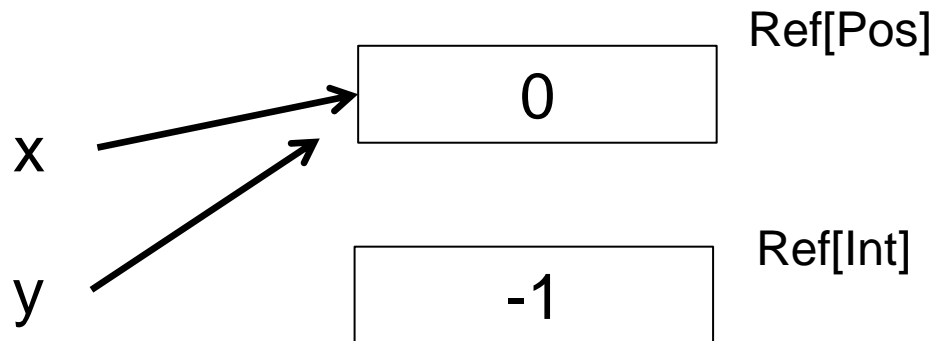
Simple Parametric Class

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

Can we use the subtyping rule

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

```
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

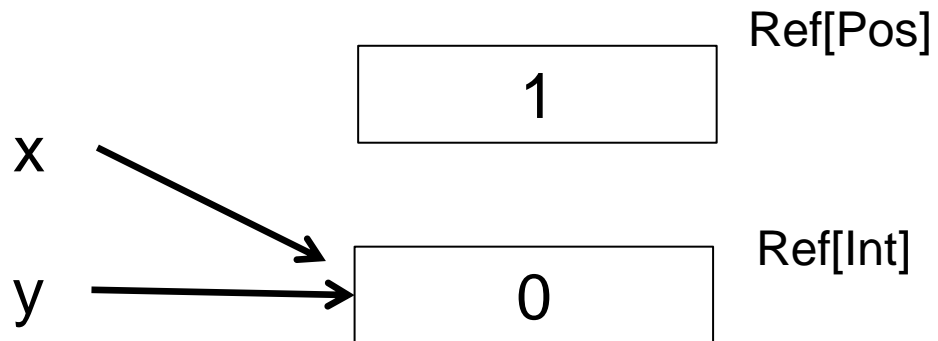
Analogously

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

Can we use the converse subtyping rule

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

```
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 $T <: T'$,

$\text{Ref}[T]$ and $\text{Ref}[T']$ 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]
```

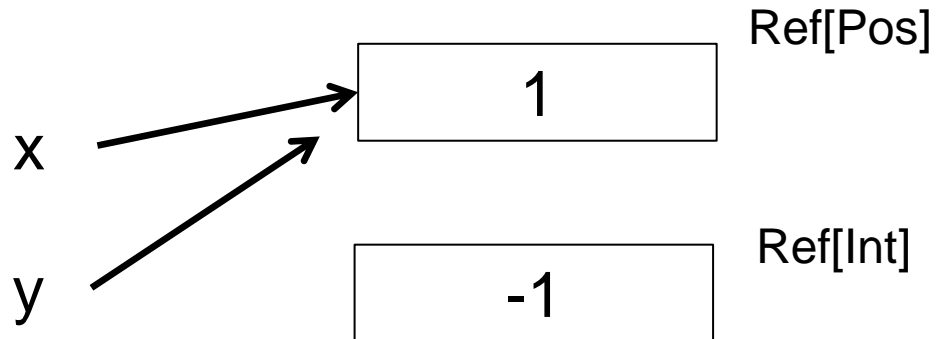
Case in Soundness Proof Attempt

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

Can we use the subtyping rule

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

```
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 $(A \rightarrow T) <: (A \rightarrow T')$
covariant rule for functions, so $\text{Coll}[T] <: \text{Coll}[T']$
- **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 $(T \rightarrow B) <: (T' \rightarrow B)$
contravariance for arguments, so $\text{Coll}[T'] <: \text{Coll}[T]$
- **Read-Write data structure need both,**
so they are invariant, no subtype on Coll if $T <: T'$