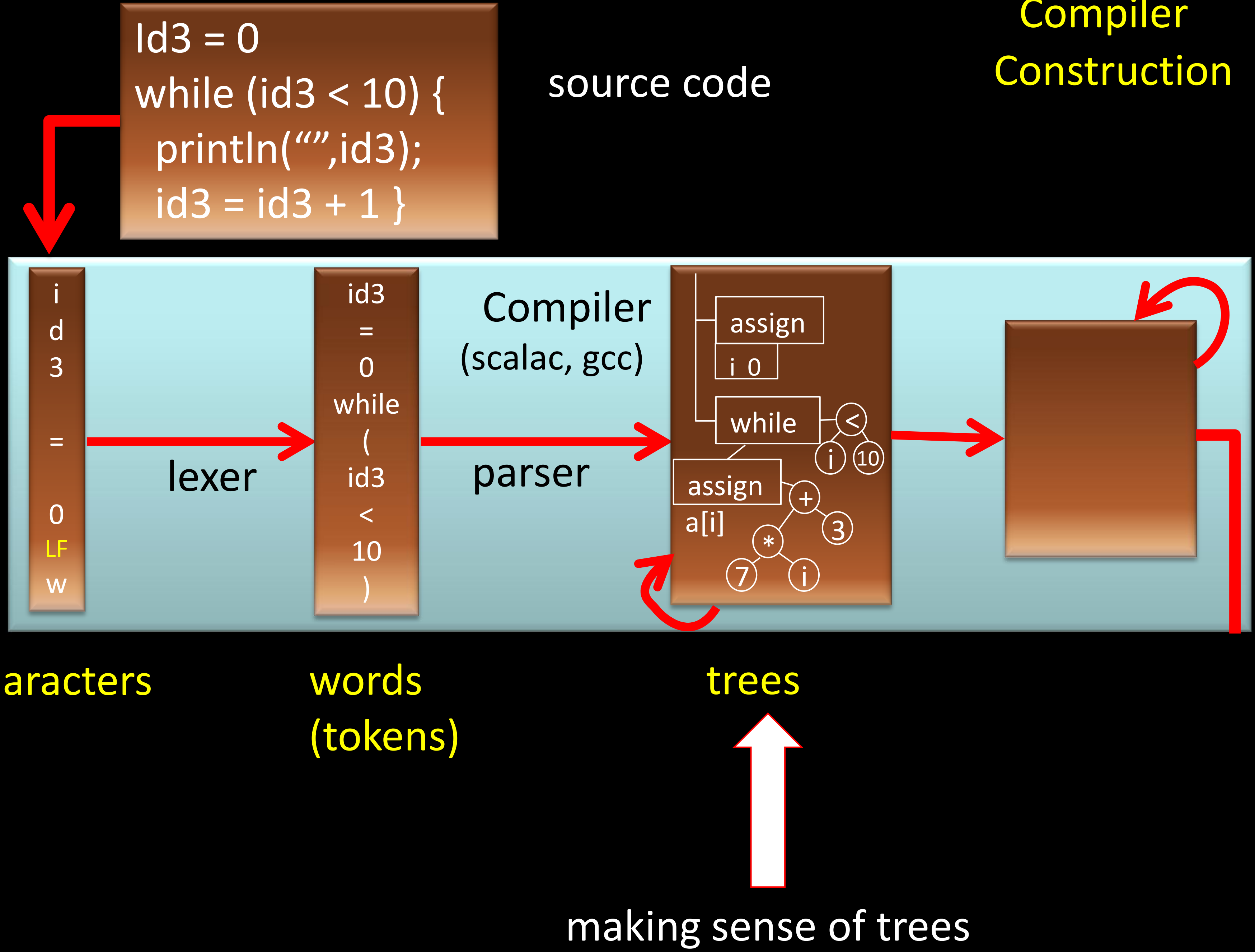
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

http://lara.epfl.ch

# Compiler Construction 2010, Lecture 7

## Type Analysis

# Compiler Construction



```
Id3 = 0
while (id3 < 10) {
  println(“”,id3);
  id3 = id3 + 1 }
```

source code

Compiler
(scalac, gcc)

lexer

parser

characters

words
(tokens)

trees

making sense of trees

# Today

- Type Checking Idea
  - Evaluation and Types
  - Type Rules for Ground Expressions
  - Type Environments
  - Assignments
  - Arrays

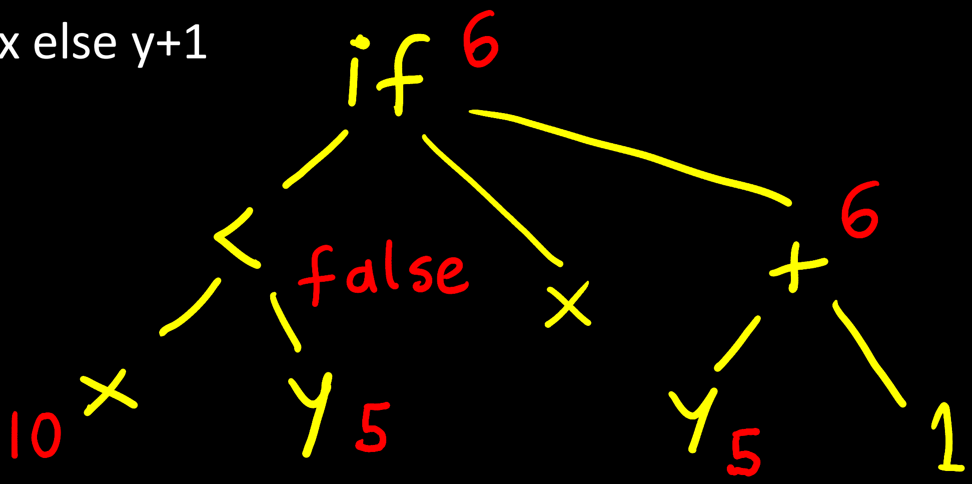
# Evaluating an Expression

scala prompt:

```
def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
min1(10,5)
res1: Int = 6
```

How can we think about this evaluation?

```
x → 10
y → 5
if (x < y) x else y+1
```



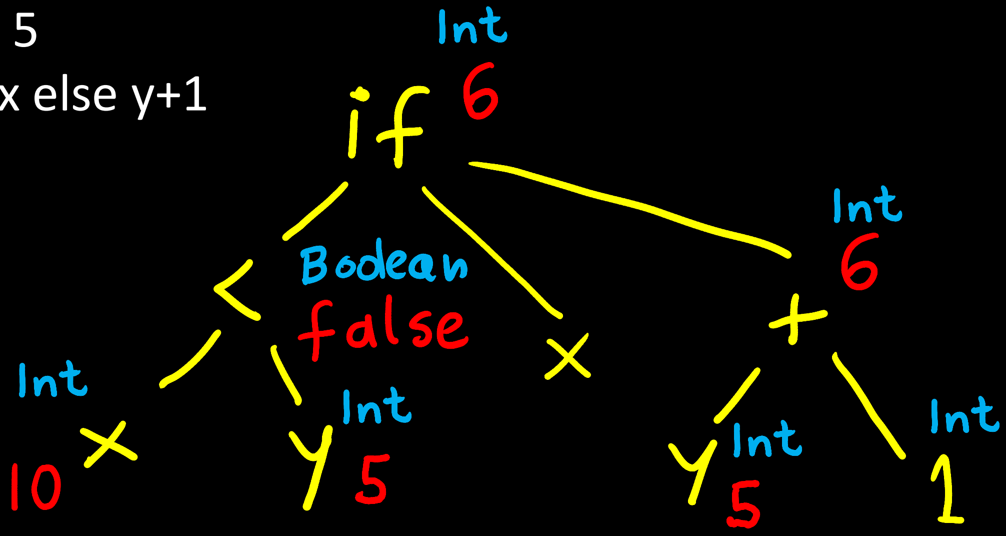
# Each Value has a Type

scala prompt:

```
def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
min1(10,5)
res1: Int = 6
```

How can we think about this evaluation?

x : Int → 10
y : Int → 5
if (x < y) x else y+1



# We can compute types without values

scala prompt:

```
def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
min1(10,5)
res1: Int = 6
```

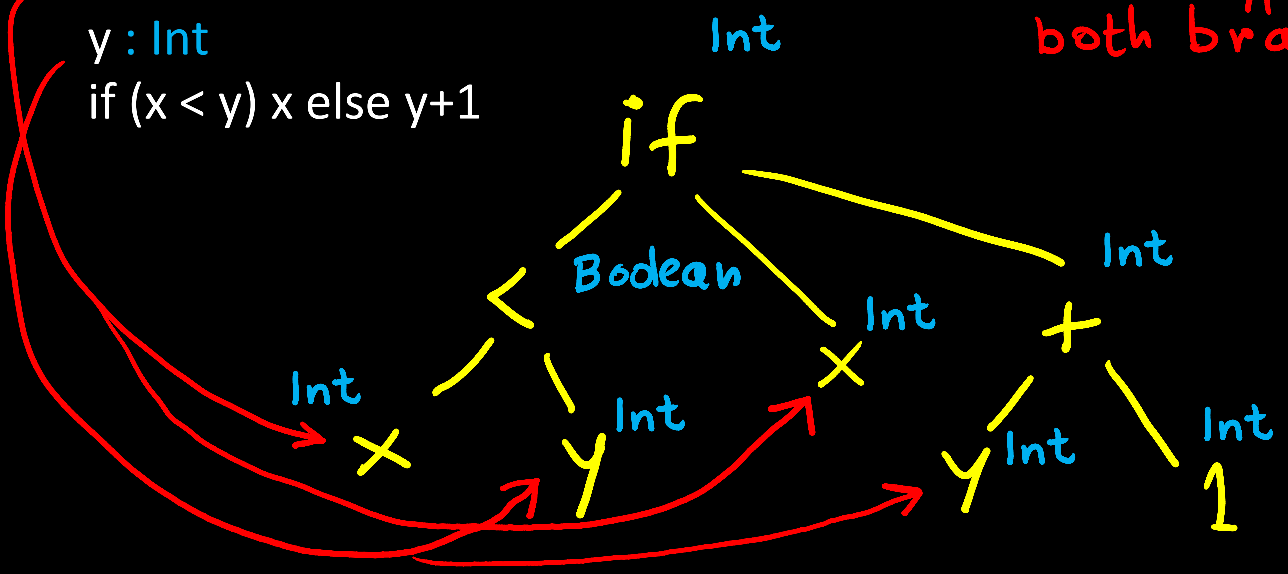
How can we think about this evaluation?

x : Int

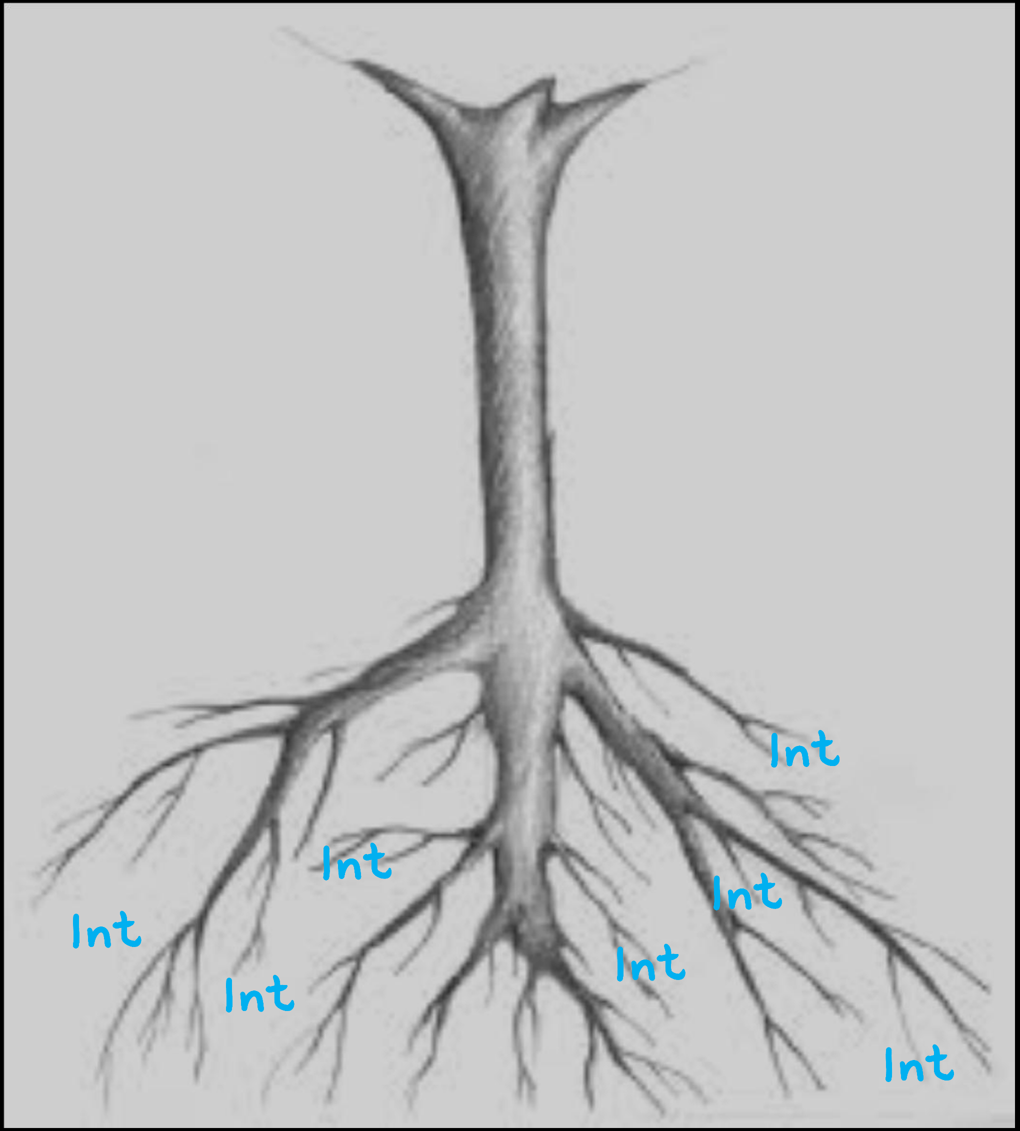
y : Int

if (x < y) x else y+1

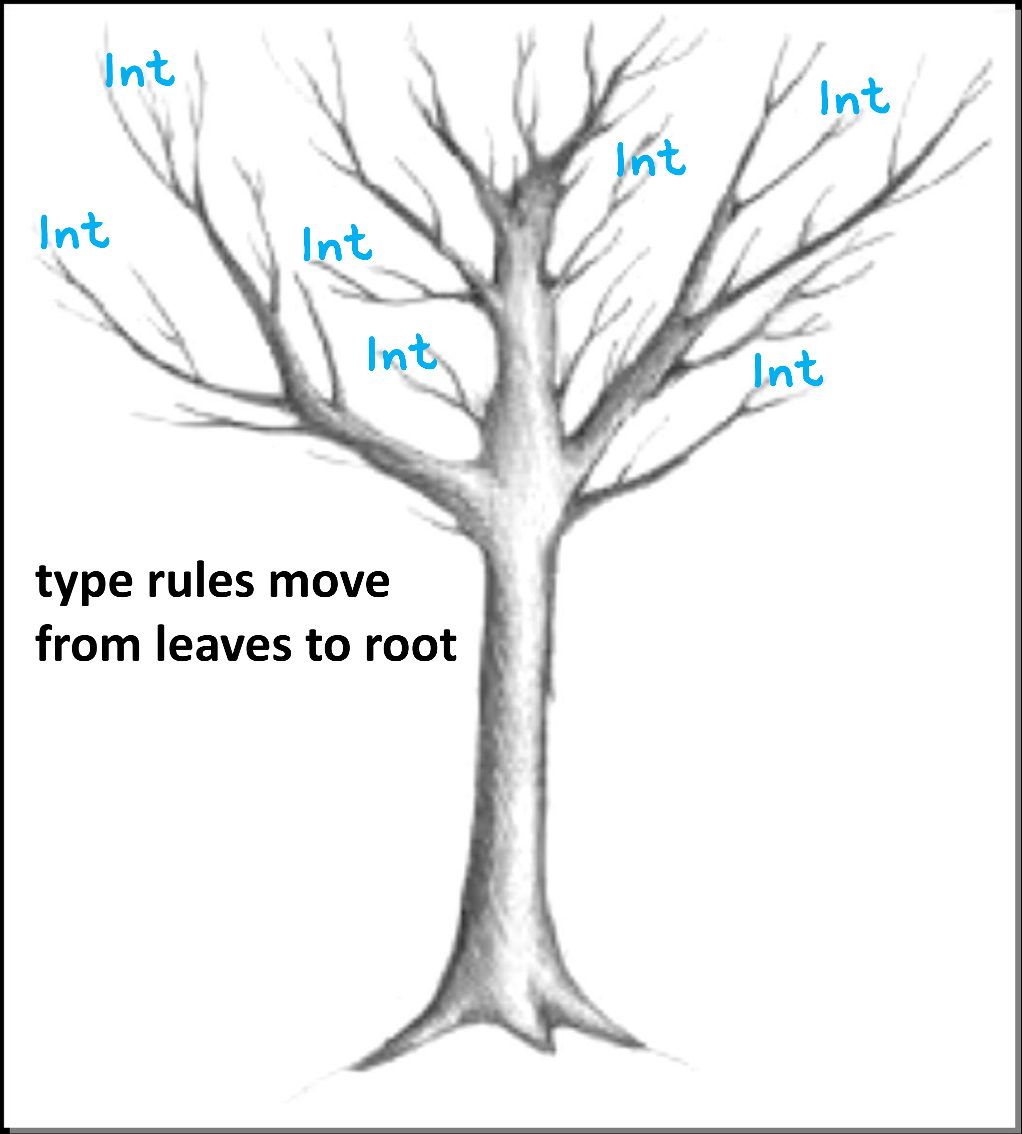
must type check both branches



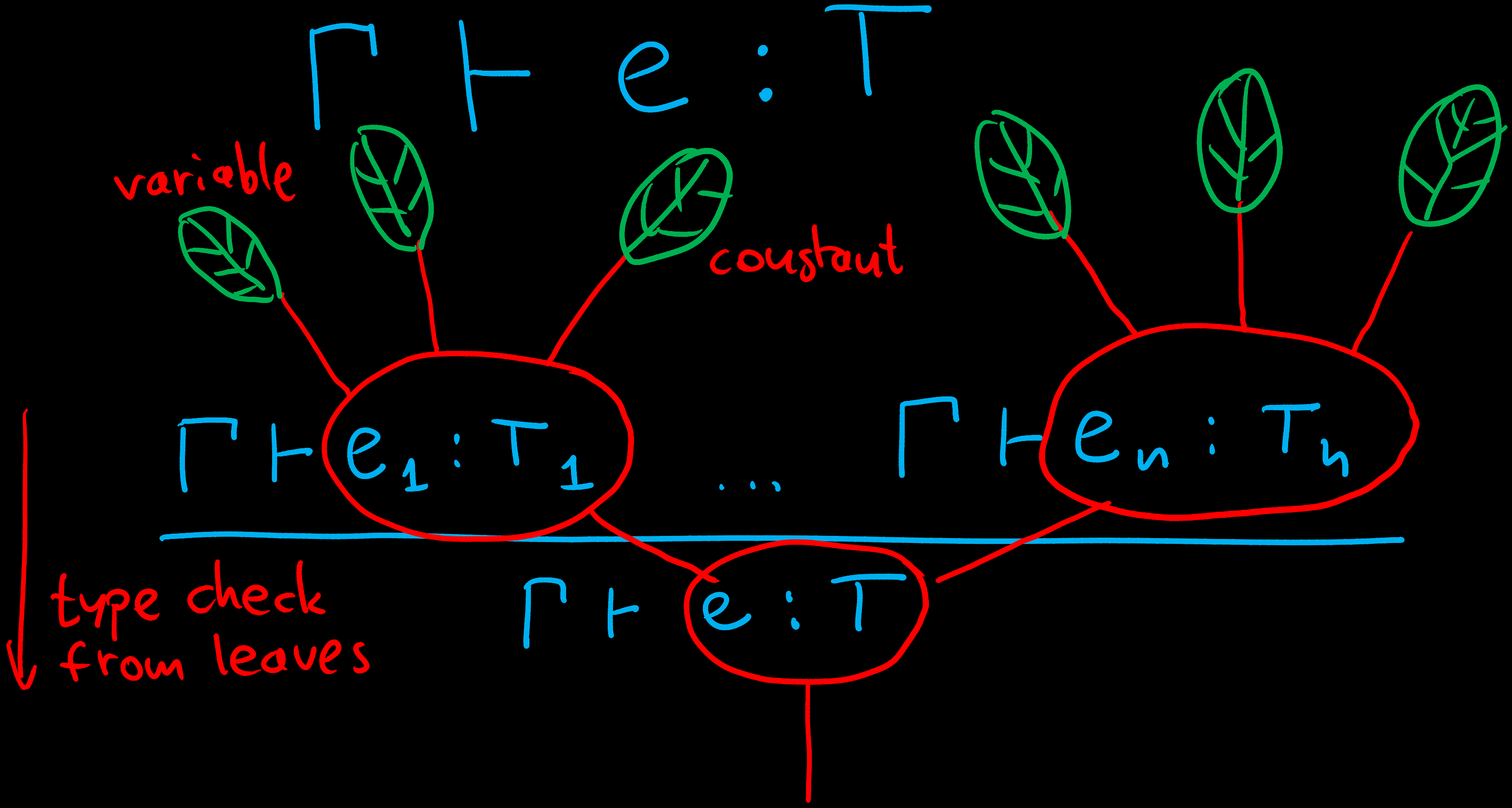
# We do not like trees upside-down



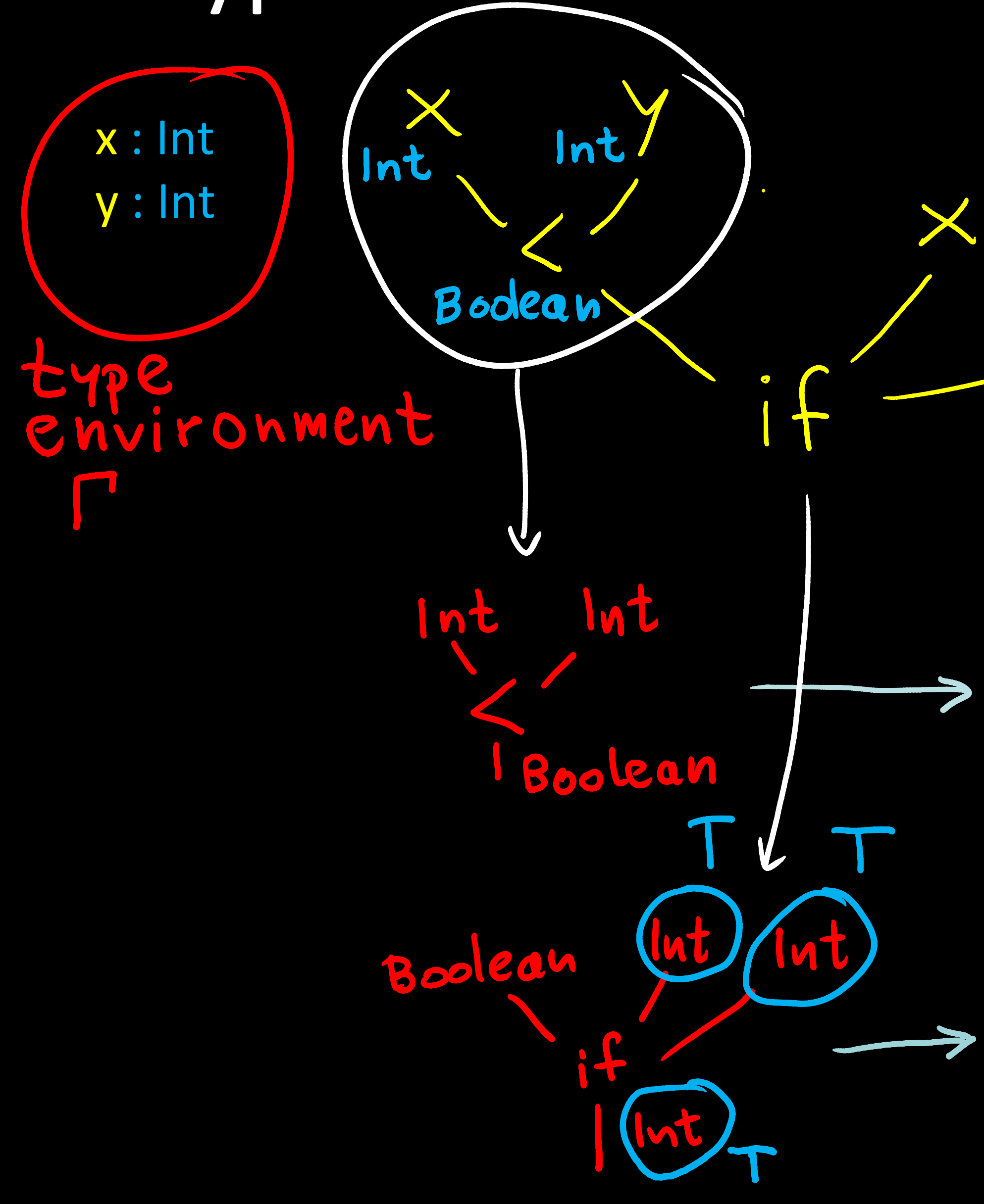
# Leaves are Up



type rules move from leaves to root


$\Gamma \vdash e : T$
variable
constant
$\Gamma \vdash e_1 : T_1$ … $\Gamma \vdash e_n : T_n$
$\Gamma \vdash e : T$
type check from leaves

# Type Rules as Local Tree Constraints



## Type Rules

$$\frac{e_1 : Int \qquad e_2 : Int}{e_1 < e_2 : Boolean}$$

for every type T, if

b has type Boolean, and ...

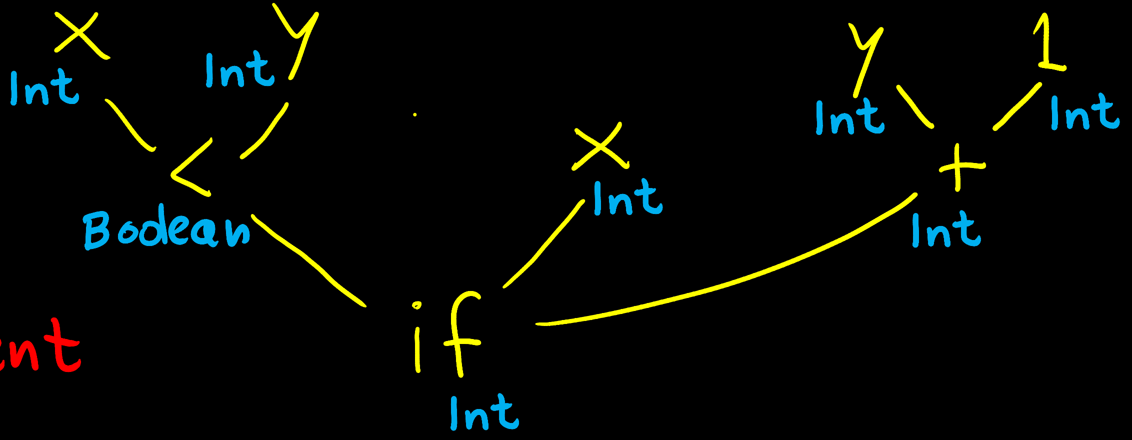
then

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

# Type Rules with Environment

x : Int
y : Int

type environment
$\Gamma$



## Type Rules

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

$$\frac{}{\text{Int Const}(k):\text{Int}}$$

$$\frac{\Gamma\vdash e_1:\text{Int} \qquad \Gamma\vdash e_2:\text{Int}}{\Gamma\vdash(e_1<e_2):\text{Boolean}}$$

...(then) in the (same) environment $\Gamma$ the expression $e_1<e_2$ has type Bool.

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

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

$$\Gamma \vdash e : T$$

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

$$\frac{\Gamma \vdash e_1 : T_1 \quad \ldots, \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash e : T}$$

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

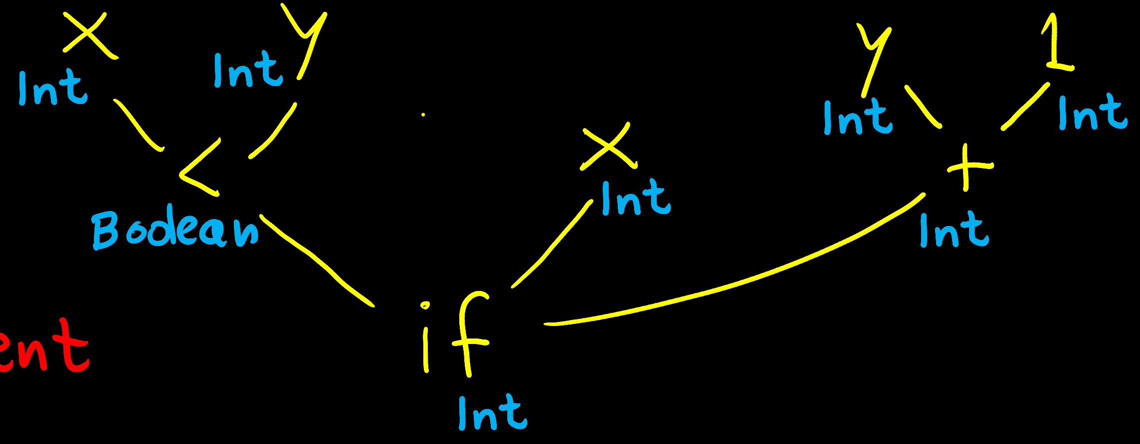
# Derivation Using Type Rules

x : Int

y : Int

type environment

$\Gamma$



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

$$\frac{\dfrac{\dfrac{(x,Int)\in\Gamma}{\Gamma \vdash x : Int} \qquad \dfrac{(y,Int)\in\Gamma}{\Gamma \vdash y : Int}}{\Gamma \vdash (x<y) : Boolean} \qquad \dfrac{(x,Int)\in\Gamma}{\Gamma \vdash x : Int} \qquad \dfrac{\dfrac{(y,Int)\in\Gamma}{\Gamma \vdash y : Int} \qquad \dfrac{}{\Gamma \vdash 1 : Int}}{\Gamma \vdash (y+1) : Int}}{\Gamma \vdash (\text{if}(x<y)\ x\ \text{else}\ y+1) : Int}$$

# Type Rule for Function Application

We can treat operators as variables that have function type

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

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

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

We can replace many previous rules with application rule:

$$\boxed{\dfrac{\Gamma \vdash e_1 : T_1 \quad \ldots \quad \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : (T_1 \times \cdots \times T_n \to T)}{\Gamma \vdash f(e_1, \ldots, e_n) : T}}$$

# Computing the Environment of a Class

```
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))
  }
}
```

$\Gamma_0 = \{$

$(data, Int),$

$(name, String),$

$(m, Int \times Int \to Boolean),$

$(n, Int \to Int),$

$(p, Int \to Int)$

$\}$

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

```
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))
  }
}
```

$$\Gamma_0 = \{ (data, Int), (name, String), (m, Int \times Int \to Boolean), (n, Int \to Int), (p, Int \to Int) \}$$

$\leftarrow \Gamma_0$

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

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

$$\frac{\Gamma_2 \vdash k : Int \qquad \dfrac{\Gamma_2 \vdash k : Int \qquad \Gamma_2 \vdash n : Int \to Int}{\Gamma_2 : n(k) : Int} \qquad \Gamma_2 \vdash m : Int \times Int \to \boxed{Int}}{\Gamma_2 \vdash m(k, n(k)) : \boxed{Int}}$$

Bool

Bool

# Remember Function Updates

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

# Type Rule for Method Bodies

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

# Type Rule for Assignments

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

# Type Rules for Block: { var $x_1$:$T_1$ ... var $x_n$:$T_n$; $s_1$; ... $s_m$; e }

$$\frac{(\Gamma \oplus \{(x_1,T_1)\}) \oplus \dots \oplus \{(x_n,T_n)\} \vdash \begin{array}{l} s_1 : \mathrm{void} \\ \quad \dots \\ s_n : \mathrm{void} \\ e : T \end{array}}{\Gamma \vdash \{\ \mathrm{var}\ x_1 : T_1 ; \dots ; \mathrm{var}\ x_n : T_n ; s_1 ; \dots ; s_n ; e\ \} : T}$$

# Blocks with Declarations in the Middle

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

just expression

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

empty

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

declaration is first

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

statement is first

# Rule for While Statement

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

# Rule for Method Call

```
class T_0 {
  ...
  def m(x_1:T_1, ..., x_n:T_n):T = {
    ...
  }
  ...
}
```

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

# Example to Type Check

object

```
class 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 }
  }
}
```

$\Gamma_0$

$\Gamma_0 = \{ (z, Boolean), (u, Int), (f, Boolean \to Int) \}$

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

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

$$\frac{\quad}{\Gamma_1 \vdash}$$

# Overloading of Operators

$\text{Int} \times \text{Int} \to \text{Int}$

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

Not a problem for type checking from leaves to root

$\text{String} \times \text{String} \to \text{String}$

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

# Arrays

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

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

Assigning to an array

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

# Example with Arrays

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

$$\Gamma = \{ (a, Array(Int)), (k, Int) \}$$

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

$$\Gamma \vdash a[a[k]] : Int \;\text{ from }\; \dfrac{\Gamma \vdash a : Array(Int) \quad \dfrac{\Gamma \vdash a : Array(Int) \quad \Gamma \vdash k : Int}{\Gamma \vdash a[k] : Int}}{\Gamma \vdash a[a[k]] : Int}$$

# Type Rules (1)

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

variable

$$\frac{}{\mathit{Int\,Const}(k):\mathit{Int}}$$

constant

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

function application

$$\frac{\Gamma\vdash e_1:\mathit{Int} \qquad \Gamma\vdash e_2:\mathit{Int}}{\Gamma\vdash (e_1+e_2):\mathit{Int}}$$

plus

$$\frac{\Gamma\vdash e_1:\mathit{String} \qquad \Gamma\vdash e_2:\mathit{String}}{\Gamma\vdash (e_1+e_2):\mathit{String}}$$

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

if

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

assignment

$$\frac{\Gamma\vdash b:\mathit{Boolean} \qquad \Gamma\vdash s:\mathit{void}}{\Gamma\vdash (\mathrm{while}\ (b)\ s):\mathit{void}}$$

while

# Type Rules (2)

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

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

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

block

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

array use

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

array assignment

# Type Rules (3)

$\Gamma^{C}$ – top-level environment of class C

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

$$\Downarrow$$

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

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

method invocation

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

field use

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

field assignment

# Type Rules (1)

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

variable

$$\frac{}{\mathrm{Int}\ \mathrm{Const}(k):\mathrm{Int}}$$

constant

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

function application

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

plus

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

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

if

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

assignment

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

while

# Type Rules (2)

$$\left.\begin{array}{c}
\dfrac{\Gamma \vdash e : T}{\Gamma \vdash \{e\} : T} \qquad \dfrac{}{\Gamma \vdash \{\} : \text{void}} \\[2ex]
\dfrac{\Gamma \oplus \{(x,T_1)\} \vdash \{t_2; \ldots; t_n\} : T}{\Gamma \vdash \{\text{var } x : T_1; t_2; \ldots; t_n\} : T} \\[2ex]
\dfrac{\Gamma \vdash s_1 : \text{void} \qquad \Gamma \vdash \{t_2; \ldots; t_n\} : T}{\Gamma \vdash \{s_1; t_2; \ldots; t_n\} : T}
\end{array}\right\} \text{block}$$

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

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

# Type Rules (3)

$\Gamma^C$ – top-level environment of class C

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

$$\Downarrow$$

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

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

method invocation

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

field use

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

field assignment

# 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 establishe 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

# 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
- Main rule for subtyping $\approx$ 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

$Pos <: Int$

$Neg <: Int$

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

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

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

type checks

$Pos \subseteq Int$

$Neg \subseteq Int$

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

$$\frac{x \in Pos \qquad y \in Neg}{x \cdot y \in Neg}$$

$$\frac{x \in Pos \qquad \overbrace{y \in Pos}^{y \text{ not zero}}}{\underbrace{x/y}_{\text{well defined}} \in Pos}$$

# More Rules

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

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

More rules for division?

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

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

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

# Making Rules Useful

- Let x be a variable

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

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

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

← type system proves: no division by zero!

# Subtyping Example

Pos <: Int

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

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

→ q = f(p)

- type checks

$\Gamma :$

$f : Int \to Pos$

$p : Pos \qquad (p, Pos) \in \Gamma$

$q : Int$

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

# Using Subtyping

```
Pos <: Int

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

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

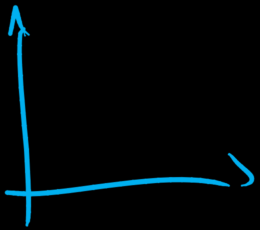
```
Pos <: Int

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

var p : Int
var q : Int

q = f(p)
```

- does not type check

# Subtyping for Products

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

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



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

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

So, we might as well add

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

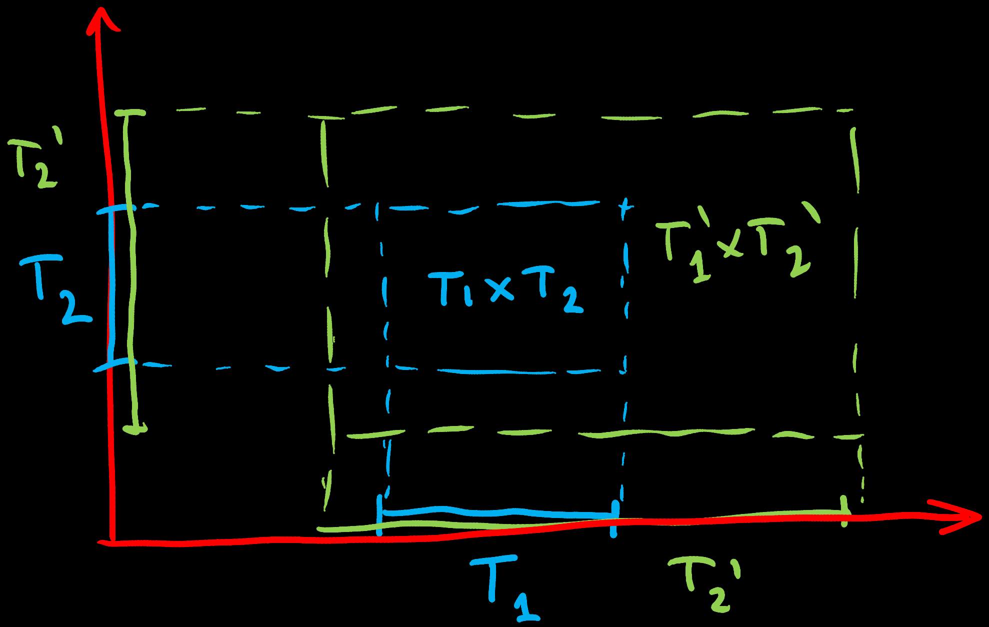
covariant subtyping for pairs

$\text{Pair}[T_1, T_2]$

# Analogy with Cartesian Product

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

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



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

# Subtyping and Function Types

# Subtyping for Function Types

$$T_1 <: T_2 \quad \underset{\xleftarrow{\qquad}}{\xrightarrow{\text{implies}}} \quad \text{for all } e: \quad \frac{\Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

$$\text{Function}[\underset{\text{contra-}}{\underset{\uparrow}{T_1}}, \underset{\text{co-}}{\underset{\nwarrow}{T}}]$$

$$\frac{\overbrace{T_1' <: T_1 \quad \dots \quad T_n' <: T_n}^{\text{contravariance}} \quad \overbrace{T <: T'}^{\text{covariance}}}{T_1 \times \dots \times T_n \to T \; <: \; T_1' \times \dots \times T_n' \to T'}$$

Consequence:

$$\frac{\boxed{\Gamma \vdash m : T_1 \times \dots \times T_n \to T} \quad \frac{\boxed{\Gamma \vdash e_1 : T_1'} \; \boxed{T_1' <: T_1}}{\Gamma \vdash e_1 : T_1} \;\dots\; \frac{\boxed{\Gamma \vdash e_n : T_n'} \; \boxed{T_n' <: T_n}}{\Gamma \vdash e_n : T_n}}{\frac{\Gamma \vdash m(e_1, \dots, e_n) : T \qquad \boxed{T <: T'}}{\boxed{\Gamma \vdash m(e_1, \dots, e_n) : T'}}}$$

as if $\Gamma \vdash m : T_1' \times \dots \times T_n' \to T'$

# Function Space as Set

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

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

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

$\neq$

$\subseteq T_1 \times T_2$

$f : D \to D$

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

We can prove

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

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

# Proof

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

Let $T_1' \subseteq T_1$ and $T_2 \subseteq T_2'$.

Let $f \in T_1 \to T_2$

Thus $\forall x.\ x \in T_1 \to f(x) \in T_2$

Let $x \in T_1'$. From $T_1' \subseteq T_2$, also $x \in T_1$

Thus $f(x) \in T_2$. By $T_2 \subseteq T_2'$, also $f(x) \in T_2'$

Thus, $\forall x.\ x \in T_1' \to f(x) \in T_2'$

Therefore, $f \in T_1' \to T_2'$

Thus, $T_1 \to T_2 \subseteq T_1' \to T_2'$.

# 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

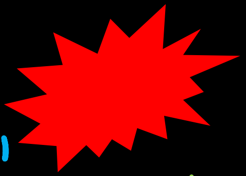
# Type System for *Tootool 0.1*

Pos <: Int

Neg <: Int

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

assignment



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

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)
```

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

Runtime error: intSqrt invoked with a negative argument!

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

# What went wrong in *Tootool 0.1* ?

Pos <: Int

Neg <: Int

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

assignment

GUILTY!

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

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)
```

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

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

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

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

# Recall Our Type Derivation

Pos <: Int

Neg <: Int

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

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

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

Runtime error: intSqrt invoked with a negative argument!

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

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

# Corrected Type Rule for Assignment

Pos <: Int

Neg <: Int

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

assignment GUILTY!

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

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)
```

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

does not type check

x must be able to store any value from T

e can have any value from T

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

$\Gamma$ 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

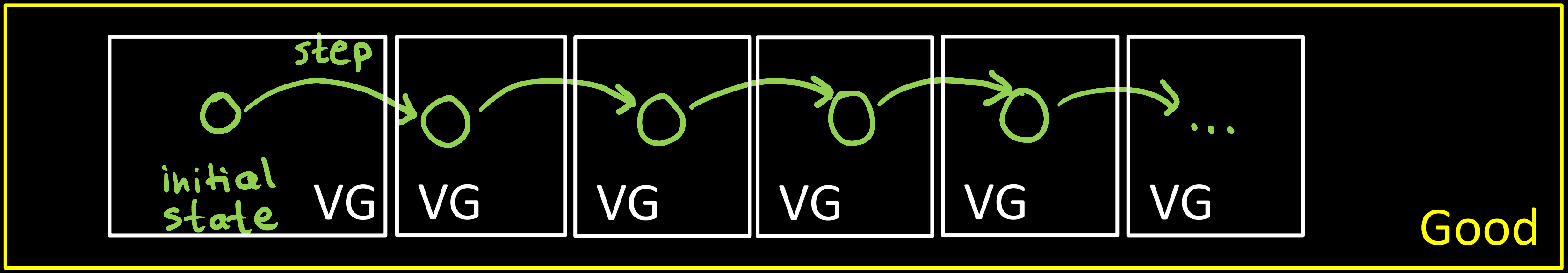
# Proving Soundness of Type Systems

- 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.)

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

Type rules:

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

$$Pos <: Int$$

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

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

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

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

# 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 + x
```

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
```

position in source

```
x = x + z
y = x / z
z = z + x
```

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 ∉ 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.)

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

## Type rules:

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

$$Pos <: Int$$

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

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

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

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

# Checking Properties in Our Case

Holds: in initial state, variables are =1

$1 \in Pos$
$1 \in Int$ ✓

- 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.*

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

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

values of variables:

x = 1
y = 3
z = 2

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.

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

Type rules:

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

$$Pos <: Int$$

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

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

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

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

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

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

```
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
```

$\Gamma$

type checks.

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

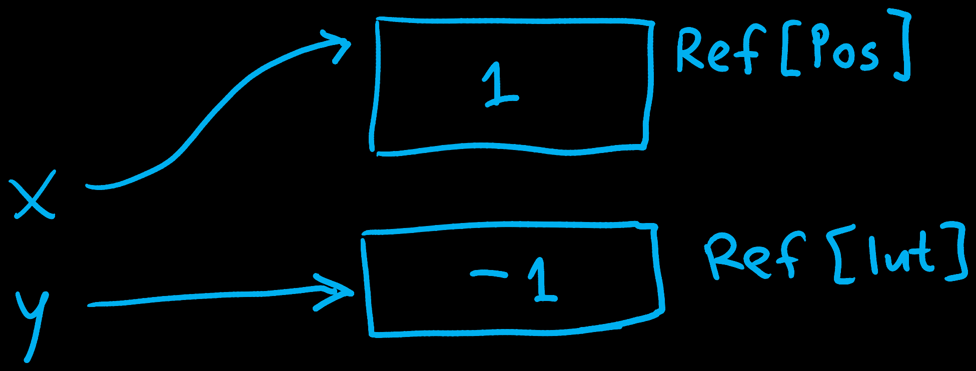
# Simple Parametric Class

```
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
```



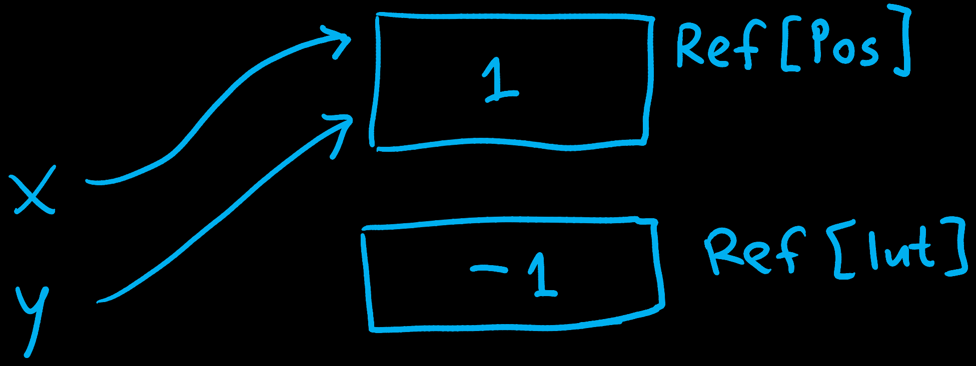
# Simple Parametric Class

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

Can we use the subtyping rule

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

```
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

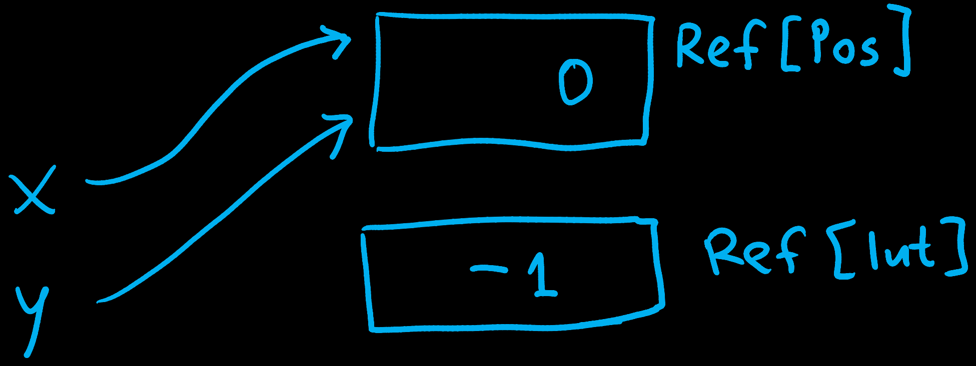
```
class Ref[T](var content : T)
```

Can we use the subtyping rule

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

```
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
```

CRASHES



# Analogously

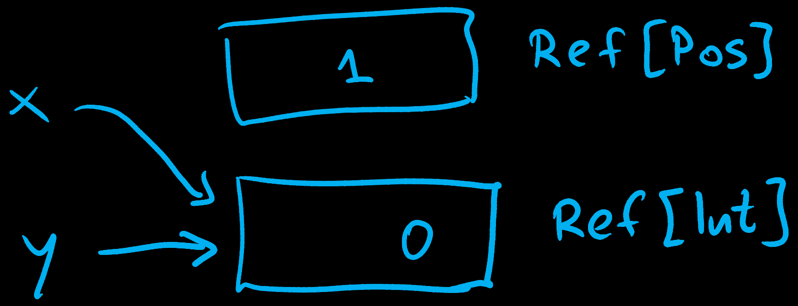
```
class Ref[T](var content : T)
```

Can we use the converse 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
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]
```

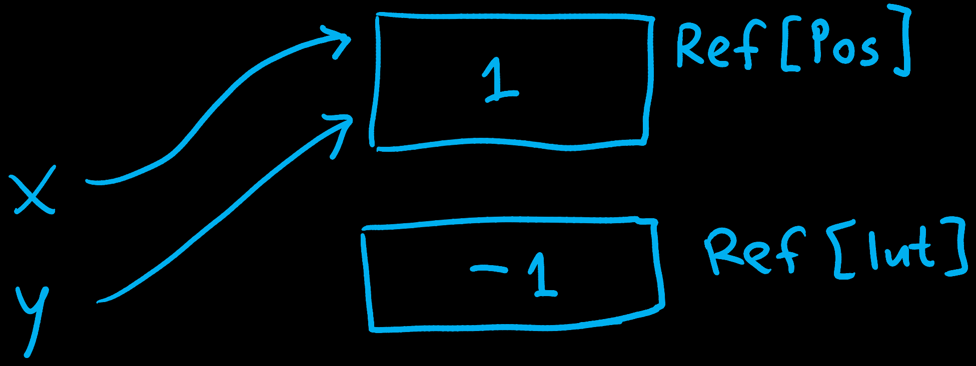
# Case in Soundness Proof Attempt

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

Can we use the subtyping rule

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

```
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
```



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’