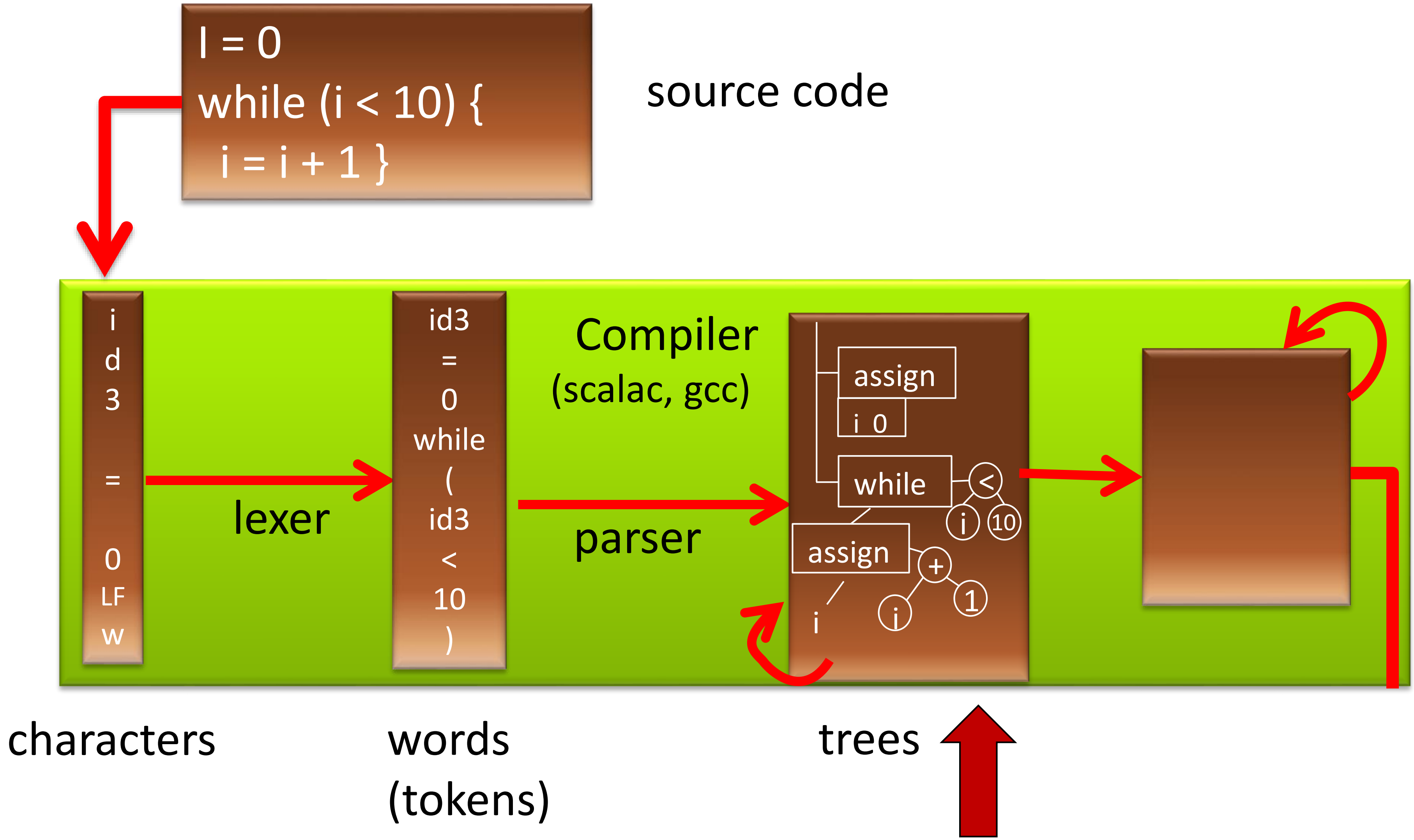


Type Checking

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)

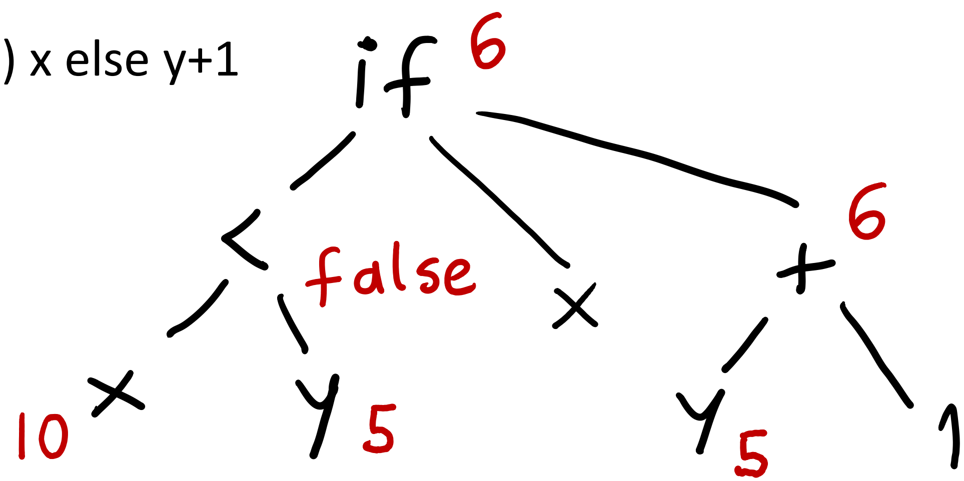
res1: Int = 6

How can we think about this evaluation?

x → 10

y → 5

if (x < y) x else y+1



Computing types using the evaluation tree

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)

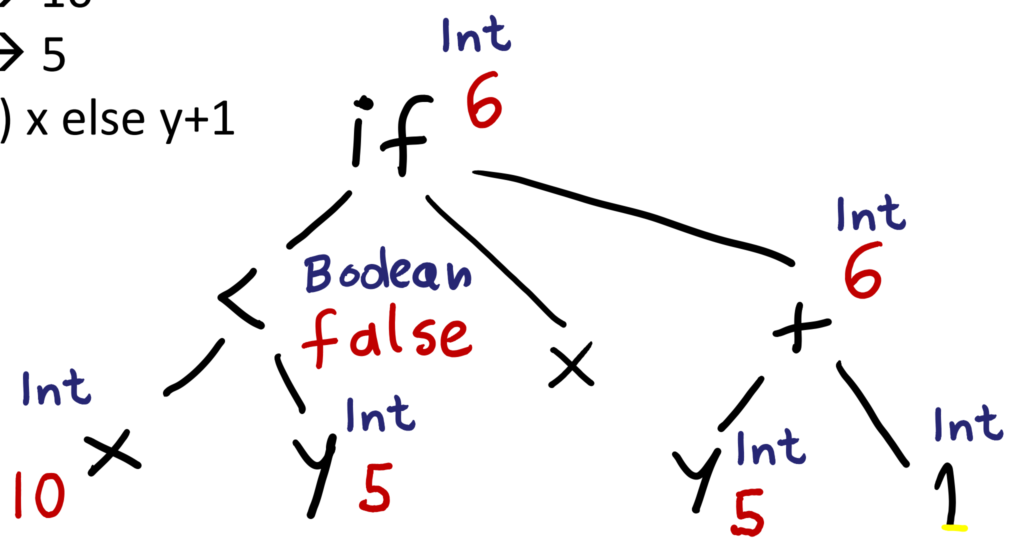
res1: Int = 6

How can we think about this evaluation?

x : Int → 10

y : Int → 5

if (x < y) x else y+1



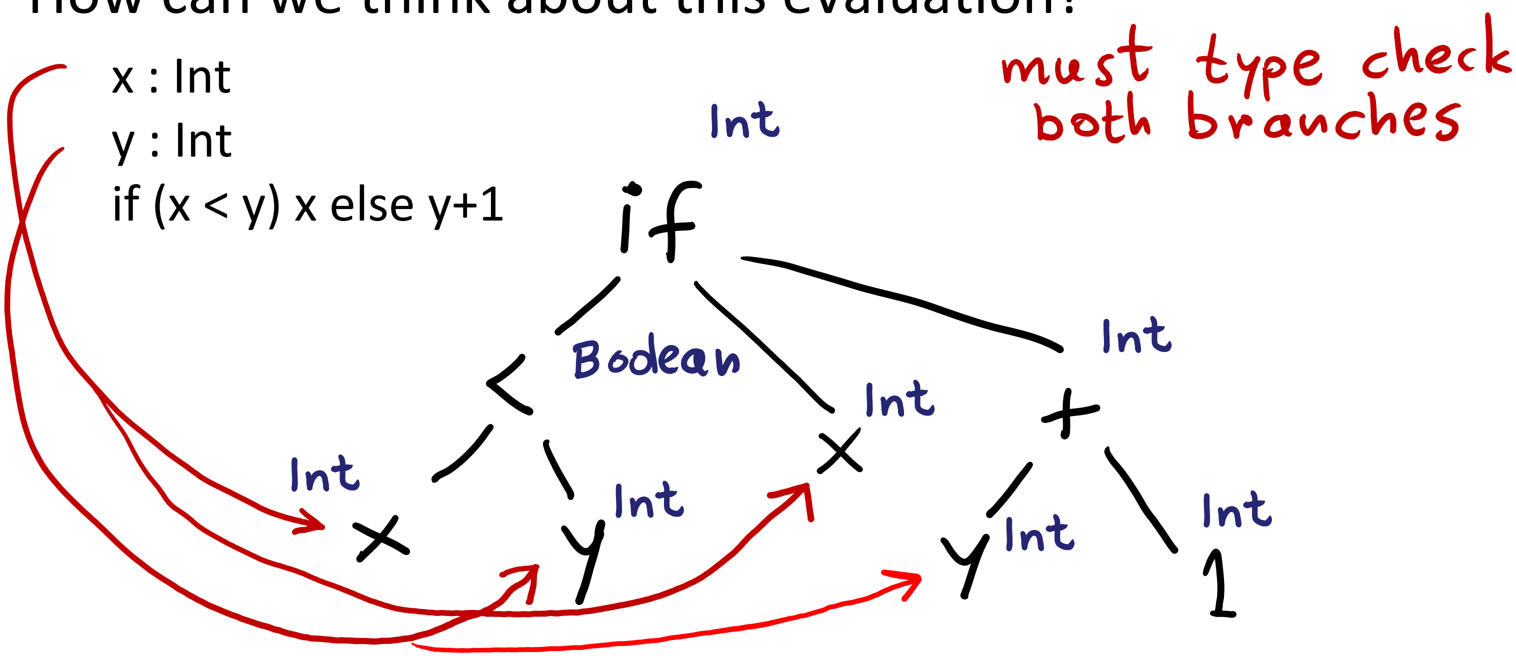
We can compute types without values

scala prompt:

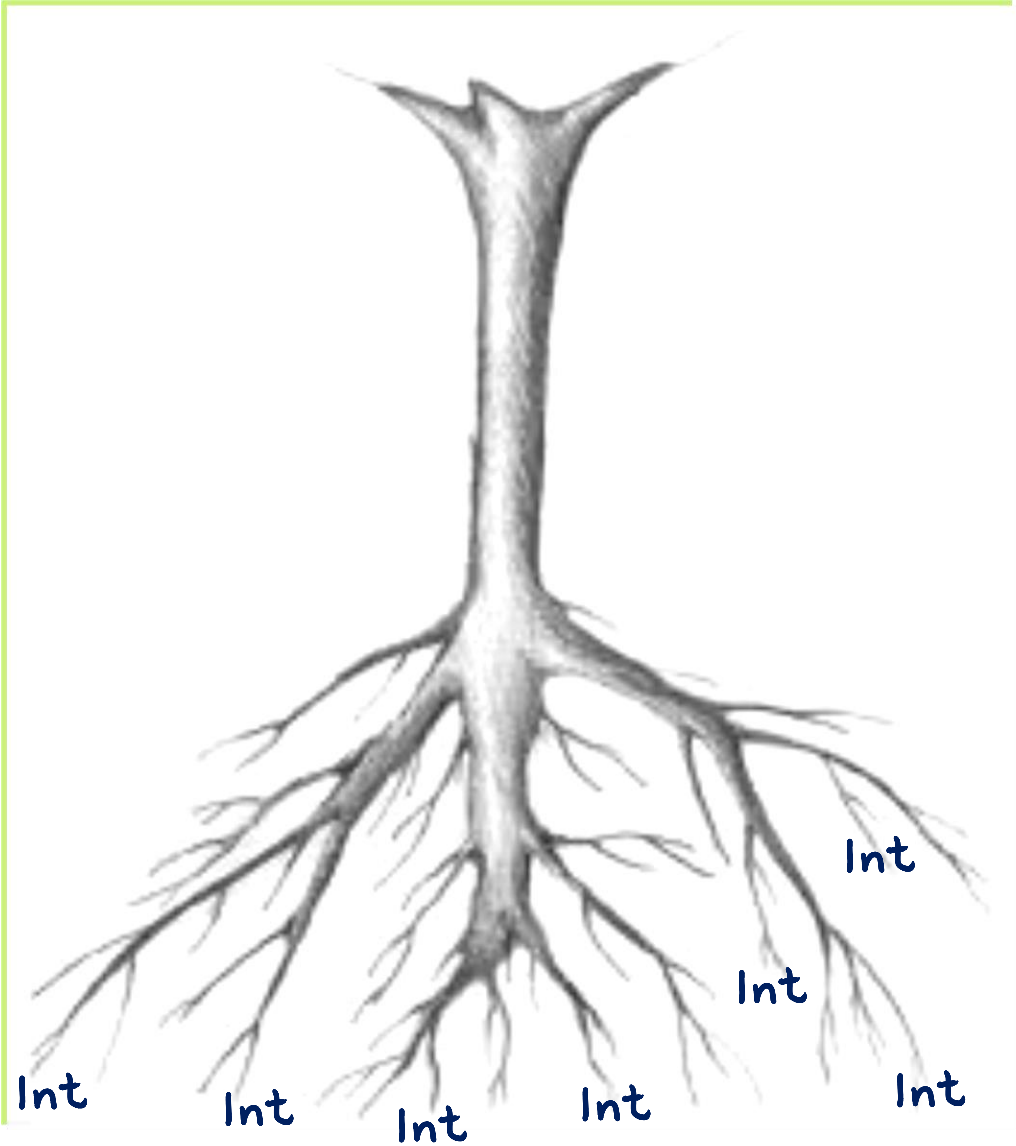
```
>def min1(x : Int, y : Int) : Int = { if (x < y) x else y+1 }
min1: (x: Int,y: Int)Int
>min1(10,5)
res1: Int = 6
```

How can we think about this evaluation?

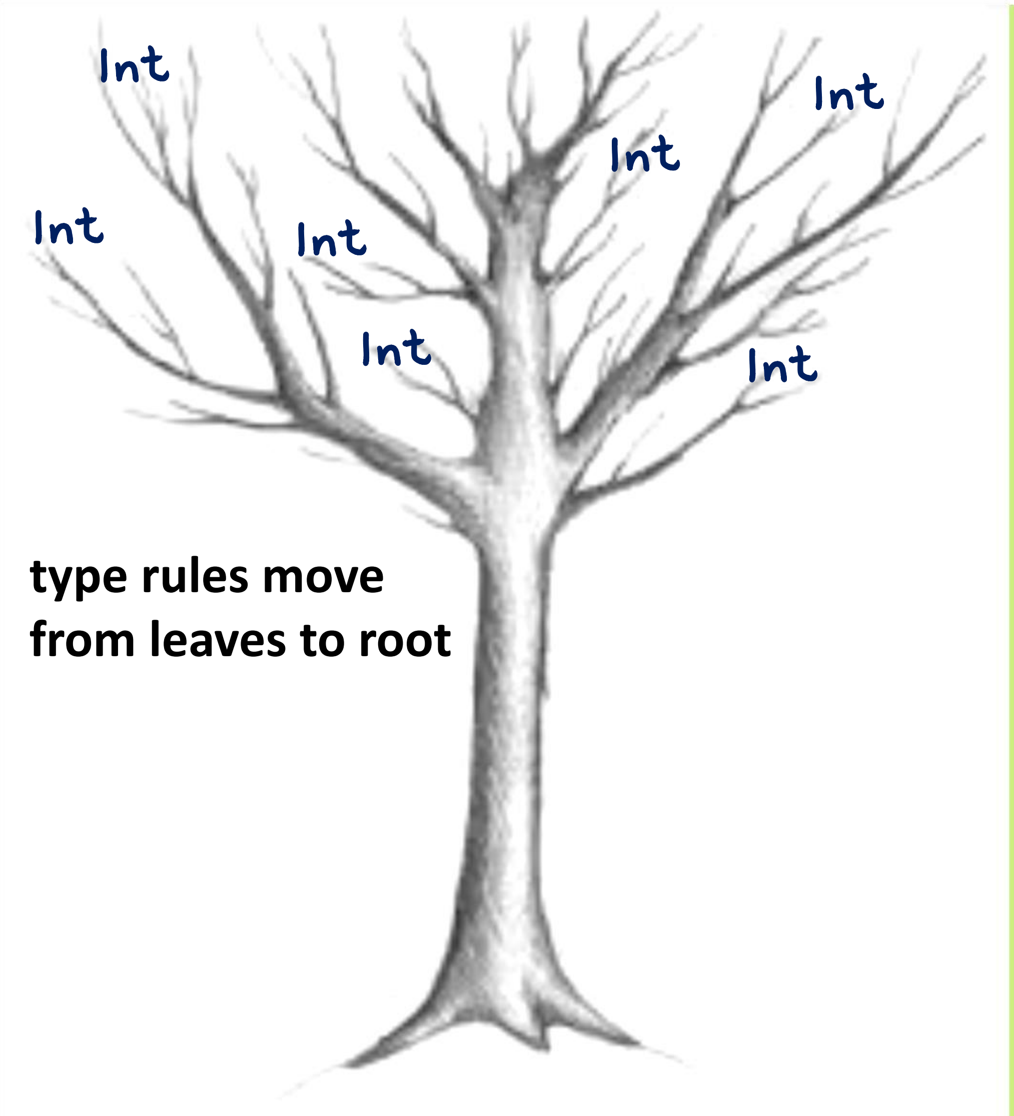
x : Int
y : Int
if (x < y) x else y+1



We do not like trees upside-down



Leaves are Up



Type Judgements and Type Rules

- e type checks to T under Γ (type environment)

$$\Gamma \vdash e : T$$

  - Types of constants are predefined
  - Types of variables are specified in the source code
- If e is composed of sub-expressions

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

↓ type check from leaves

Type Judgements and Type Rules

$$\Gamma \vdash e : T$$

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

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

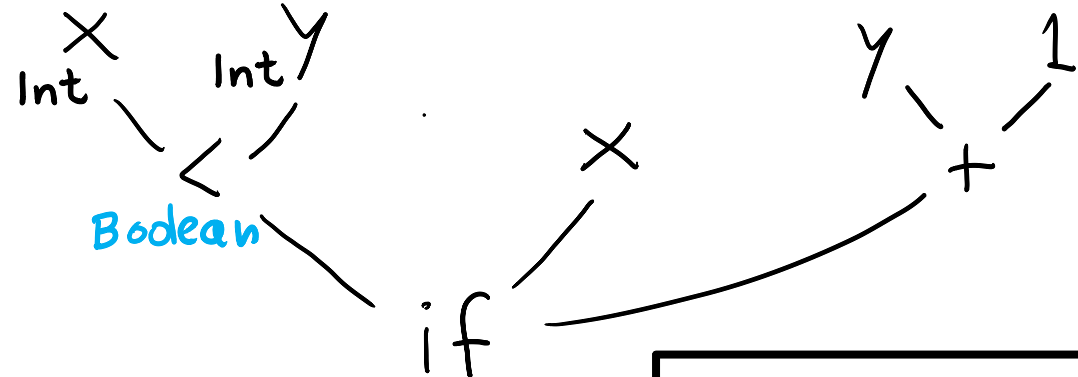
If e_1 type checks in gamma and has type T_1 and ... and e_n type checks in gamma and has type T_n then e type checks in gamma and has type T

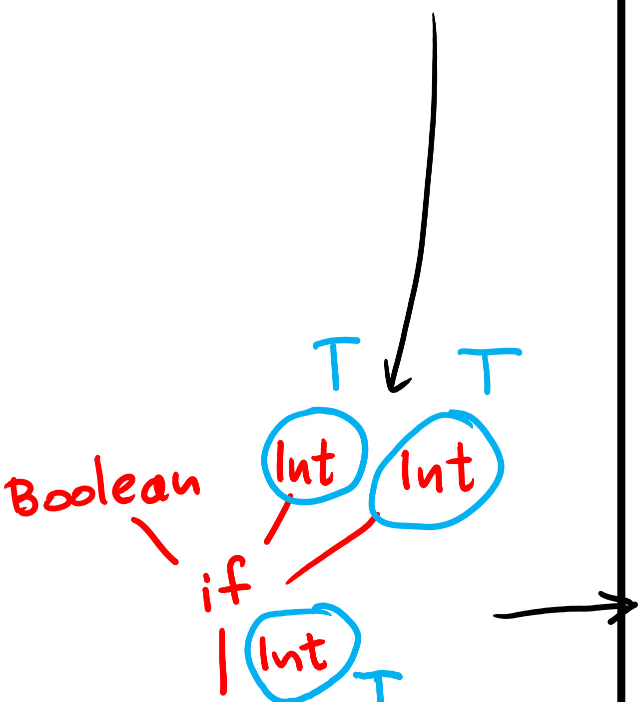
type rule

Type Rules as Local Tree Constraints

x : Int

y : Int





Type Rules

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

for every type T, if

b has type Boolean, and ...

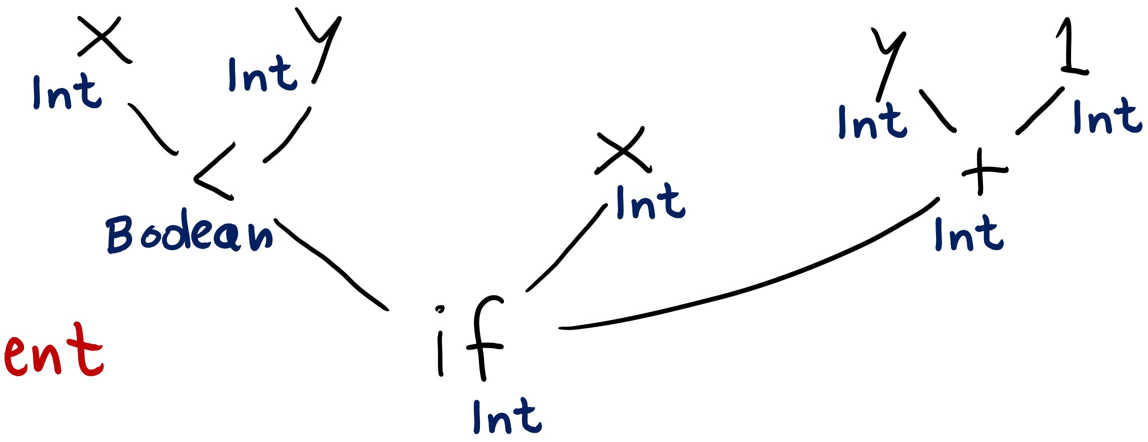
then

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

Type Rules with Environment

x : Int
y : Int

type environment Γ



Type Rules

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

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

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

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

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

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

Type Checker Implementation Sketch

```
def typeCheck(Γ : Map[ID, Type],  e : ExprTree) : TypeTree = {
 e match {
   case Var(id) => {  ??    }
   case If(c,e1,e2) => { ?? }
   ...
}}

   case Var(id) => { Γ(id) match
     case Some(t) => t
     case None => error(UnknownIdentifier(id,id.pos))
   }
```

Type Checker Implementation Sketch

- **case** If(c,e1,e2) => {

  val tc = typeCheck(Γ,c)

  if (tc != BooleanType) error(IfExpectsBooleanCondition(e.pos))

  val t1 = typeCheck(Γ, e1); **val** t2 = typeCheck(Γ, e2)

  if (t1 != t2) error(IfBranchesShouldHaveSameType(e.pos))

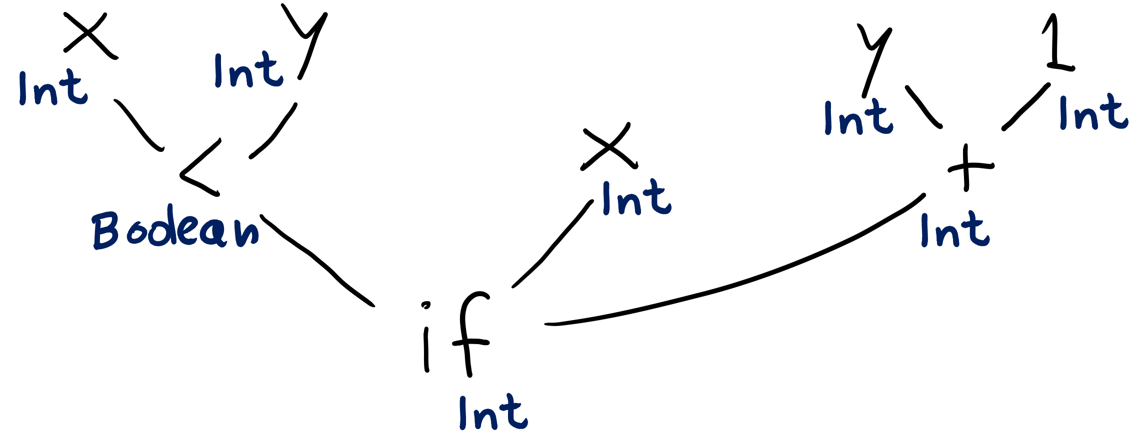
  t1

  }

Derivation Using Type Rules

x : Int

y : Int



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

$$\dfrac{\dfrac{(x, Int) \in \Gamma}{\Gamma \vdash x : Int} \quad \dfrac{(y, Int) \in \Gamma}{\Gamma \vdash y : Int}}{\Gamma \vdash (x < y) : Boolean} \quad \dfrac{(x, Int) \in \Gamma}{\Gamma \vdash x : Int} \quad \dfrac{\dfrac{(y, Int) \in \Gamma}{\Gamma \vdash y : Int} \quad \dfrac{}{\Gamma \vdash 1 : Int}}{\Gamma \vdash (y+1) : Int}$$

$$\overline{\Gamma \vdash (\text{if}\,(x<y)\ x\ \text{else}\ y+1) : Int}$$

Type Rule for Function Application

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

Type Rule for Function Application [Cont.]

We can treat operators as variables that have function type

$+ : Int \times Int \rightarrow Int$

$< : Int \times Int \rightarrow Boolean$

$\&\& : Boolean \times Boolean \rightarrow Boolean$

We can replace many previous rules with application rule:

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \quad \Gamma \vdash e_2 : \mathrm{Bool} \quad \Gamma \vdash \&\& : (\mathrm{Bool} \times \mathrm{Bool}) \rightarrow \mathrm{Bool}}{\Gamma \vdash e_1 \;\&\&\; e_2 : \mathrm{Bool}}$$

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 x Int → Boolean),
(n, Int → Int),
(p, Int → Int)
$\}$

We can 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 = {        ← Γ0
                                  ← Γ1 = Γ0 ⊕ { (r, Int) }
    var k:Int = r + 2             ← Γ2 = Γ1 ⊕ { (k, Int) } = Γ0 ∪ { (r, Int), (k, Int) }
    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 x Int → Boolean), (n, Int → Int), (p, Int → Int) $\}$

← Γ_0

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

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

Remember

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

$$\frac{\Gamma_2 \vdash \mathrm{k}: \mathrm{Int} \qquad \dfrac{\Gamma_2 \vdash n: \mathrm{Int} \to \mathrm{Int} \quad \Gamma_2 \vdash k: Int}{\Gamma_2 \vdash n(\mathrm{k}): \mathrm{Int}} \qquad \Gamma_2 \vdash \mathrm{m}: \mathrm{Int} \times \mathrm{Int} \to \mathrm{Bool}}{\Gamma_2 \vdash m\big(k, n(k)\big): Bool}$$

Type Rule for Method Definitions

$\text{def } m(x_1 : T_1, \cdots, x_n : T_n) : T = e$

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

Type Rule for Assignments

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

Unit

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

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

Blocks with Declarations in the Middle

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

just expression

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

empty

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

declaration is first

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

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 a Method Call

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

$\Gamma_{T_0} \vdash m : T_0 \times T_1 \times \ldots \times T_n \to T$

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

$m(x, e_1, \ldots, e_n)$

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),
$(f, Boolean \rightarrow Int) \}$

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

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

Exercise:

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

Overloading of Operators

Int x Int $\rightarrow$ Int

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

Not a problem for type checking from leaves to root

String x String $\rightarrow$ String

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

Arrays

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

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

Assigning to an array

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

Example with Arrays

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

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

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

Type Rules (1)

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

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

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

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

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

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

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

Type Rules (2)

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

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

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

block

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

array use

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

array assignment

Type Rules (3)

Γ^c - top-level environment of class C

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

↓

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

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

method invocation

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

field use

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

field assignment

Does this program type check?

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

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

Type check: area

Type check: resize

Semantics of Types

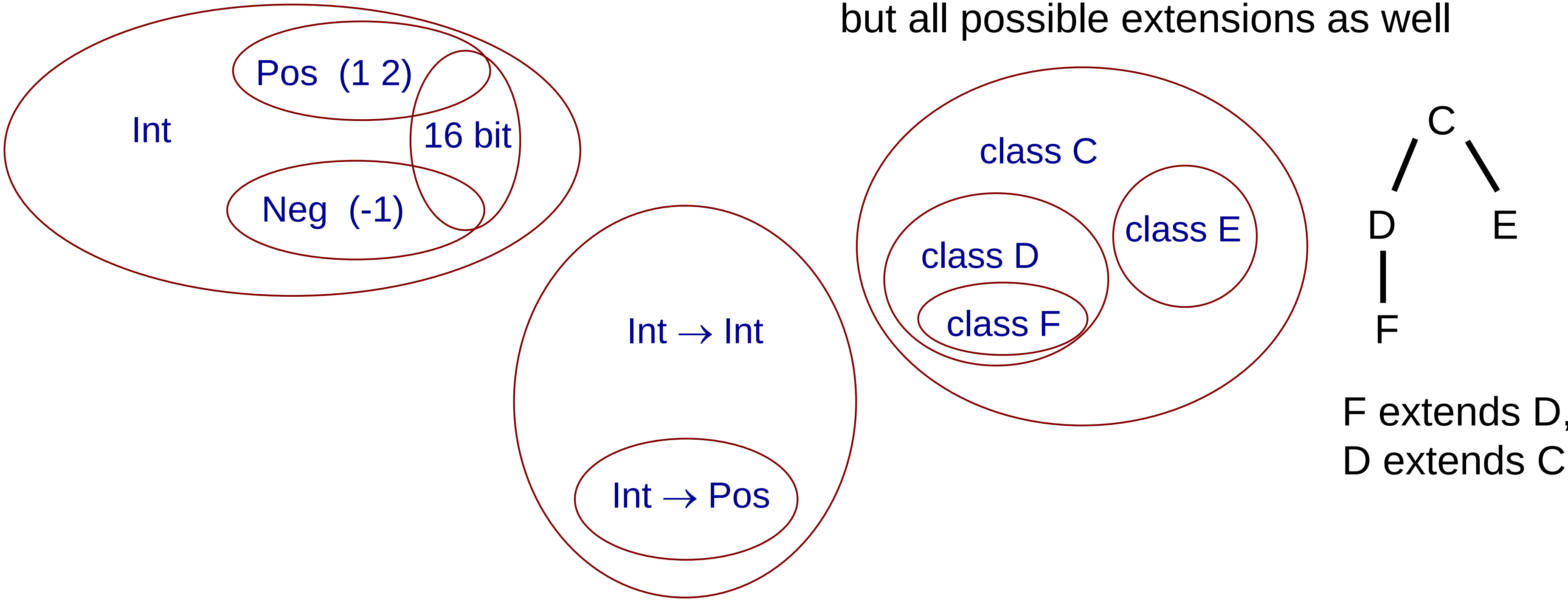
- Operational view: Types are named entities
  - such as the primitive types (Int, Bool etc.) and explicitly declared classes, traits …
  - their meaning is given by methods they have
  - constructs such as inheritance establish relationships between classes
- Mathematically, Types are sets of values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

Types as Sets

- Sets so far were disjoint

- Sets can overlap

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



SUBTYPING

Subtyping

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

In terms of sets

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

Types for Positive and Negative Ints

Int = { … , -2, -1, 0, 1, 2, … }

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

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

types:

Pos <: Int
Neg <: Int

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

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

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

sets:

Pos $\subseteq$ Int
Neg $\subseteq$ Int

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

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

(y not zero)

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

(x/y well defined)

Rules for Neg, Pos, Int

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

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

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

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

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

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

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

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

type system proves: no division by zero

Subtyping Example

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

← Does this statement type check?

Given:

$$Pos <: Int$$

$$\Gamma \vdash f: Int \rightarrow Pos$$

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

Subtyping Example

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

← Does this statement type check?

does not type check

What Pos/Neg Types Can Do

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

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

Subtyping and Product Types

Subtyping for Products

T_1 <: T_2 implies for all e:

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

Type for a tuple:

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

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

So, we might as well add:

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

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

Analogy with Cartesian Product

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

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

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

Subtyping and Function Types

Subtyping for Function Types

$T_1 <: T_2$ implies for all e:

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

contravariance covariance

$$\frac{T_1' <: T_1 \quad \dots \quad T_n' <: T_n \qquad T <: T'}{(T_1 \times \dots \times T_n \to T) <: (T_1' \times \dots \times T_n' \to T')}$$

Consequence:

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

as if $\Gamma \vdash m: T'_1 \times \dots \times T_n' \to T'$

Function Space as Set

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

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

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

We can prove

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

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
- For class sub-typing, we must require (at least) that methods named the same are subtypes

Example

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

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

Therefore, we need to have:

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

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