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

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

assignment

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

subtyping

does it type check?

def intSqrt(x:Pos) : Pos = { ...}

var p : Pos

var q : Neg

var r : Pos

q = -5

p = q

r = intSqrt(p)

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

Runtime error: intSqrt invoked with a negative argument!

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

What went wrong in *Tootool 0.1* ?

Pos <: Int

Neg <: Int

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

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

does it type check? – yes

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

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

Runtime error: intSqrt invoked with a negative argument!

x must be able to store any value from T

e can have any value from T

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

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

Recall Our Type Derivation

Pos <: Int

Neg <: Int

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

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

Runtime error: intSqrt invoked with a negative argument!

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

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

Corrected Type Rule for Assignment

Pos <: Int

Neg <: Int

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

assignment

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

subtyping

does it type check?

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

¡ = {(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

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

Γ stores declarations (promises)

Corrected Type Rule for Assignment

Pos <: Int

Neg <: Int

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

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

does it type check?

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

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

does not type check

Is there another way to fix the type system ?

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

Type System Soundness

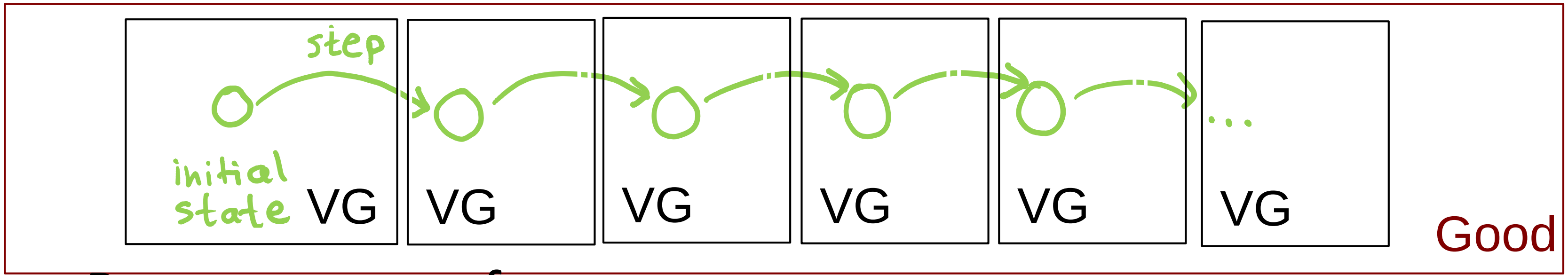
Proving Soundness of Type Systems

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

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

    e.g. 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

Operational semantics

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

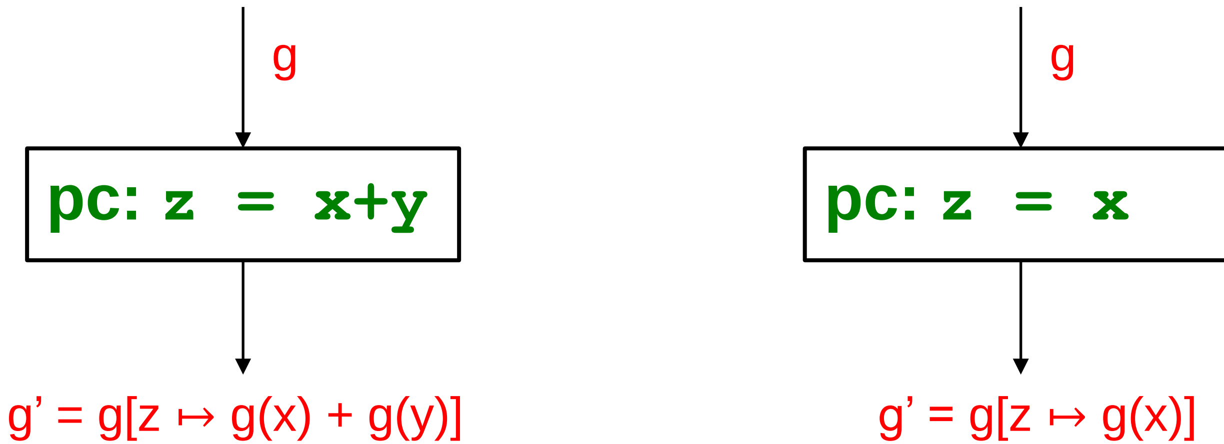
- Small-step Operational Semantics (SOS): consider individual steps (e.g. z = x + y)

V:	set of variables in the program
pc:	integer variable denoting the program counter
$g: V \to Int$	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.

- Big-step semantics: consider the effect of entire blocks

Operational semantics



Operation semantics

- If pc: z = x + y, (g, pc) $\rightarrow$ (g’, pc + 1), where g’ = g[z $\mapsto$ g(x)+g(y)]
- If pc: z = x, (g, pc) $\rightarrow$ (g’, pc + 1), where g’ = g[z $\mapsto$ g(x)]

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:
Γ = { (x_1, Pos),
(x_2, Int),
...
(x_n, Pos)}
Pos <: Int

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

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

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

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

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

Bad State: About to Divide by Zero (Crash)

var x : Pos

var y : Int

var z : Pos

x = 1

y = -1

z = x + y

x = x + z

y = x / z

z = z +

position in source

values of variables:

x = 1

y = -1

z = 0

Definition: state is *bad* if the next instruction is of the form

$x_i = x_j / x_k$ and x_k has value 0 in the current state.

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

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

position in source

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

Good

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

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

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

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

position in source

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

Good

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

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

Moved from Good to Bad in One Step!

Being good is not preserved by one step, not inductive!

It is very local property, does not take future into account.

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

position in source

values of variables:

x = 1

y = -1

z = 0

Bad

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

Definition: state is *bad* if the next instruction is of the form

$x_i = x_j / x_k$ and x_k has value 0 in the current state.

Being Very Good: A Stronger Inductive Property

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

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

This state is already not *very good.*

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).

Proving Soundness - Intuition

We want to show if a program *type checks:*

- It will be *very good* at the start
- if it is *very good* in the current step, it will remain *very good* in the next step
- If it is *very good*, it will not *crash*

Hence, please type check your program, and it will never crash!

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

Proving Soundness in Our Case

Holds: in initial state, variables are =1

- If a program *type checks* :
  - √ – It will be *very good* from at start.

    $1 \in \text{Pos}$
    $1 \in \text{Int}$
  - – if it is *very good* in the current step, it will remain *very good* in the next
  - √ – If it is *very good*, it 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:Pos, then z is strictly positive).

Proving that “very goodness” is preserved by state transition

- How do we prove
  - if you are *very good*, then you will remain *very good* in the next step
  - Irrespective of the actual program
- We could use SOS – small step operational semantics here.

Proving that “very goodness” is preserved by state transition

Hypothesize that g is very good

Prove that g’ is very good
When the program type checks

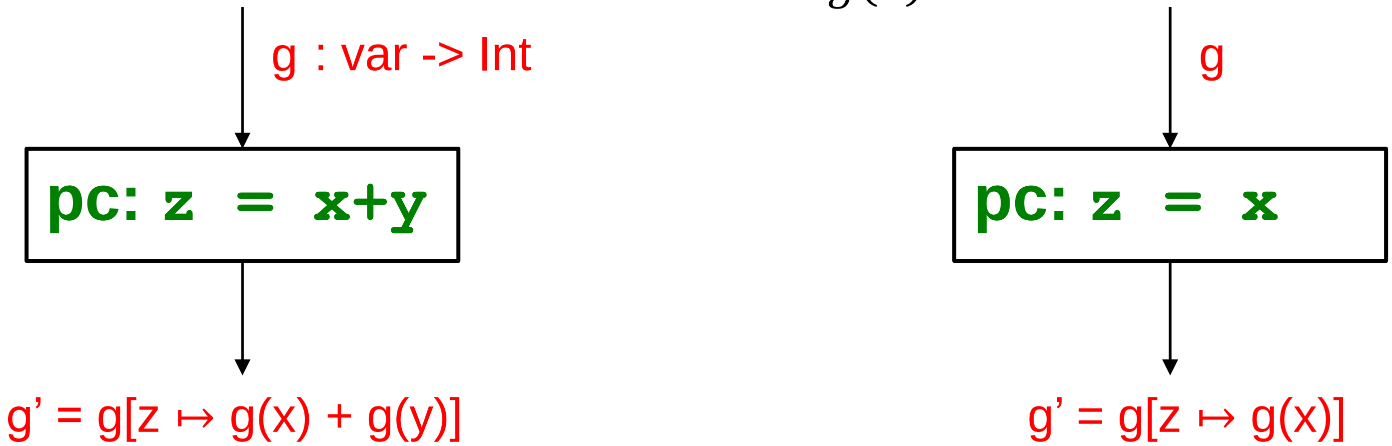
- Do this for every possible “step” of the operational semantics

Proving this for our little type system

Hypothesize that the following holds in g

For all vars x, x:Pos => x is strictly positive

$$\forall x. \Gamma \vdash x : Pos \Rightarrow g(x) > 0$$



Prove that the following holds in g'

For all vars x, x:Pos => x is strictly positive

$$\forall x. \Gamma \vdash x : Pos \Rightarrow g'(x) > 0$$

- Can we prove this ?
  - Only if we are given that the program type checks

Recall the Type Rules

Pos <: Int

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

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

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

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

$$\frac{}{\text{k: Pos}}$$

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

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

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

Back to the start

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

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

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

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

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

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

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

Does the proof still work?

If not, where does it break?

Let's type check some programs
Example 1

```
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: provide a type derivation for the program

Example 2

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 the program type checks
impossible, because z=x+y would not type check

How do we know it could not type check?

Must Carefully Check Our Type Rules

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

Conclude that the only types we can derive are:
x : Pos, x : Int
y : Int
x + y : Int

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

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

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

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

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

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

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

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

Remark

- We used in examples   Pos <: Int

- Same examples work if we have

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

and is therefore relevant for OO languages

What if we want more complex types?

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

- Should it type check?
- Does this type check in Java?
  - can you run it?
- Does this type check in Scala?

What if we want more complex types?

Suppose we add to our language a reference type:

class Ref[T](var content : T)

Programs:

var x_1 : Pos
var x_2 : Int
var x_3 : Ref[Int]
var x_4 : 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?

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

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
```

$\underbrace{\text{var x, var y, var z}}_{\Gamma}$

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

type checks

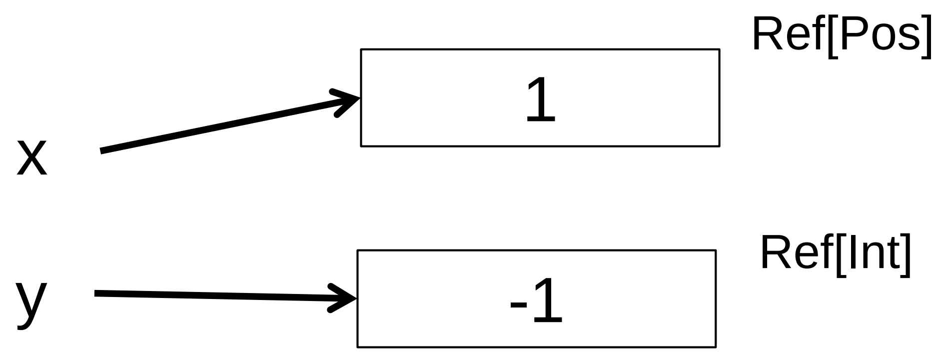
Simple Parametric Class

class Ref[T](var content : T)

Can we use the subtyping rule

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

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



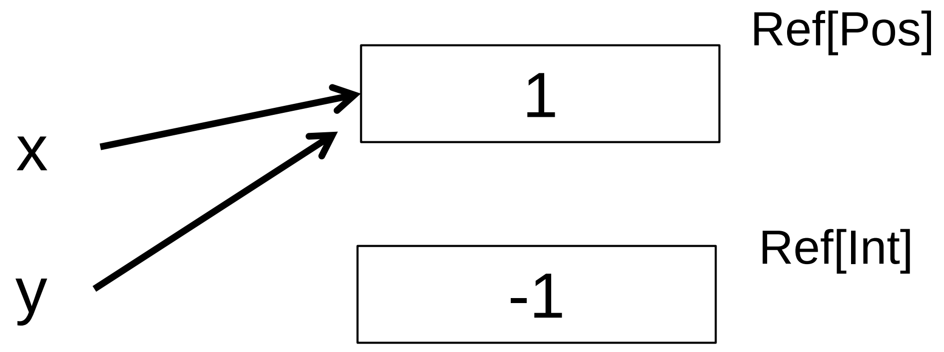
Simple Parametric Class

class Ref[T](var content : T)

Can we use the subtyping rule

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

```
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
```



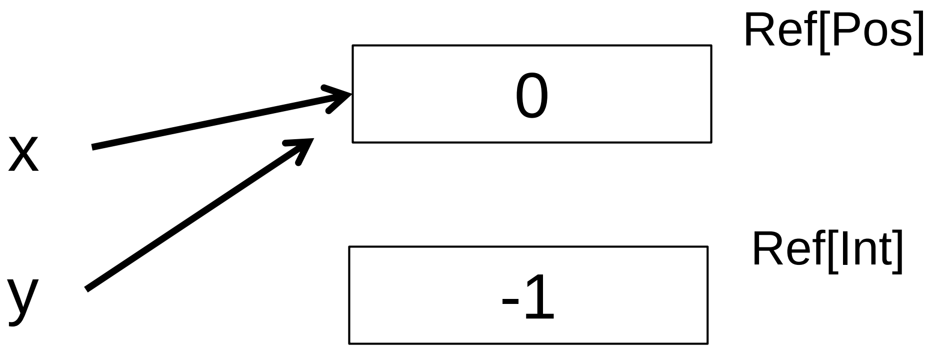
Simple Parametric Class

class Ref[T](var content : T)

Can we use the subtyping rule

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

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



CRASHES

Analogously

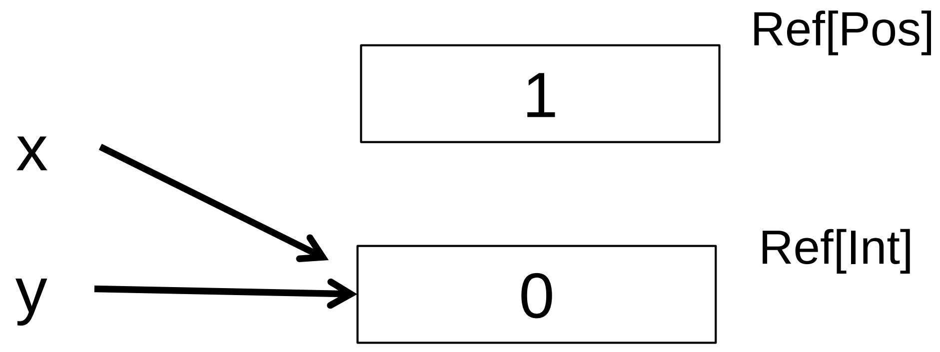
class Ref[T](var content : T)

Can we use the converse subtyping rule

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

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

CRASHES



Mutable Classes do not Preserve Subtyping

class Ref[T](var content : T)

Even if T <: T’,

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

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

Same Holds for Arrays, Vectors, all mutable containers

Even if T <: T’,

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

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

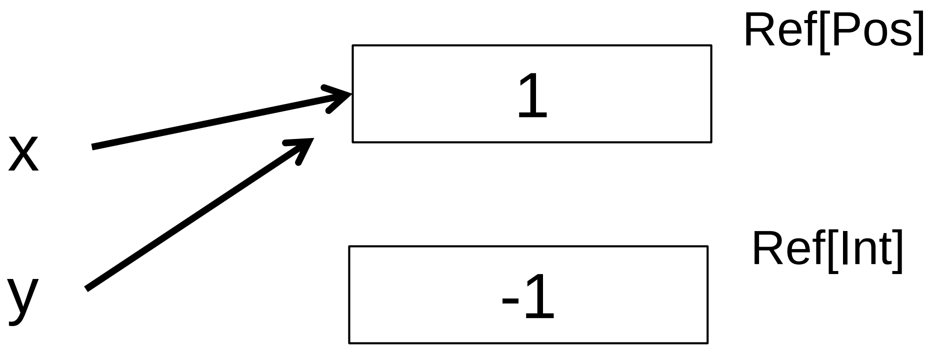
Case in Soundness Proof Attempt

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

Can we use the subtyping rule

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

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

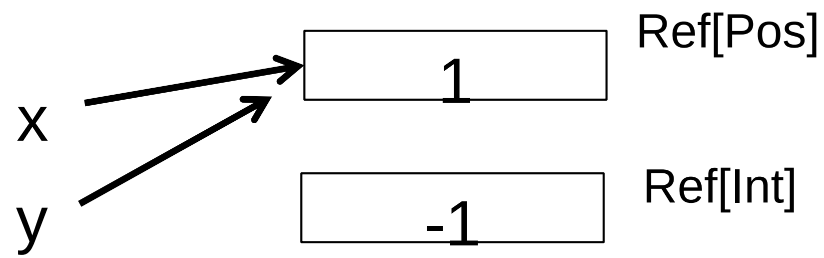


prove that runtime value of each variable belongs to its type.

Soundness Proof Attempt [Cont.]

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

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



prove that runtime value of each variable belongs to its type.

- Need to have an operational semantics for the language
- State g : (Var ∪ Addr) -> (Int ∪ Addr)
- A very good property that we need :
  - $\forall x. \Gamma \vdash \mathrm{x} : \mathrm{Ref[Pos]} \Rightarrow g(g(x)) > 0$
  - Cannot prove this property is preserved because $g(x) = g(y)$ and "y.content = 0" may update $g(g(x))$ (given by operational semantics)
- Proof will not work for any stronger properties also

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. That is coll[T] is *invariant* in T

A cool exercise – Physical Units as Types

- Define a “unit type” by the following grammar
- $u \rightarrow b \mid u^{-1} \mid u \; * u$
- $b \rightarrow \;\; kg \mid m \mid s \mid A \mid K \mid mole \mid cd$
- We use the syntactic sugar
  - u^n to denote u multiplied with u n-times
  - $\frac{u_1}{u_2}$ to denote $u_1 * u_2^{-1}$
- Give the type rules for the arithmetic operations $+, *, /,$ $sqrt$, $\sin$, abs.
- Trigonometric functions take argument without units
- An expression has no units if $\Gamma \vdash \; e\colon \; 1$

Physical Units as Types
Part 2

- The unit expressions are strings, so
- $\frac{s^2m^2}{m^2s}$ and s will not be considered as same types though they have same units
- How can we modify the type rules so that they type check expressions, whenever their units match as per physics?

Physical Units as Types
Part 3

Determine the type of T in the following code fragment.

- val x: <m> = 800
- val y: <m> = 6378
- val g: <m/(s*s)> = 9.8
- val R = x + y
- val w = sqrt(g/R)
- val T = (2 * Pi) / w

Physical Units as Types
Part 4

Suppose you want to use the unit *feet* in addition to the SI units. How can you extend your type system to accommodate for this?

(Assume that 1m = 3.28084 feet.)