Type soundness

In a more formal way

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

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)

Soundness here: no division by zero

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

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

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

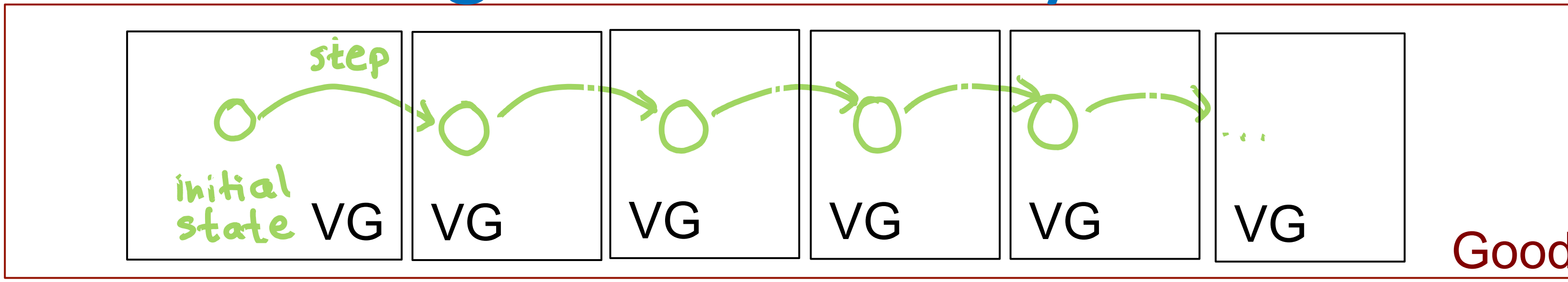
$$\frac{(x,T) \in \Gamma}{\Gamma \vdash x : T}$$

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

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

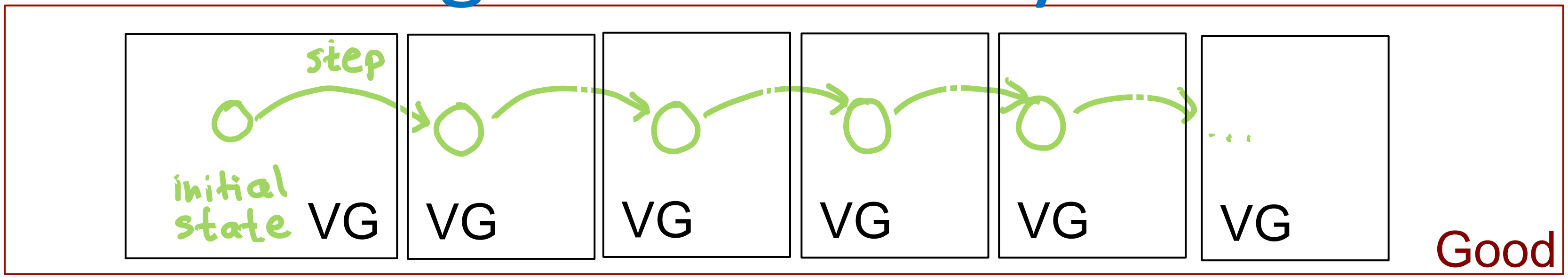
$$\frac{e_1 : Pos \qquad e_2 : Pos}{e_1 + e_2 : Pos}$$

Proving Soundness by Induction



- Program moves from state to state
- Bad state = state where program is about to exhibit a bad operation (3 / 0)
- 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)

Proving Soundness by Induction



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)

Given program statements and type rules, and under the assumption that programs type check

1. Define a formal description of program execution (operational semantics)
2. Find an invariant to describe very good states
3. Prove that the invariant is preserved for each execution step
4. Prove that the invariant implies no division by zero

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
- <u>small-step semantics</u>: consider individual steps (e.g. z = x + y)

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

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

Step 1: operational semantics

Give the operational semantics for our simple language.

Programs:

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

variable declarations
var x: Pos (assume default value 1)
or
var x: Int (assume default value 0)

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)

Notation:

$g(x := e)$ function update

$g(x)$ value of variable x

Step 2: invariant

"A state is very good, if each variable belongs to the domain determined by its type."

Find the invariant that formalizes this.

Step 3: invariant is inductive

Show that if a program type checks,

- invariant holds in program's initial state
- if the invariant holds in one state, it holds in the next state

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

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

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

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

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

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

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

Step 4: invariant implies no crash

Show that assuming a program type checks, its execution will not divide by zero.

Back to the start

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

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

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

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

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

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

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

Does the proof still work?

If not, where does it break?

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?
- 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?

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