Static types for Information Flow

We care no more interested in integrity, but confidentiality.

Why Care About Information Flow?

- Sensitive data (e.g., passwords, medical info) must not leak.
- Bugs or malicious code may accidentally (or intentionally) bypass protections.
- Classic type systems catch errors, what if they could catch leaks?

Access control: you associate several rights, but these are not enough to manage confidentiality issues.
You have a problem between secret and public data (1)

Access control is not enough

Control not only accesses to resources (data at rest)
but also information flow between resources. (data in transit)

(1) Confidentiality policy: information can flow only from less secret to more secret.

$$\text{public} \to \text{public} \quad \text{public} \to \text{secret}$$
$$\text{secret} \to \text{secret} \quad \text{secret} \not\to \text{public}$$

Integrity policy: information can flow only from more reliable to less reliable.

$$\text{reliable} \to \text{reliable} \quad \text{reliable} \to \text{dubious}$$
$$\text{dubious} \to \text{dubious} \quad \text{dubious} \not\to \text{reliable}$$

What is information flow

Tracking How Data Moves

- **Explicit flow:** data directly copied from secret to public variables.
- **Implicit flow:** control flow reveals information

```
let secret = true;
let mut public = false;
if secret {
    public = true;
}
```

mutable

Condition

To clarify consider this simple program:

Overall result of this program: attacker can control only public data. Attacker can check public data. Attacker can grasp that secret is true because public goes from false to true.

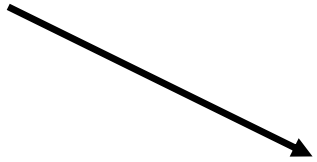
(and observe)

What is information flow

Tracking How Data Moves

- **Explicit flow:** data directly copied from secret to public variables.
- **Implicit flow:** control flow reveals information

```
let secret = true;
let mut public = false;
if secret {
    public = true;
}
```



public leaks info about secret.

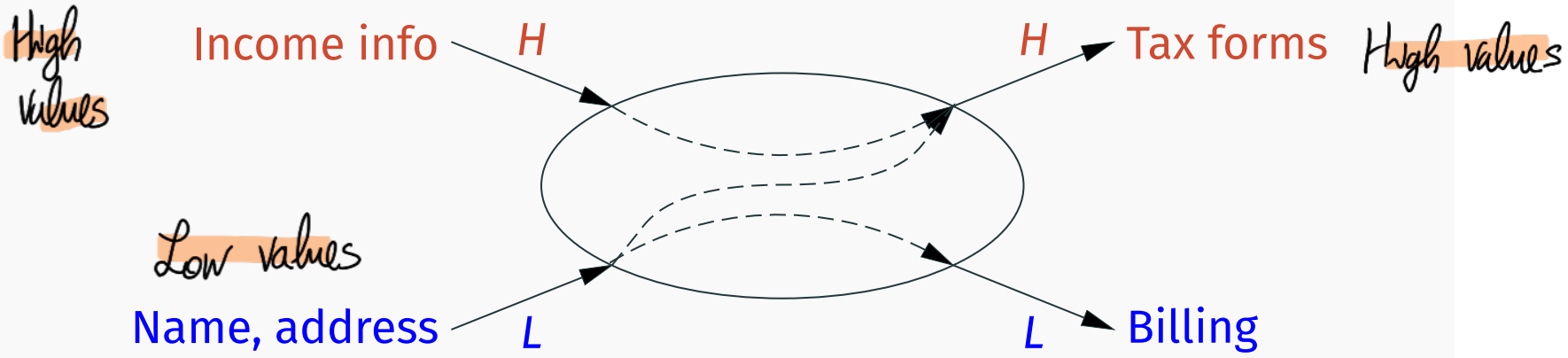
This issue very well known because it is called implicit flow: a control statement reveals information. It is different from explicit flow. Leakage is related to IF.

Information flow policies

- Information flow policies specify the security of the system by stating which flows are allowed.
- Intuition: Suppose **L** (**low**) and **H** (**high**) were the only users in our system. The policy
  - ***no actions issued by H should have any effect visible to L***.

Example:

Associate confidentiality levels to inputs, outputs, and intermediate results of the program.



Check that information flows always "go up":
an output with level ℓ depends only on inputs with levels $\ell' \sqsubseteq \ell$.

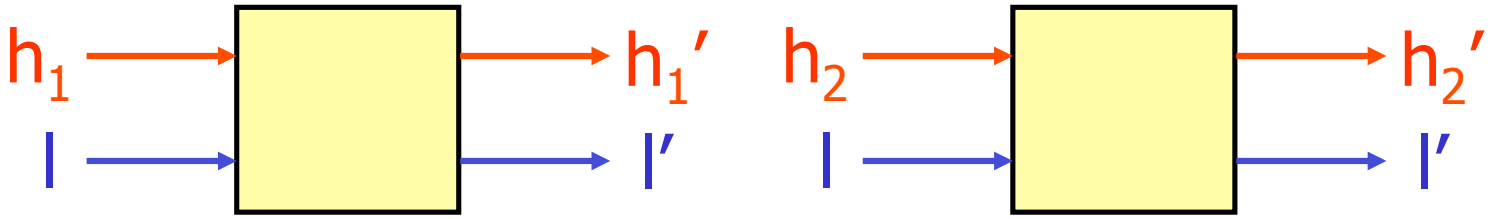
Legal flow: You don't want flow of information that goes from high to low.

What Are We Trying to Guarantee?

- **Noninterference:** changes in secret inputs should not affect public outputs.
- Goal: attacker learns *nothing* from observable public behavior.

system represented with 2 inputs (public and secret) and you want to avoid plugging different high inputs and get a different result (low output)

- Noninterference [Goguen & Meseguer]: as high input varied, low-level outputs unchanged



- How do we formalize noninterference in terms of program semantics?

nontermintation

$$[\![C]\!] : \text{Int} \times \text{Int} \to (\text{Int} \times \text{Int})_\perp$$

high input | low input | high output | low output

Memory equivalence:
two pairs (h, l) are equivalent if and only if they are equal at the low value.

Semantics-based security

- Semantics-based security for C: as high input varied, low-level behavior unchanged:

C is secure iff

We start from two memories that are low level equivalent

We want that the result of C is indistinguishable

$$\forall m_1, m_2.\ m_1 =_L m_2 \Rightarrow [\![C]\!]m_1 \approx_L [\![C]\!]m_2$$

low memory equivalent

Low-memory equality: $(h,l) =_L (h',l')$ iff $l=l'$

C's behavior: semantics $[\![C]\!]$

Low view $\approx_L$: indistinguishability by attacker

Not a surprise, attacker can only see low values. Attacker cannot notice modification of high level values by looking at high level values

m: state of the memory seen by the attacker.

memory state

Non Interference: property that cannot be verified just by looking at one execution

- Non interference is not a property of a single run

- Non Interference relates **multiple executions** of a program with **different secret inputs**, checking whether they produce the **same public outputs**.
- Noninterference is a **hyper-prooperty property**:

```
fn simple_fun(secret: int) -> int {
    let mut public = 0;
    if secret > 0 {
        public = 1;
    }
    return public;
}
```

An attacker won't know secret value, but will know if secret > 0.

secret is **high-security input** (private).
public is **low-security output** (observable).

Run simple_fun once!

```
fn simple_fun(secret: int) -> int {
    let mut public = 0;
    if secret > 0 {
        public = 1;
    }
    return public;
}
```

simple_fun(5) // returns 1

As an external observer:

You only see 1.

You can't tell whether the function always returns 1, or if it depends on secret.

Overall

From one run, you can't tell if it's leaking private info.

Run simple_fun twice!

```
fn simple_fun(secret: int) -> int {
    let mut public = 0;
    if secret > 0 {
        public = 1;
    }
    return public;
}
```

simple_fun(5) // returns 1
simple_fun(-2) // returns 0

As an external observer you can infer

Output 1 corresponds to secret > 0

Output 0 corresponds to secret ≤ 0

Overall

The attacker now learns something about the secret just by watching the output.

Run simple_fun twice!

```
fn simple_fun(secret: int) -> int {
    let mut public = 0;
    if secret > 0 {
        public = 1;
    }
    return public;
}
```

simple_fun(5) // returns 1
simple_fun(-2) // returns 0

As an external observer you can infer

Output 1 corresponds to secret > 0

Output 0 corresponds to secret ≤ 0

This is a leak!!
You only know that by comparing runs!

Overall

The attacker now learns something about the secret just by watching the output.

Key insight

A single execution shows you what happened. Instead, multiple executions show you *what depends on what*.

Noninterference is not a property of a single run — it's a property of how a program behaves across **varying secret inputs** while keeping the public context fixed.

Hyper-properties

A **hyperproperty** is a property of **sets** of program executions [**Clarkson & Schneider (2008)**].

↳ Security properties are a subset of Safety properties

- **Safety**: "Something bad never happens" (property of individual traces).
- **Liveness**: "Something good eventually happens" (also individual traces).
- **Hyperproperties**: Describe **relationships** between **multiple executions**.

↳ ex: threads will access resource eventually (if it hasn't now, doesn't mean won't happen in future)

Non Interference again

Another way of formulating it

- Exec(P) = set of all possible executions of program P.
- Each execution t ∈ Exec(P) is a trace (sequence of states/outputs).

Noninterference holds iff:

For all t1, =t2 ∈ Exec(P):

- If t1 and t2 agree on public inputs, then t1 and t2 also agree on public outputs.
- NI is a predicate on **sets of traces**, this is what makes it a **hyperproperty**.

Why This Matters

- One **can't verify** noninterference by looking at just one execution.
- It is required to **analyze and relate** multiple traces.
- This affects:
  - How we **formally verify** systems.
  - How we **design type systems**
  - How we **test for leaks**

Our interest is on static analysis: we would like to establish a type system that ensures NI if type checks are okay.


Confidentiality

Managing confidentiality

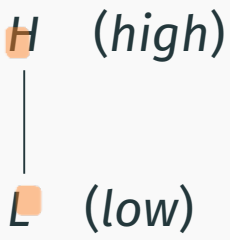
A partial order over confidentiality levels

$$A \sqsubseteq B$$

"B is more secret or as secret as A."

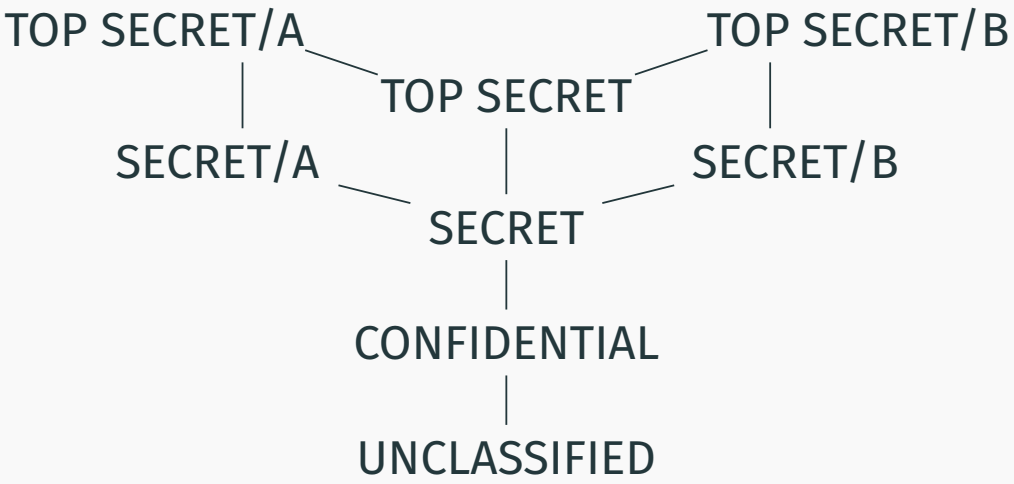
"Someone with credentials B can access information classified A."

Example: the public/secret classification.



Managing confidentiality

Example: the US government classification.



Using Types to Enforce Security

- Types for information flow: confidentiality
- We consider a lattice of different security levels
- For simplicity, just two levels But techniques can be extended
  - **H(igh) or confidential, secret**
  - **L(ow) or public**
  - **Assume that L < H**
- Simple security structure (easy to generalize to arbitrary lattices):
  - **Intended security: low-level observations reveal nothing about high-level input**

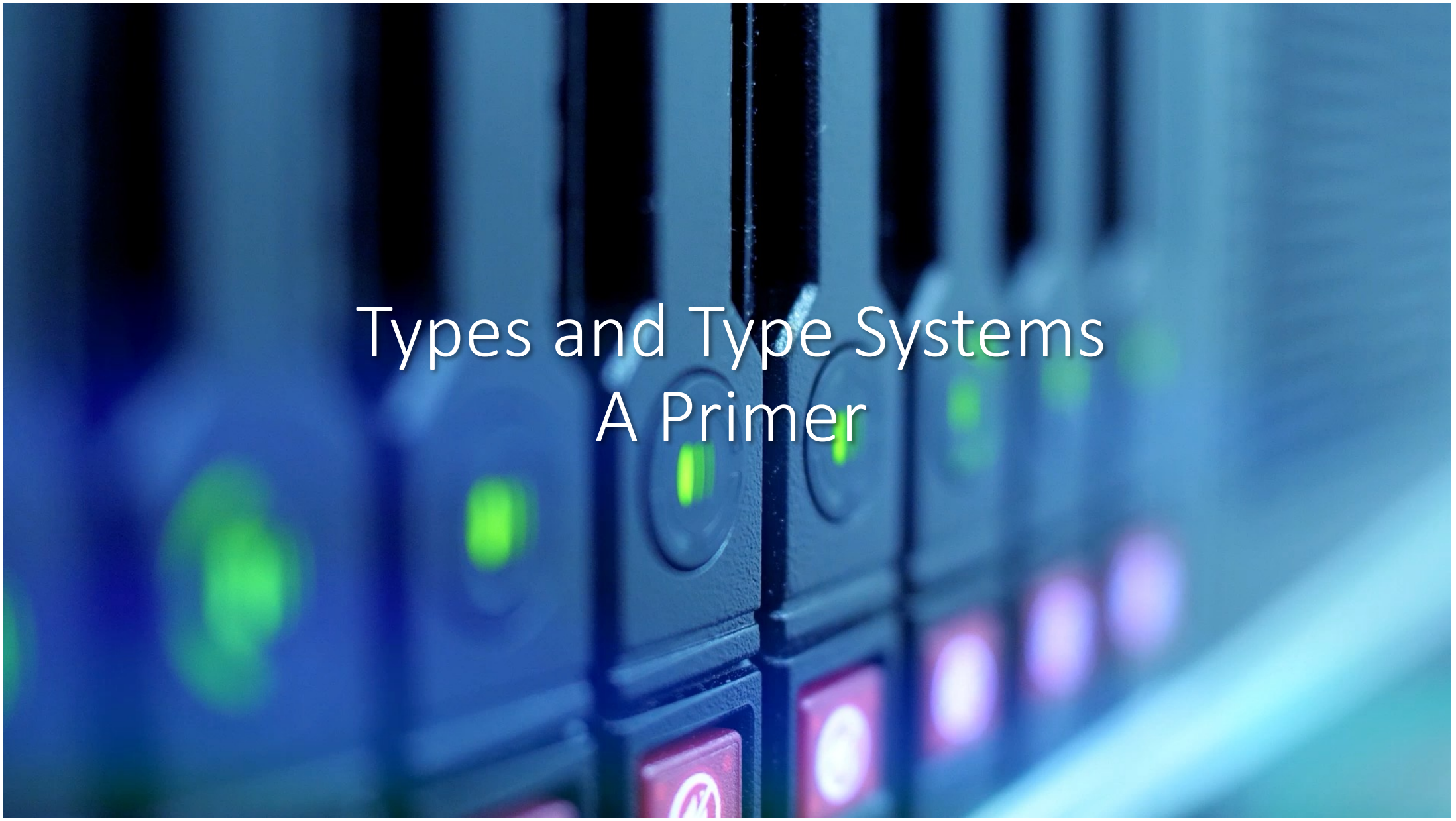
Static certification

- Security enforcement by the usage of type systems

Only run programs which can be statically **verified** as secure **before** running them

Static certification for inclusion in the language tool-chain

Enforcement by **security-type systems**


Types and Type Systems
A Primer

Type: classification of a property of an entity in a programming language. Types tell a sort of over approx. of values a variable can have at runtime.

A primer on types in programming languages

What is a type?

Types classify program phrases according to the kinds of values they compute

They are predictions about values

Static approximation of runtime behavior – conservative

When someone says, **"Types classify program phrases according to the kinds of values they compute,"** they're basically saying that **types are like labels** we attach to pieces of code to indicate **what kind of result** that code gives us when it runs.

A "program phrase" just means any little chunk of code — like an expression, a function, a variable, or even a full block. And "the kinds of values they compute" means what comes out of them: maybe it's a number, a string, a boolean, a list of cats, whatever.

Why Types?

Static analysis: detect (potential) runtime errors before code is run:

```
# 1 + true;;
Error: This expression has type bool but an expression was expected of
type int
#
```

Why types?

Enforcing access policy (private/protected/public methods)

Guiding implementation (type-based programming)

Documentation: types tell us a lot about functions and provide a documentation that is repeatedly checked by the compiler (unlike comments)

Help compilers choose (more/most) efficient runtime value representations and operations

Maintenance: if we change a function's type, the type checker will direct us to all use sites that need adjusting

Typed-based programming

```
# let rec map f = function
    | [] -> []
    | h :: t -> f h :: map f t;;
    val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let add1 = map (fun x -> x + 1);;
val add1 : int list -> int list = <fun>

# let concat_bang = map (fun x -> x ^ "!");;
val concat_bang : string list -> string list = <fun>
```

map takes a function that takes a generic value of type a

?

A map function takes another function as a parameter and applies the function to all the elements of a list.

Type based programming (cont.)

Certain programs won't be accepted because of types

takes two strings and return string

RUST: Lifetime

```
// this code sample does *not* compile
fn f(s: &str, t: &str) -> &str {
            if s.len() > 5 { s } else { t } }
```

- This won't compile because of the concept of lifetime of a variable.

```
fn f<'a>(s: &'a str, t: &'a str) -> &'a str {
            if s.len() > 5 { s } else { t }
}
```

What is a type system?

- **A tractable syntactic method for proving the absence of certain program behavior** (quoting Benjamin Pierce book on Types and Programming Languages)
- A branch of mathematics/logic: type theory
  - Original motivation: avoiding Russell's paradox
- Type systems are generally conservative:
  - **1 + (if true then 10 else "hello")**
    would behave OK at runtime, but is, nevertheless, typically rejected statically → because if the else branch is evaluated we would have a string: we expect number but we can get two types.
- Type checker & type inference (e.g., OCAML)

> "A tractable syntactic method for proving the absence of certain program behavior."

Let's translate that in a more casual way:

- **"Tractable"** just means "we can actually do it with reasonable effort" — it's practical, not some crazy complex system.
- **"Syntactic method"** means the type system uses just the *structure* (the syntax) of your code, not how it actually runs, to reason about it. So it's analyzing *what you wrote*, not what happens at runtime.
- **"Proving the absence of certain program behavior"** — this is key. The type system is there to **prevent** certain errors, like trying to add a number to a string, or calling a function that doesn't exist, *before* you even run your code.

Type systems

Dynamic types are checked at runtime, typically when operations are performed on values

Static types are checked before program is (compiled and) run

Dynamic type checking

Some type checks are done dynamically

Consider Python

```
>>> 1 + "hello"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Type systems are constructed at AST level, so before static analysis

Specifying type systems

A type system is typically specified as a set of rules which allow assigning types to abstract syntax trees – similarly to how an interpreter evaluates abstract syntax trees to yield values

The goal: determine what type an expression (program phrase) has – if it has one

Type system =

Abstract syntax trees + Types + Typing rules

Notion of

So the syntax info

for constructs

Typing rules

Typing rules tell us how to determine the type for a particular expression-statement

Typing rules are syntactic methods – type checking and typing rules are **syntax-directed**

for each statement we define rules that tell whether or not statement is well typed

"Type checking and typing rules are syntax-directed."

Let's start with the parts:

- **Type checking** is when the language checks that your program follows the type rules — like, "you're using this variable as an integer, but is it actually an integer?"
- **Typing rules** are the rules that define how types behave in the language. For example: "If `x` is an `int` and `y` is an `int`, then `x + y` is also an `int`."
- **Syntax-directed** means the rules and checks are based on the *structure* of the code — its syntax.

So basically, it means that **the structure of your code directly determines how the type system checks it**.

Typing rules

On paper, typing rules are usually expressed as inference rules:

$$\frac{\text{1st premise} \quad \text{2nd premise} \quad \text{3rd premise} \ \ldots}{\text{conclusion}}$$

Similar to op semantics

Such a rule can be read as "If 1st premise AND 2nd premise AND 3rd premise . . . are all true, then the conclusion is also true"

If we can show that the premises hold, the rule allows us to conclude that is below the line

If a rule has no premises, it is an axiom

Typing rules: examples

$$\frac{n \text{ is an integer value}}{\vdash n : \text{Integer}}$$

"If n is an integer value, then the type of n is Integer"

$$\frac{\vdash e_1 : \text{Integer} \quad \vdash e_2 : \text{Integer}}{\vdash e_1 + e_2 : \text{Integer}}$$

"If the type of e_1 is Integer and the type of e_2 is Integer, then the type of expression $e_1 + e_2$ is also Integer"

Typing judgments

Typing rules are **syntactic methods** establishing certain properties of programs,

In the type literature they are also called **typing *judgements*** since we judge the expression to have our desired type (e.g. property), assuming certain preconditions.

How do we apply typing rules?

We build derivations!

But what are derivations?

A derivation is a (proof) tree built by *consistently* replacing variables in inference rules by concrete terms

At the bottom of the tree is the typing judgment we are trying to show

Example

$$\dfrac{\dfrac{\vdash \text{Bool True} : \text{TyBool}}{\vdash \text{Not (Bool True)} : \text{TyBool}} \quad \dfrac{\vdash \text{Bool False} : \text{TyBool} \quad \vdash \text{Bool True} : \text{TyBool}}{\vdash \text{And (Bool False) (Bool True)} : \text{TyBool}}}{\vdash \text{And (Not (Bool True)) (And (Bool False) (Bool True))} : \text{TyBool}}$$

Example

$$\dfrac{\dfrac{\dfrac{\vdash 3:\ int \qquad \vdash 4: int}{\vdash (3 < 4\ ): bool}}{\vdash\ !(3 < 4): bool} \qquad \vdash 3: int \qquad \vdash 5: int}{\vdash\ if\ !(3 < 4) then\ 3\ else\ 5: int}$$

Example

$$\frac{\vdash 3 : int \qquad \vdash true : bool}{\vdash 3 + true : ??}$$

How might we handle variables?

We don't seem to have enough information in our typing judgement to keep track of what variables are in scope, and of what types.

We need to generalize our judgement

$$\Gamma \vdash e : \tau$$

which is read, "In context Γ, expression e has type τ."

A **typing context** is simply a list of bindings of variable names to types.

We need a static environment to keep track of the types

Static environment mapping variables into types

(vs Dynamic Env: var → values)

This at compiler level is usually called symbol table

Example

We lookup to see if var has been bound to a type

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

The rule confirms that a variable has a given type if the context maps that variable to that type.

e must be of declared type

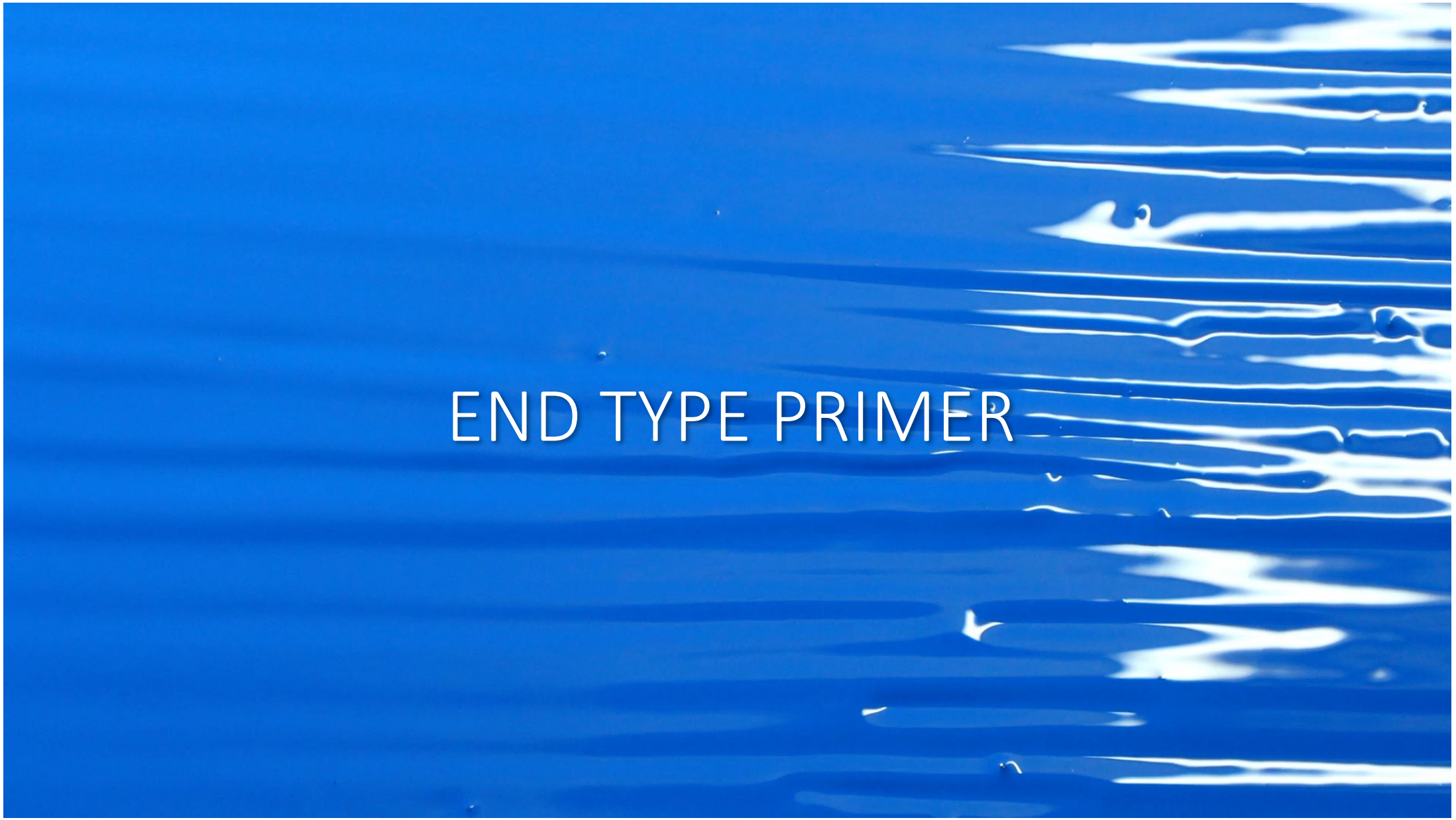
We need to evaluate the body and extend environment gamma with variable X.

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma, x : \tau_1 \vdash b : \tau_2}{\Gamma \vdash (let\ x : \tau_1 = e\ in\ b) : \tau_2}$$

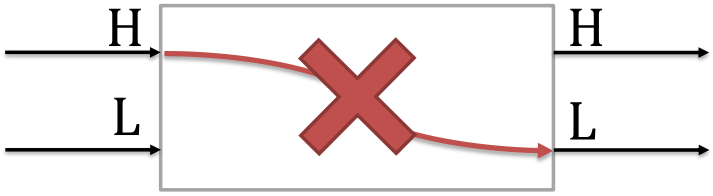
The rule typechecks the let-binding itself in the original context Γ. The body typechecks with the desired type in a context that has been enhanced with the type binding for x.

of type T1 of numerical value e in body b

if result is T2, then same for let.


END TYPE PRIMER

The methodology in a nutshell



- We are given
  - the lattice $L \sqsubseteq H$ of security levels (labels),
  - a program P
  - A typing environment Γ that maps variables to labels.
- Attacker knows L inputs and can observe L outputs.
- Static Program certification: non interference

Static Type system: design choices

- The type environment Γ is fixed
- Security levels (labels) as types
  - Security level Γ(**x**) is the type of **x**.

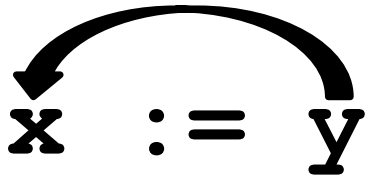
A simple imperative language

```
e ::= x | n | e1+e2 | ...

c ::= x := e
    | if e then c1 else c2
    | while e do c
    | c1; c2
```

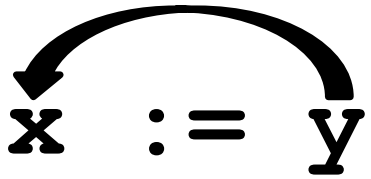
Certifying assignments

Assignments cause **explicit** flows of values.



Certifying assignments

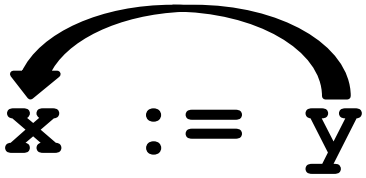
Assignments cause **explicit** flows of values.



We already discussed the data flow of assignments when illustrating taint analysis, highlighting how information propagates through a system and identifying potential security vulnerabilities as data moves between different components.

Certifying assignments

Assignments cause **explicit** flows of values.



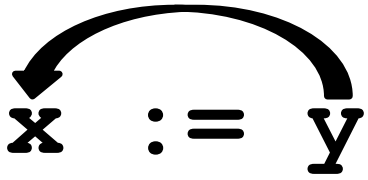
The basic question:

Which conditions need to be verified to ensure that the assignment x :=y satisfies non-interference, thus preserving confidentiality of data?

Define typing rules in a way that guarantees non-interference

Certifying assignments

Assignments cause **explicit** flows of values.



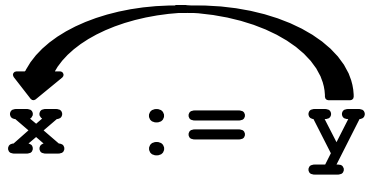
Data Sensitivity:
If y is bound to confidential information then allowing x to take on the same value as *y* might pose a risk to the confidentiality of the data.

Level of confidentiality of Y must be compatible with level of confidentiality of X.

Certifying assignments

Types = levels of confidentiality we are considering

Assignments cause **explicit** flows of values.



Level of confidentiality of variables we take from environment

The assignment satisfies Non Interference if the Security level of y is less that the security level of x

$$\Gamma(y) \sqsubseteq \Gamma(x).$$

Variable x has more secret than variable y

NI Certification Constraint:

If $\Gamma(\mathtt{y}) \sqsubseteq \Gamma(\mathtt{x})$, then **x:=y** satisfies NI.

If $\Gamma(\mathtt{y})$=L, $\Gamma(\mathtt{x})$= L, then **x:=y** satisfies NI 

If $\Gamma(\mathtt{y})$=L, $\Gamma(\mathtt{x})$= H, then **x:=y** satisfies NI 

If $\Gamma(\mathtt{y})$=H, $\Gamma(\mathtt{x})$= L, then **x:=y does not** satisfies NI 

What about ….

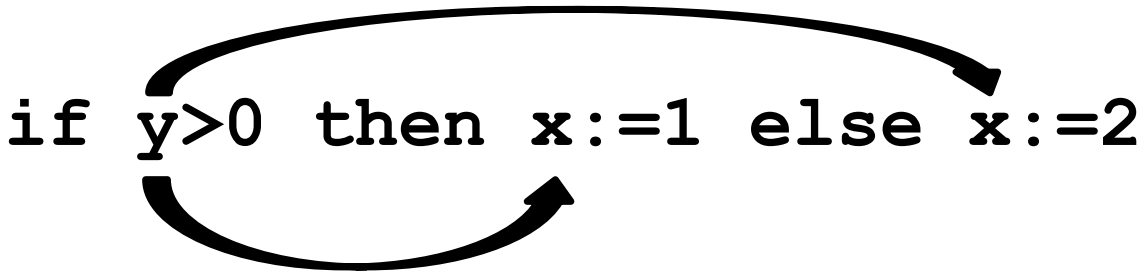
X := Y + Z

The assignment satisfies NI, if $\Gamma(\mathtt{y+z}) \sqsubseteq \Gamma(\mathtt{x})$.

The assignment satisfies NI, if $\Gamma(\mathtt{y}) \sqcup \Gamma(\mathtt{z}) \sqsubseteq \Gamma(\mathtt{x})$.

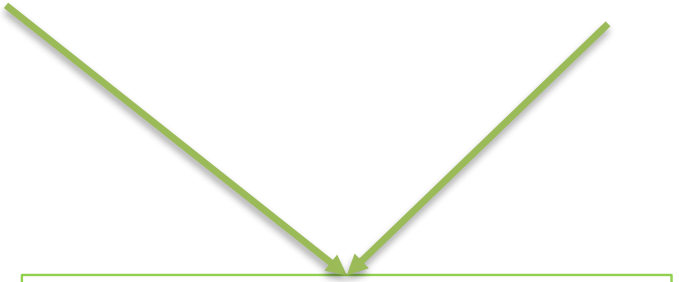
Transfer function: least upper bound!

Conditionals



Conditional commands (if-statements and while-statements) determines **implicit flows** of values.

```
if y>0 then x:=1 else x:=2
```



To manage type of conditionals we introduce sec context, (1)

Typing conditionals

if y>0 then x:=1 else x:=2

"manage type of conditionals"

#1 The security level of x is not sufficent to type IF-branches"

We need contxtual information to type IF-brances

(1) -> Type associated to the guard of a condition. So here associated to confidentiality

The security context (*cxt*) of both branches is that of Γ(y) level associated to y.

#2 Check if $ctx \sqsubseteq$ Γ(x) where ctx = Γ(y).

security level of y is compatible with security level of x.

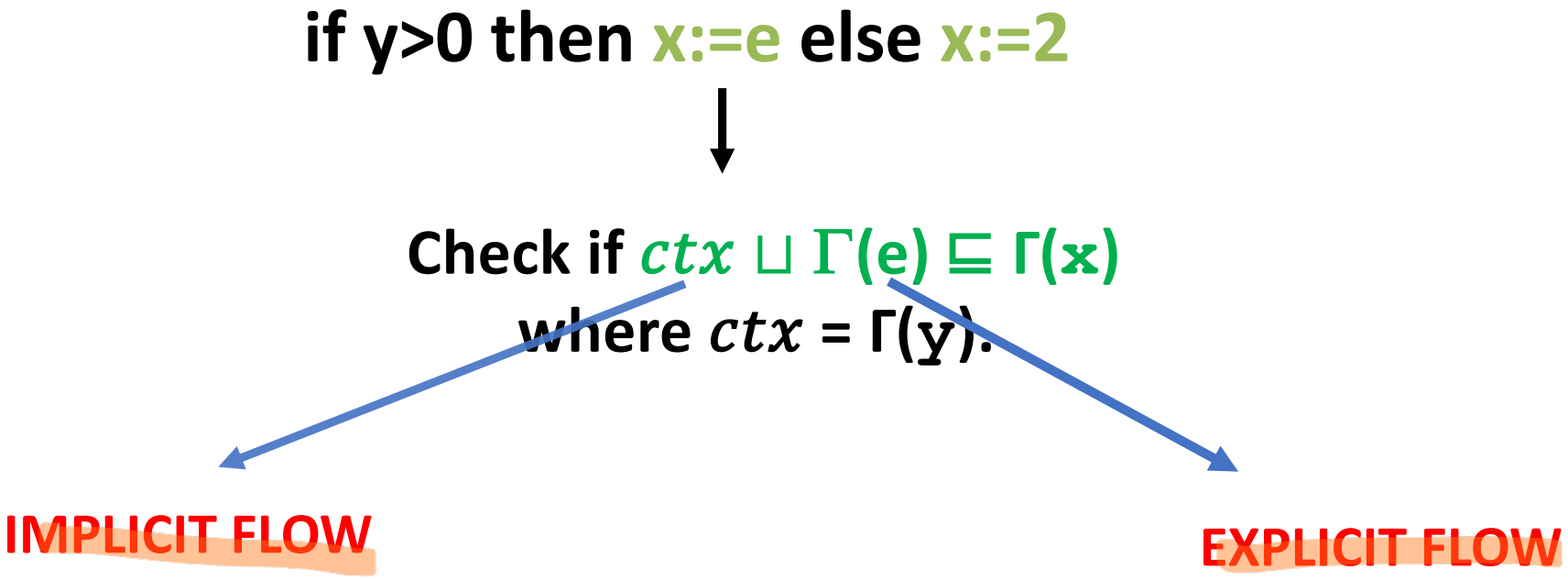
Typing conditionals (cont)

if y>0 then x:=e else x:=2

Check if $ctx \sqcup \Gamma(e) \sqsubseteq \Gamma(x)$
where ctx = Γ(y).

Direct and indirect flow together

Typing conditionals (cont)



```
if y>0 then x:=z+1 else x:=z+2
```

CHECK: $\Gamma(y) \sqcup \Gamma(z) \sqsubseteq \Gamma(x)$

$\Gamma(x) = L, \Gamma(y) = L, \Gamma(z) = L$

The IF stat. satisfies NI?

$\Gamma(x) = H, \Gamma(y) = L. \Gamma(z) = H$

The IF stat. satisfies NI?

```
if y>0 then x:=z+1 else x:=z+2
```

CHECK: $\Gamma(y) \sqcup \Gamma(z) \sqsubseteq \Gamma(x)$

$\Gamma(x) = L, \Gamma(y) = L. \Gamma(z) = L$

The IF stat. satisfies NI? 😀

$\Gamma(x) = H, \Gamma(y) = L. \Gamma(z) = H$

The IF stat. satisfies NI? 😀

```
if y>0 then x:=z+1 else x:=z+2
```

CHECK: $\Gamma(y) \sqcup \Gamma(z) \sqsubseteq \Gamma(x)$

$\Gamma(x) = L, \Gamma(y) = L. \Gamma(z) = H$

The IF stat. does satisfies NI 😱

```
if z>0 then
     if y>0 then x:=1 else x:=2
else
     x:=3
```

Check if $ctx \sqsubseteq \Gamma(\mathbf{x})$, where $ctx = \Gamma(\mathbf{z}) \sqcup \Gamma(\mathbf{y})$.

Check if $ctx \sqsubseteq \Gamma(\mathbf{x})$, where $ctx = \Gamma(\mathbf{z})$.

We need an outer context (outer if) and an inner one.

To get an intuition of the context

But … what is the context?

Imagine dynamic checks for confidentiality.
So you have values tagged with their security level

- Intuition: assume to perform dynamic verification of confidentiality policies instead of being static
- Replace each piece of run time data by a pair consisting of
  - **(value, (security) level)**
- The compiler instrument the code: the instrumentation checks the levels at each operation over the data.

Example

Manage analysis with instrumented code
to check compile-time label level

```
                                assert (x.level <= z.level);
z := (x + y) / 2        ⇒       assert (y.level <= z.level);
                                z.value := (x.value + y.value) / 2
```

The code **The instrumented code**

Example

This suffices to control explicit flows.

```
                              assert (x.level <= z.level);
z := (x + y) / 2      ⇒       assert (y.level <= z.level);
                              z.value := (x.value + y.value) / 2
```

The code **The instrumented code**

Implicit flow (again)

Consider two Boolean variables:

`x`, which is secret (level *H*) and `y`, which is public (level *L*).

```
if x then y := true else y := false
```

This code behaves like `y := x`.

Yet, only public values (`true` or `false`) are assigned to `y`.

How to dynamically instrument this program? We should also bring around with us: we need something that tells the security level of the PC.

PC security level is the context!

Implicit flow is considered through analyzing security level of PC

Great Idea

- We add a variable **pc** of type "level" to keep track of information revealed by conditional branches.
- The pc variable keeps track of the security level of the execution evaluating the guard of conditional
  - The pc variable represents the information that might be learned from the knowledge that the statement or expression was evaluated.
- **The pc variable is the context!!!**

The pc variable

This variable is updated at conditional statements and loops:

```
if x then ... else ...   ⇒   pc := max(pc, x.level);
                             if x.value then ... else ...
```

This variable is taken into account during assignments:

```
                         assert (x.level <= z.level);
z := (x + y) / 2   ⇒     assert (y.level <= z.level);
                         assert (pc <= z.level);
                         z.value := (x.value + y.value) / 2
```

Program variable
pc is at runtime,
it is an instrumented
variable, not viewable
by programmer

Instrumentation is much more complicated: security level of
pc should be comparable with z to avoid leakages.

Go back to our example

Consider two Boolean variables:
`x`, which is secret (level *H*) and `y`, which is public (level *L*).

```
if x then y := true else y := false
```

This code behaves like `y := x`.

Yet, only public values (`true` or `false`) are assigned to `y`.

The pc variable

This succeeds in controlling implicit information flows:

```
if x then y := true else y := false

   ⇒    pc := max(pc, x.level);   H
        if x.value
        then assert (pc <= y.level); y := true
        else assert (pc <= y.level); y := false
```

The pc variable

This succeeds in controlling implicit information flows:

```
if x then y := true else y := false
```

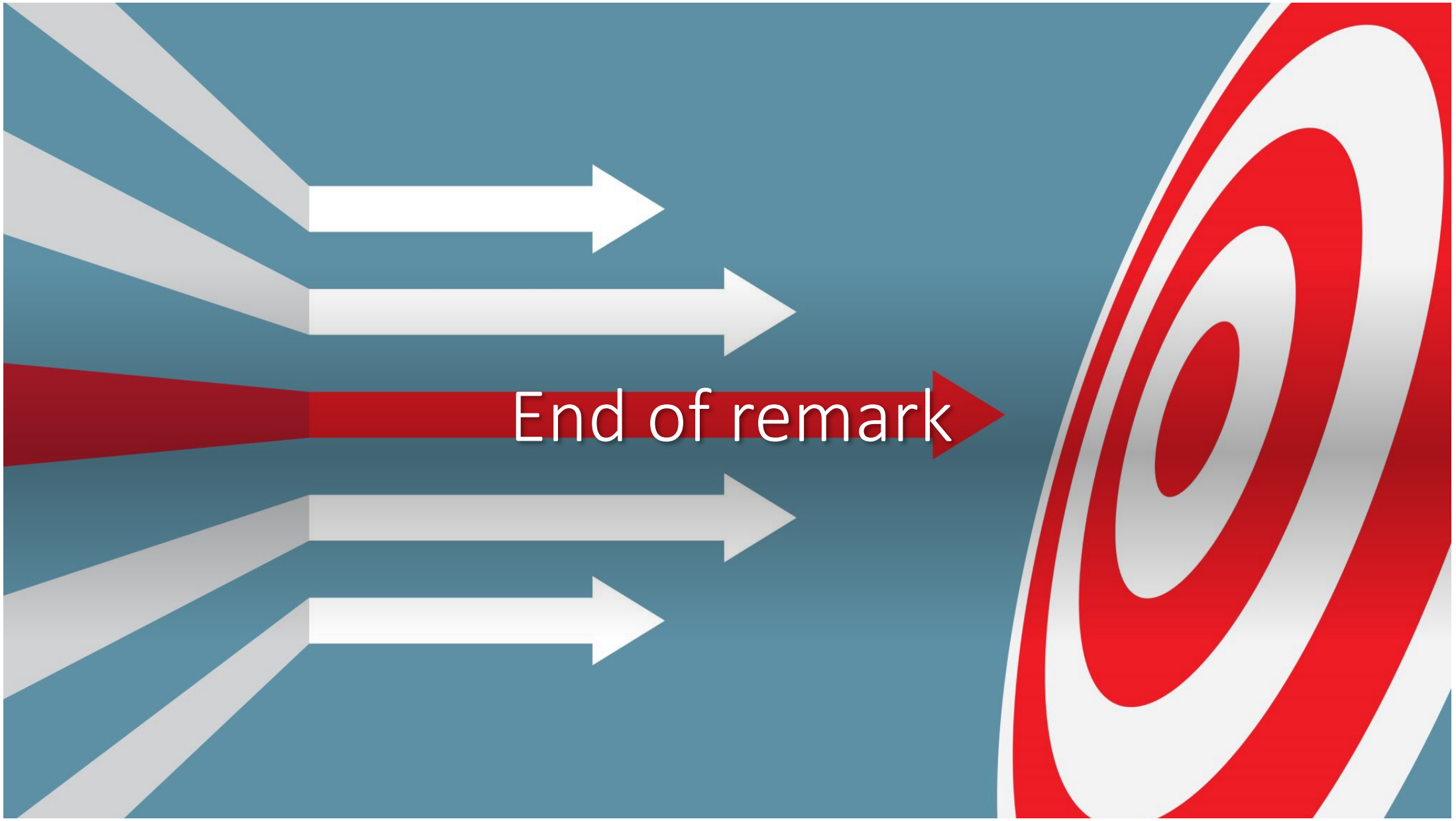
$\Rightarrow$

```
pc := max(pc, x.level);
if x.value
then assert (pc <= y.level); y := true
else assert (pc <= y.level); y := false
```

H

$$H \not\leq L$$

8


End of remark

The type system

Typing expressions

- Judgement $\Gamma \vdash \mathbf{e} : \ell$
- Intuitive meaning:
  According to environment Γ, expression **e** has type (i.e., label) ℓ.

We define the type system in default

any constant will be public

$$\frac{}{\Gamma \vdash n: \bot}$$

$\bot$ Is the least restrictive type

The minimum of the lattice (e.g. L)

$$\frac{\Gamma(x) = \ell}{\Gamma \vdash x: \ell}$$

$$\frac{\Gamma \vdash e1: \ell 1, \Gamma \vdash e2: \ell 2}{\Gamma \vdash e1 + e2: \ell 1 \sqcup \ell 2}$$

↑ sup in the lattice of security level

Let Γ(**x**)= L and Γ(**y**)= H.
What is the type of **x+y+1**?

$$\dfrac{\dfrac{\Gamma(\mathtt{x}) = \mathrm{L}}{\Gamma \vdash \mathtt{x} : \mathrm{L}} \qquad \dfrac{\Gamma(\mathtt{y}) = \mathrm{H}}{\Gamma \vdash \mathtt{y} : \mathrm{H}} \qquad \Gamma \vdash \mathtt{1} : \mathrm{L}}{\Gamma \vdash \mathtt{x + y + 1} : \mathrm{H}}$$

Typing judgment for statements:

Typing statements: implicit flow

Judgements of the form

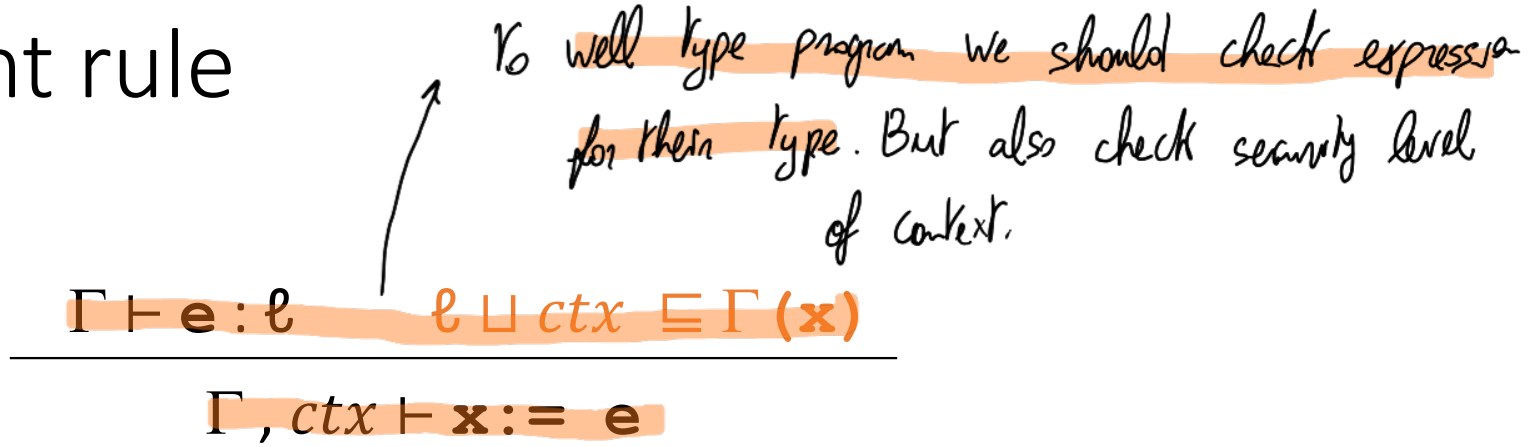
$$\Gamma, ctx \vdash \mathbf{c}$$

Γ is the **environment** mapping variables to security labels

cxt is the **context** label collecting the type information associated to the implicit flow

Intuitive meaning: According to environment Γ, and context label ctx, statement **c** is type correct.

Assignment rule



INTUITION

Γ(x) should be at least as restrictive as $\ell = \Gamma$(e) (**to prevent explicit flows**) and at least as restrictive as ctx (**to prevent implicit flows**)

IF Statement : type the guard: extend context taking into account security level of the guard : we evaluate c_1 or c_2 by taking security level that is the least upper bound.

$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c1} \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c2}}{\Gamma, ctx \vdash \mathbf{if\ e\ then\ c1\ else\ c2}}$$

↳ Conditional is well typed in case guard is comparable

To say if is type correct, then the premises must hold; assuming e is of type ℓ, the the blocks c_1/c_2 have to be correct

$$\frac{\Gamma \vdash \mathbf{y>0} : \Gamma(\mathbf{y}) \qquad \dfrac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \Gamma(\mathbf{y}) \sqcup L \vdash \mathbf{x:=1}} \qquad \dfrac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \Gamma(\mathbf{y}) \sqcup L \vdash \mathbf{x:=2}}}{\Gamma, L \vdash \mathbf{if\ y>0\ then\ x:=1\ else\ x:=2}}$$

↑ context is Low.

$$\dfrac{\Gamma \vdash \mathbf{y>0} : \Gamma(\mathbf{y}) \qquad \dfrac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \Gamma(\mathbf{y}) \sqcup L \vdash \mathbf{x:=1}} \qquad \dfrac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \Gamma(\mathbf{y}) \sqcup L \vdash \mathbf{x:=2}}}{\Gamma, L \vdash \mathbf{if\ y>0\ then\ x:=1\ else\ x:=2}}$$

QUESTION:

*What is the relation between Γ(**x**) and Γ(**y**), such that the above judgement can be proved?*

$$\frac{\Gamma \vdash \mathbf{y>0} : \Gamma(\mathbf{y}) \qquad \dfrac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \Gamma(\mathbf{y}) \sqcup L \vdash \mathbf{x:=1}} \qquad \dfrac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \Gamma(\mathbf{y}) \sqcup L \vdash \mathbf{x:=2}}}{\Gamma, L \vdash \mathbf{if\ y>0\ then\ x:=1\ else\ x:=2}}$$

QUESTION:

*What is the relation between Γ(**x**) and Γ(**y**), such that the above judgement can be proved?*

$$\frac{\Gamma(\mathbf{y}) \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, L \vdash \mathbf{if\ y>0\ then\ x:=1\ else\ x:=2}}$$

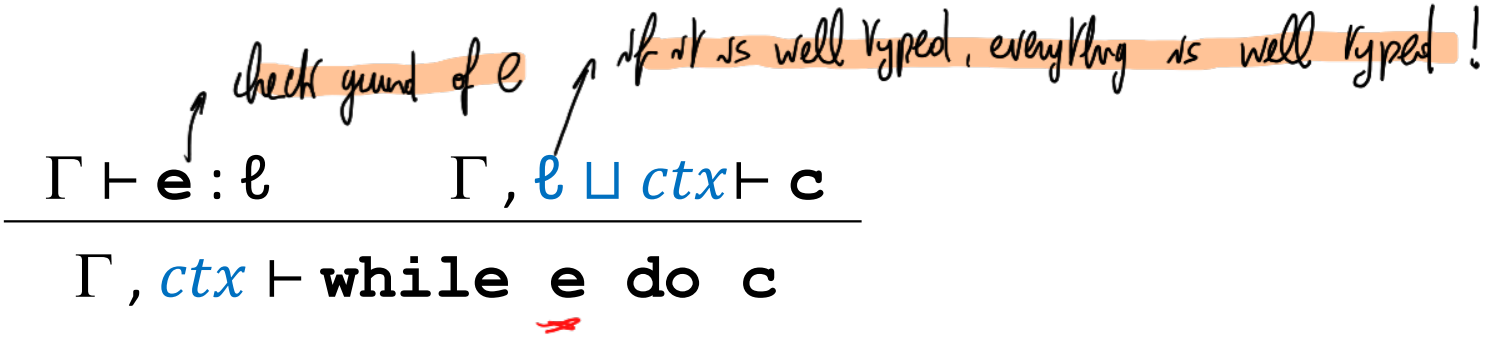
context for outer conditional

We reapply rule of conditionals, so $\Gamma(z) \sqcup \Gamma(y) \sqcup L$

$$\Gamma, \Gamma(\mathbf{z}) \sqcup L \vdash \mathbf{if\ y>0\ then\ x:=1\ else\ x:=2}$$

$$\frac{\Gamma \vdash \mathbf{z>0} : \Gamma(\mathbf{z}) \qquad \Gamma, \Gamma(\mathbf{z}) \sqcup L \vdash \mathbf{x:=3}}{\Gamma, L \vdash \mathbf{if\ z>0\ then\ \{if\ y>0\ then\ x:=1\ else\ x:=2\}\ else\ \{x:=3\}}}$$

While Statement



Sequencing

$$\frac{\Gamma, ctx \vdash \texttt{c1} \qquad \Gamma, ctx \vdash \texttt{c2}}{\Gamma, ctx \vdash \texttt{c1;c2}}$$

evaluate c_1 then c_2. If well typed, great.

If c_1 is well typed as well as c_2, $c_1; c_2$ is well typed

$$\dfrac{\dfrac{\Gamma, \ell \sqcup \Gamma(\mathtt{e}) \vdash \mathtt{x:=1} \quad \Gamma, \ell \sqcup \Gamma(\mathtt{e}) \vdash \mathtt{x:=2}}{\Gamma, \ell \vdash \mathtt{if\ e\ then\ \{x:=1\}\ else\ \{x:=2\}}} \qquad \Gamma, \ell \vdash \mathtt{x:=3}}{\Gamma, \ell \vdash \mathtt{if\ e\ then\ \{x:=1\}\ else\ \{x:=2\};\ x:=3}}$$

If premises are satisfied, result is satisfied

$\Gamma(x) = Y$

The conclusion is well typed if the premises are satisfied.

Static type system

Assignment-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \ell \sqcup ctx \sqsubseteq \Gamma(\mathtt{x})}{\Gamma, ctx \vdash \mathtt{x := e}}$$

If-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathtt{c1} \qquad \Gamma, \ell \sqcup ctx \vdash \mathtt{c2}}{\Gamma, ctx \vdash \mathtt{if\ e\ then\ c1\ else\ c2}}$$

While-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathtt{c}}{\Gamma, ctx \vdash \mathtt{while\ e\ do\ c}}$$

Sequence-Rule:
$$\frac{\Gamma, ctx \vdash \mathtt{c1} \qquad \Gamma, ctx \vdash \mathtt{c2}}{\Gamma, ctx \vdash \mathtt{c1;c2}}$$

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}; \; y := z$$

Note:
the context label ctx can be set to the bottom of the lattice.
Why?

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}; \; y := z$$

SEQUENCING RULE

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}$$

$$\Gamma, \bot \vdash y := z.$$

$$\Gamma, \bot \vdash y := z$$

Holds IF

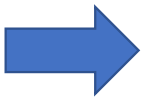
$$\Gamma(z) \sqcup \bot \sqsubseteq \Gamma(y)$$

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}$$

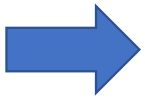
HOLDS IF

$$\Gamma \vdash x > 0 : \Gamma(x)$$

$$\Gamma, \bot \sqcup \Gamma(x) \vdash z := 1$$
$$\Gamma, \bot \sqcup \Gamma(x) \vdash z := 2$$



$$\Gamma, \Gamma(x) \vdash z := 1$$
$$\Gamma, \Gamma(x) \vdash z := 2.$$



$$\Gamma(x) \sqsubseteq \Gamma(z)$$

Putting everything together

$$\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}$$

HOLDS IF

$$\Gamma(x) \sqsubseteq \Gamma(z) \; and \; \Gamma(z) \sqsubseteq \Gamma(y)$$

$$\dfrac{\dfrac{\Gamma \vdash x > 0 : \Gamma(x) \quad \dfrac{\Gamma(x) \sqsubseteq \Gamma(z)}{\Gamma, \Gamma(x) \vdash z := 1} \quad \dfrac{\Gamma(x) \sqsubseteq \Gamma(z)}{\Gamma, \Gamma(x) \vdash z := 2}}{\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}} \quad \dfrac{\Gamma(z) \sqsubseteq \Gamma(y)}{\Gamma, \bot \vdash y := z}}{\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}; \; y := z}$$

$$\Gamma(x) \sqsubseteq \Gamma(z) \; and \; \Gamma(z) \sqsubseteq \Gamma(y)$$

Program is well typed if
Γ(x) = L, Γ(z) = L and Γ(y) = H

$$\dfrac{\dfrac{\Gamma \vdash x > 0 : \Gamma(x) \quad \dfrac{\Gamma(x) \sqsubseteq \Gamma(z)}{\Gamma, \Gamma(x) \vdash z := 1} \quad \dfrac{\Gamma(x) \sqsubseteq \Gamma(z)}{\Gamma, \Gamma(x) \vdash z := 2}}{\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}} \quad \dfrac{\Gamma(z) \sqsubseteq \Gamma(y)}{\Gamma, \bot \vdash y := z}}{\Gamma, \bot \vdash \textbf{if } x > 0 \textbf{ then } z := 1 \textbf{ else } z := 2 \textbf{ end}; \; y := z}$$

$$\Gamma(x) \sqsubseteq \Gamma(z) \; and \; \Gamma(z) \sqsubseteq \Gamma(y)$$

Program is not well typed if
Γ(x) = L, Γ(z) = H, Γ(y) = L

Type checking Information Flow

Static type system

Assignment-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \ell \sqcup ctx \sqsubseteq \Gamma(\mathtt{x})}{\Gamma, ctx \vdash \mathtt{x := e}}$$

If-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathtt{c1} \qquad \Gamma, \ell \sqcup ctx \vdash \mathtt{c2}}{\Gamma, ctx \vdash \mathtt{if\ e\ then\ c1\ else\ c2}}$$

While-Rule:
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathtt{c}}{\Gamma, ctx \vdash \mathtt{while\ e\ do\ c}}$$

Sequence-Rule:
$$\frac{\Gamma, ctx \vdash \mathtt{c1} \qquad \Gamma, ctx \vdash \mathtt{c2}}{\Gamma, ctx \vdash \mathtt{c1;c2}}$$

Type system and Non Interference

If a program does not satisfy NI, then type checking fails.

– Example, where $\Gamma(\mathbf{x}) = \mathrm{L}$ and $\Gamma(\mathbf{y}) = \mathrm{H}$:

$$\frac{\Gamma \vdash \mathbf{y} : \mathrm{H} \qquad \mathrm{H} \sqcup \mathrm{L} \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, \mathrm{L} \vdash \mathbf{x} := \mathbf{y}}$$

Fails

Does not satisfy NI

Type system and Non Interference

$$\Gamma(\mathtt{x}) = \mathrm{L} \text{ and } \Gamma(\mathtt{y}) = \mathrm{H}$$

But, if type checking fails, then the program might or might nor satisfy NI.

$$\frac{\Gamma \vdash \mathtt{y*0} : \mathrm{H} \quad \mathrm{H} \sqcup \mathrm{L} \sqsubseteq \Gamma(\mathtt{x})}{\Gamma, \mathrm{L} \vdash \mathtt{x := y*0}}$$

Fails

Satisfies NI

This type system is not complete.

There is a command **c**, such that noninterference is satisfied, but **c** is not type correct.

$$\Gamma(\mathbf{x}) = \mathrm{H}, \Gamma(\mathbf{y}) = \mathrm{L}$$

if x>0 then y:=1 else y:=1

satisfies noninterference but it does not typecheck.

Why?

This type system is not complete.

The type system has *false negatives:*

- **There are programs that are not type correct, but that satisfy noninterference.**

Can we build a complete mechanism?

- Is there an enforcement mechanism for information flow control that has no false negatives?
  - **A mechanism that rejects only programs that do not satisfy noninterference?**

Can we build a complete mechanism?

No! [Sabelfeld and Myers, 2003]: ***The general problem of confidentiality for programs is undecidable***."

Intuition of the proof strategy:

The halting problem can be reduced to the information flow control problem.

```
fn leak_if_halts(secret: bool, program: fn()) -> bool {
    if halts(program) {
        return secret;
    } else {
        return false;
    }
}
```

Secret is a high-security input.

The function returns secret only **if program halts**.

Otherwise, it returns false.

```
fn leak_if_halts(secret: bool, program: fn()) -> bool {
    if halts(program) {
        return secret;
    } else {
        return false;
    }
}
```

If program **does not halt**, then secret is **never leaked** no interference satisfied.

If program **halts**, then secret may influence the output no interference failure

To check whether this function is noninterfering, we need to decide whether program halts.

```
fn leak_if_halts(secret: bool, program: fn()) -> bool {
    if halts(program) {
        return secret;
    } else {
        return false;
    }
}
```

Wait!! The Halting Problem is **undecidable**.
We **cannot decide**, in general, whether this function leaks information or not.

Noninterference is undecidable in general, so **no static type system can be both sound and complete**.


but all this is not magic ...
We have to demonstrate that the non-interference property holds

Operational semantics (aka the interpreter)

STORE

$s \in STORE, s: Var \rightarrow Value$

INTEREPRETER OF EXPRESSIONS
(as usual)

$\langle e, s \rangle \Downarrow v \in Value$

INTEREPRETER OF COMMANDS

$\langle c, s \rangle \Downarrow s'$

Operational semantics for commands

$$\langle SKIP, s\rangle \Downarrow s$$

$$\frac{\langle e, s\rangle \Downarrow v}{\langle x = e, s\,\rangle \Downarrow s[x = v]}$$

$$\frac{\langle b, s\,\rangle \Downarrow true \qquad \langle c_1, s\,\rangle \Downarrow s'}{\langle if\ b\ then\ c_1\ else\ c_2, s\rangle \Downarrow s'}$$

$$\frac{\langle b, s\,\rangle \Downarrow false \qquad \langle c_2, s\,\rangle \Downarrow s'}{\langle if\ b\ then\ c_1\ else\ c_2, s\rangle \Downarrow s'}$$

$$\frac{\langle b, s\,\rangle \Downarrow false}{\langle while\ b\ do\ c, s\rangle \Downarrow s}$$

$$\frac{\langle b, s\,\rangle \Downarrow true \qquad \langle c, s\,\rangle \Downarrow s' \quad \langle while\ b\ do\ c, s'\rangle \Downarrow s''}{\langle while\ b\ do\ c, s\rangle \Downarrow s''}$$

$$\frac{\langle c_1; c_2, s\rangle \Downarrow s_1' \qquad \langle c_2, s_1'\rangle \Downarrow s_2'}{\langle c_1; c_2, s\rangle \Downarrow s_2'}$$