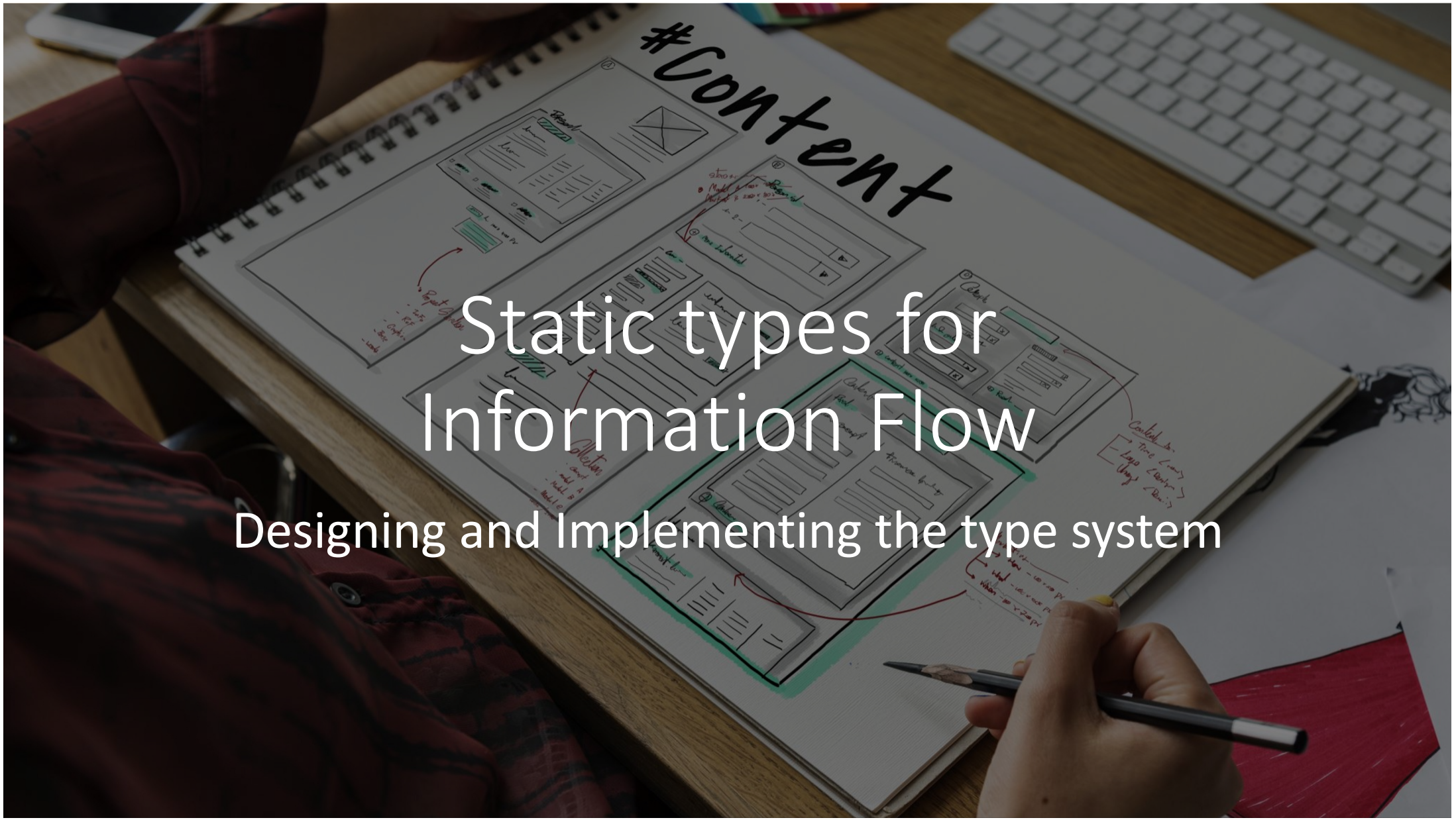


Static types for Information Flow

Designing and Implementing the type system

Managing confidentiality

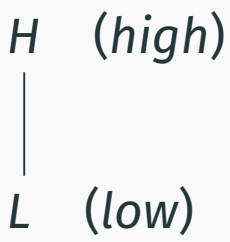
A partial order over confidentiality levels

$$A \sqsubseteq B$$

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

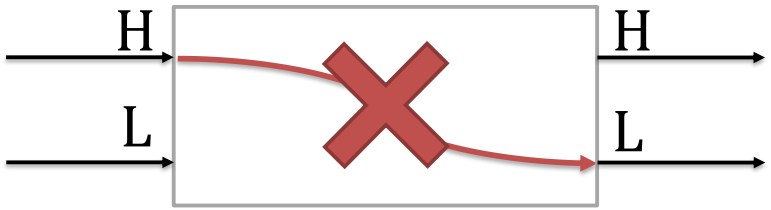
"Someone with credentials B can access information classified A."

Example: the public/secret classification.



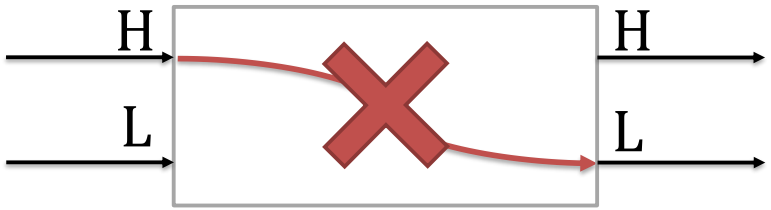
The methodology in a nutshell

- We are given
  - a lattice $L, \sqsubseteq$ of security levels (labels),
  - a program
  - an environment Γ that maps variables to labels.
- Attacker knows L inputs and can observe L outputs.



The methodology in a nutshell

- We are given
  - a lattice $L, \sqsubseteq$ of security levels (labels),
  - a program
  - an environment Γ that maps variables to labels.
- Attacker knows L inputs and can observe L outputs.



Static Program certification: non interference

The Security Lattice in OCAML

```
# type slevel = High | Low;;
type slevel = High | Low

# let leq a b = match a,b with
    | Low, _ -> true
    | _, High -> true
    | _, _    -> false;;
val leq : slevel -> slevel -> bool = <fun>
```

```
# leq Low Low;;
- : bool = true
# leq High Low;;
- : bool = false
#
```

The Security Lattice in OCAML

```
# let join e e' = if e = Low then e' else High ;;
val join : slevel -> slevel -> slevel = <fun>

# let meet e e' = if e = Low then Low else e';;
val meet : slevel -> slevel -> slevel = <fun>
#   let eq = (=);;
val eq : 'a -> 'a -> bool = <fun>
#
```

```
val eq : 'a -> 'a -> bool = <fun>
# eq Low High ;;
- : bool = false
#
```

A simple imperative language

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

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

Abstract Syntax Tree (aka Syntax)

```
# type ide = string;;
type ide = string

# type exp = Var of ide
       | CstI of int
       | CstB of bool
       | Times of exp * exp
       | Eq of exp * exp
       | Not of exp;;
type exp =
    Var of ide
  | CstI of int
  | CstB of bool
  | Times of exp * exp
  | Eq of exp * exp
  | Not of exp
#
```

Abstract Syntax Tree (aka Syntax)

```
# type com =
  Assign of ide * exp
| IF of exp * com * com
| While of exp * com
| Seq of com * com;;
type com =
    Assign of ide * exp
  | IF of exp * com * com
  | While of exp * com
  | Seq of com * com
#
```

The type checker

Our type checker will be a function

```
let rec type_check_com (c:com) (tenv: sleve):  env) (cxt: slevel): bool =
  match c with
    | ... do stuff with the command definitions ... I

type_check_exp exp tenv …..
```

that takes the command to be type checked, the initial type environment, the context, and returns a Boolean value
We return a type error message if the program fails to typecheck.

Type checking expressions

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

type_check_exp (CstI 5) (tenv: slevel env) = Low

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

type_check_exp (var x) (tenv: slevel env)=
lookup tenv x

Summing up

```
let rec type_check_exp (e:exp) (tenv: slevel env) =
  match e with
  | CstI i -> Low
  | CstB b -> Low
  | Var x -> lookup  tenv x
  | _ -> failwith "type error";;
:
# let empty = [];;
val empty : 'a list = []
:
# let te2 = extend te1 "y" High;;
val te2 : (string * slevel) list = [("y", High); ("x", Low)]
# type_check_exp (CstI 5) empty;;
- : slevel = Low
# type_check_exp (Var "y") te2;;
- : slevel = High
#
```

Basic operations

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

```
# let rec type_check_exp (e:exp) (tenv: slevel env) =
  | Times(e1,e2) -> join (type_check_exp e1 tenv)     (type_check_exp e2 tenv)
  | Eq(e1,e2) -> join (type_check_exp e1 tenv) (type_check_exp e2 tenv)
  | Not e1 -> let type_check_exp e1 tenv in
val type_check_exp : exp -> slevel env -> slevel = <fun>
```

```
let rec type_check_exp (e:exp) (tenv: slevel env) =
  match e with
   | CstI i -> Low
   | CstB b -> Low
   | Var x -> lookup  tenv x
   | Times(e1,e2) -> join (type_check_exp e1 tenv) (type_check_exp e2 tenv)
   | Eq(e1,e2) -> join (type_check_exp e1 tenv) (type_check_exp e2 tenv)
   | Not e1 -> type_check_exp e1 tenv
```

TYPE CHECKING STATEMENTS

The type checker function

$$\Gamma, cxt \vdash c$$

let rec type_check_com (c:com) (tenv slevel env) (cxt: slevel) =
match c with ... do stuff ..

val type_check_com : com -> slevel env -> slevel -> bool = <fun>

Assignment

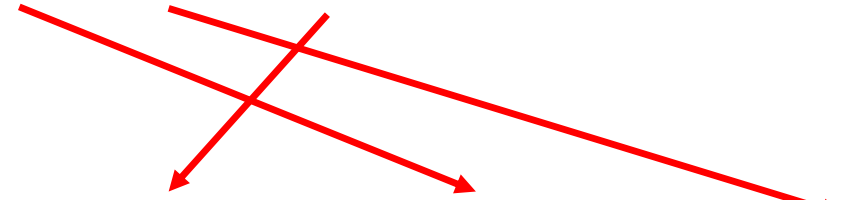
$$\frac{\Gamma \vdash \mathtt{e} : \ell \qquad \ell \sqcup ctx \sqsubseteq \Gamma\,(\mathtt{x})}{\Gamma, ctx \vdash \mathtt{x} := \mathtt{e}}$$

let rec type_check_com (c:com) (tenv: slevel env) (cxt: slevel)

Assign(i, e) -> let t = type_check_exp e tenv in
let cxt1 = (join cxt t) in (sleq cxt1 (lookup tenv i))

Assignment

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



let rec type_check_com (c:com) (tenv: slevel env) (cxt: slevel)

Assign(i, e) -> let t = type_check_exp e tenv in
let cxt1 = (join cxt t) in (sleq cxt1 (lookup tenv i))

Assignment

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

```
let rec type_check_com (c:com) (tenv: slevel env) (cxt: slevel)

Assign(i, e) -> let t = type_check_exp e tenv in
                let cxt1 = (join cxt t) in (sleq cxt1  (lookup tenv i))
```

Assignment

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

```
let rec type_check_com (c:com) (tenv: slevel env) (cxt: slevel)

Assign(i, e) -> let t = type_check_exp e tenv in

                let cxt1 = (join cxt t) in (sleq cxt1  (lookup tenv i))
```

Conditional

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

```
IF(e, c1, c2) -> let t = type_check_exp e tenv in
                 let cxt1 = (join cxt t) in
                     (type_check_com c1 tenv cxt1) &&
                     (type_check_com c2 tenv cxt1)
```

While

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

```
While(e,body) ->  let t = type_check_exp e tenv in
                        let cxt1 = (join cxt t) in
                          (type_check_com body tenv cxt1)
```

Sequencing

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

Seq(c1, c2) -> (type_check_com c1 tenv cxt) &&
(type_check_com c2 tenv cxt);;

Summing up

```
let rec type_check_com (c:com) (tenv: slevel env) (cxt: slevel) =
  match c with
    | Assign(i, e) -> let t = type_check_exp e tenv in
                          let cxt1 = (join cxt t) in (sleq cxt1  (lookup tenv i))
    | IF(e, c1, c2) -> let t = type_check_exp e tenv in
                          let cxt1 = (join cxt t) in
                                  (type_check_com c1 tenv cxt1) &&
                                  (type_check_com c2 tenv cxt1)
    | While(e,body) ->  let t = type_check_exp e tenv in
                           let cxt1 = (join cxt t) in
                          (type_check_com body tenv cxt1)
    | Seq(c1, c2) ->  (type_check_com c1 tenv cxt) &&
                        (type_check_com c2 tenv cxt);;
```

An example

```
# let c1 = IF(Eq(Var "x", CstI 0), Assign("z", CstI 1), Assign("z", CstI 2));;
val c1 : com =
  IF (Eq (Var "x", CstI 0), Assign ("z", CstI 1), Assign ("z", CstI 2))
# let c2 = Assign("y", Var "z");;
val c2 : com = Assign ("y", Var "z")
# let c = Seq(c1, c2);;
val c : com =
  Seq (IF (Eq (Var "x", CstI 0), Assign ("z", CstI 1), Assign ("z", CstI 2)),
   Assign ("y", Var "z"))
```

If x = 0 then z = 1 else z = 2 ;

y = z

An example

```
val tenv : (string * slevel) list = [("z", Low); ("y", High); ("x", Low)]
```

```
# type_check_com c tenv Low;;
- : bool = true
```