Type checking Information Flow

Type checker to avoid that high level info is leaked through low level, public information

Static type system:

check whether or not, in this sequential prog lang, a program is correct wrt flow of info given info Γ, mapping from vars to security level and a context carrying around flow info

Context: something like security level associated to PC.

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

What are the constraints that a, b have to respect to ensure well typedness?

Example

```
if (a==b) then {while (a>0){a = a-1;}
          else {a = a -1;}
```

QUIZ: Identify the information flow security policies over the lattice $\mathbf{L} \sqsubseteq \boldsymbol{H}$, namely the typing context $\boldsymbol{\Gamma}$, under which the program above is well typed

$$\frac{?\,?\,?\,?\,?}{\Gamma, \bot \vdash if\,(a == b)\;then\;\{while\;(a > 0)\{a = a - 1;\}\}else\;\{a = a - 1\}}$$

Context is bottom: This is an assumption. At beginning, we work with no secret information

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

Approach of type system is syntax directed.
Take main operation (conditional) and apply the rule.

check type of guard and check then-else branch with guard for implicit flow to ensure
context

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \bot \sqcup \ell \vdash while(a > 0)\{a = a - 1\} \quad \Gamma, \bot \sqcup \ell \vdash a = a - 1}{\Gamma, \bot \vdash if\,(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

- Find a, b types to ensure well typedness

NOTE: least upper bound between $\bot$ and ℓ is ℓ.

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

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \bot \sqcup \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \bot \sqcup \ell \vdash a = a - 1}{\Gamma, \bot \vdash if\,(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

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

Now focus on premises: we have 3. The proof goes upwards

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \quad \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if\,(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

$$\frac{\Gamma(x) = \ell}{\Gamma \vdash x : \ell} \qquad \frac{\Gamma \vdash e1 : \ell 1, \Gamma \vdash e2 : \ell 2}{\Gamma \vdash e1\ op\ e2 : \ell 1 \sqcup \ell 2}$$

explain typing rule for variable (just lookup in env) and for expression

lookup in env for type of a

$$\frac{\boldsymbol{\Gamma \vdash a = \Gamma(a) = \ell_a, \Gamma \vdash b = \Gamma(b) = \ell_b \quad \ell = \ell_a \sqcup \ell_b}}{\boldsymbol{\Gamma \vdash (a == b) : \ell}}$$

$$\frac{\boldsymbol{\Gamma \vdash (a == b) : \ell} \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \quad \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

While-Rule:

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

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$

from eval of the guard we have lsr const.

constant low, $\bot \sqcup \ell_a$

extend ctx with type of $(a>0)$ expression

$$\frac{\Gamma \vdash a > 0 : \ell_a \qquad \Gamma, \ell_a \sqcup \ell \vdash a = a - 1}{\Gamma, \ell \vdash while(a > 0)\{a = a - 1\}}$$

Then branch: check if it is well typed!

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \quad \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if\,(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

While-Rule:

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

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$

Assignment-Rule:

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

$$\frac{\Gamma \vdash a > 0 : \ell_a \qquad \Gamma, \ell_a \sqcup \ell \vdash a = a - 1}{\Gamma, \ell \vdash while(a > 0)\{a = a - 1\}}$$

$$\frac{\ell_a \sqcup \ell_a \sqcup \ell \sqsubseteq \ell_a}{\Gamma, \ell_a \sqcup \ell \vdash a = a - 1}$$

$\Gamma(a) \sqcup \bot$

evaluate type of RHS assignment with rule of assignment

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \quad \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$

$\ell_a \sqcup \ell \sqsubseteq \ell_a$ 2nd constraint satisfy

$$\frac{\Gamma \vdash a > 0 : \ell_a \qquad \Gamma, \ell_a \sqcup \ell \vdash a = a - 1}{\Gamma, \ell \vdash while(a > 0)\{a = a - 1\}}$$

$$\frac{\ell_a \sqcup \ell_a \sqcup \ell \sqsubseteq \ell_a}{\Gamma, \ell_a \sqcup \ell \vdash a = a - 1}$$

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

Assignment-Rule:

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

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$
$\ell_a \sqcup \ell \sqsubseteq \ell_a$

$$\frac{\ell_a \sqcup \ell \sqsubseteq \ell_a}{\Gamma, \ell \vdash a = a - 1}$$

Same as before

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \quad \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

- We see case by case to see well typedness

$\ell_a = H$

$H = H \sqcup \ell_b = H$

$H \sqcup \ell = H \sqsubseteq H$

OK

CONSTRAINT

what we gather $\begin{cases} \ell = \ell_a \sqcup \ell_b \\ \ell_a \sqcup \ell \sqsubseteq \ell_a \end{cases}$

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

$\ell_a = L$

$\ell = L \sqcup \ell_b$

$L \sqcup \ell \sqsubseteq L$ Satisfied only when L is low

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$

$\ell_a \sqcup \ell \sqsubseteq \ell_a$

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

$$\ell_a = L$$

$$\ell = \boldsymbol{L} \sqcup \ell_{\boldsymbol{b}} \Rightarrow \ell_{\boldsymbol{b}} = \boldsymbol{L}$$
$$\boldsymbol{L} \sqcup \ell \sqsubseteq \boldsymbol{L} \Rightarrow \ell = \boldsymbol{L}$$

CONSTRAINT

$$\ell = \ell_{\boldsymbol{a}} \sqcup \ell_{\boldsymbol{b}}$$
$$\ell_{\boldsymbol{a}} \sqcup \ell \sqsubseteq \ell_{\boldsymbol{a}}$$

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

$\ell_a = L$

$\ell = \boldsymbol{L} \sqcup \ell_b \Rightarrow \ell_b = \boldsymbol{L}$

$\boldsymbol{L} \sqcup \ell \sqsubseteq \boldsymbol{L} \Rightarrow \ell = \boldsymbol{L}$

$\ell_a = L \; and \; \ell_b = L$ **OK**

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$

$\ell_a \sqcup \ell \sqsubseteq \ell_a$

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\; then\; \{while\; (a > 0)\{a = a - 1;\}\; else\; \{a = a - 1|}$$

$\ell_b = H$ and $\ell_a = H$ **OK**

CONSTRAINT

$\ell = \ell_a \sqcup \ell_b$

$\ell_a \sqcup \ell \sqsubseteq \ell_a$

$\ell_b = H$ and $\ell_a = L$ **KO** Config does not satisfies non interference

$$\frac{\Gamma \vdash (a == b) : \ell \quad \Gamma, \ell \vdash while(a > 0)\{a = a - 1\} \;\; \Gamma, \ell \vdash a = a - 1}{\Gamma, \bot \vdash if(a == b)\ then\ \{while\ (a > 0)\{a = a - 1;\}\ else\ \{a = a - 1|}$$

Type system and Non Interference

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

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

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

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

Security level of expression vs H because it is a bump between L and H.

This type system is not complete.

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

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

↱ indirect flow over high level variable

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.

- Intuitive idea behind proof

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

we take a secret value

if program terminates, we return secret, otherwise we return false.

There's a leakage only if program halts. If we had a complete type system we would be able to tell if this halts.

(Both soundness and completeness is unobtainable)

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

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

- We need to prove type system is sound
  - To give the intuition of proof, check that in dynamic execution you won't have problems. So we need to introduce an interpreter, abstract representation of execution environment

Operational semantics (aka the interpreter)

For mutable data
STORE

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

↑ function that goes from variables to values

INTEREPRETER OF EXPRESSIONS
(as usual)

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

INTEREPRETER OF COMMANDS

Command

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

Memory

In imperative prog. language, env is a mapping

from pVar → Loc + Value (abstract rep. of memory locations)

and then a Store, $s \in Store$ where $s: Loc \to M.Val$ (Values that can be memorized)

This cannot be stored inside of memory

Y = Y+1

P = env(x)

S(P) = S(P)+1

Operational semantics for commands

$$\langle SKIP, s\rangle \Downarrow s$$ No of

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

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

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

→ current state vs final state

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

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

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

Tech defs and Lemmas

Low Level Equality $s_1 =_L s_2 . iff \ \forall x : L . s_1(x) = s_2(x)$

if I take low level exp. (exp. producing public value)

↑ if I take two stores that are low level equivalent (they are not able to differentiate low level variables)

Lemma 1 L-expressions have the same values in stores Low Level Equivalent

$Assume \ s_1 =_L s_2 \ \langle e, s_1 \rangle \Downarrow v_1 \langle e, s_2 \rangle \Downarrow v_2 \ implies \ v_1 = v_2$

↑ evaluate expression in those states they remain the same

Lemma 2

$if \ H \vdash c \ and \ \langle c, s \rangle \Downarrow s' \ then \ s =_L s'$

↑ well typed in environment in which H is security policy

Lemma 2:

If a certain command C is well typed in an environment where security policy is H, then if C is evaluated in S to S', then $S =_L S'$.

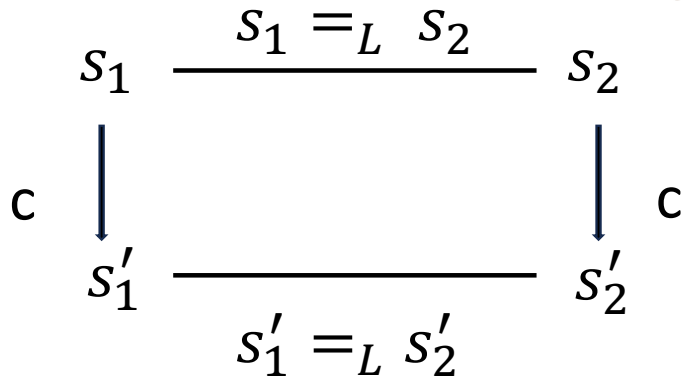
My Idea: If C is well typed you cannot tell differences.

Non Interference Theorem

We want to prove this

$$If\ \Gamma, \perp \vdash c, s_1 =_L s_2, \langle c, s_1\rangle \Downarrow s_1', \langle c, s_2\rangle \Downarrow s_2'\ then\ s_1' =_L s_2'$$

two states low level eq and execute C on s_1 and s_2



and u reach two diff. states that are low level equivalent

To prove non interference theorem, start from two states that are LLE. $s_1 =_L s_2$ and execute C on both states. We reach on both cases two different states that are LLE, We cannot differentiate high values

Proof (outline)

By induction on the derivation of the operational semantics and the structure of the command c.

Since type system is syntactic method we use syntax

$Case\ 1\ x = e\ and\ \Gamma(x) = L$ Assumption

$$\frac{\Gamma(e) \sqsubseteq L}{\Gamma, L \vdash x = e} \quad \langle e, s_1 \rangle \Downarrow v_1 \quad \langle e, s_2 \rangle \Downarrow v_2$$

IT IS WELL TYPED

$s_1 =_L s_2$

s_1 ——— s_2

x=e ↓ ↓ x=e

$s_1[x = v_1]$ ——— $s_2[x = v_2]$

$s_1' =_L s_2'$

By Lemma 1

$v_1 = v_2$

You have to prove that type system is correct

Idea behind; how can we prove? Show two rules.

We use induction over the syntax of the programming language:

Case 1: Assignment between x and e and assume x is low level.

So type of e must be compatible with L, And we apply lemma 1.

In one case we move from s_1 to $s_1[x=v_1]$, in the other s_2 to $s_2[x=v_2]$, but by lemma 1, since $s_1 =_L s_2$, $v_1 = v_2$ so $s_1[x=v_1] =_L s_2[x=v_2]$.

Proof (outline)

$Case\ 1\ x = e\ and\ \Gamma(x) = H$

As for the previous case and then it is enough apply Lemma 2

$$S_1 =_i S_1'$$

$$S_2 =_i S_2'$$

$$S_1 =_i S_2 \Rightarrow S_1' =_i S_2'$$

Other issues: termination

The type system we discussed does not prevent leaks through termination channels

Example of termination channel:

Consider this program

```
while h != 0 do skip; l:=2
```

Well typed given our type system but this does not satisfy non interference w.r.t termination: if program terminates we can say that h=0.

where `h` is a high variable and `l` is a low variable.

The program leaks over termination channel: It does not satisfy termination sensitive non interference.

But the program is type correct: It satisfies non interference.

Assume that $\Gamma, cxt \vdash skip$

- Problem is that guard is high

One can strengthen the typing rule for while-statement, to allow only low guard expression:

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

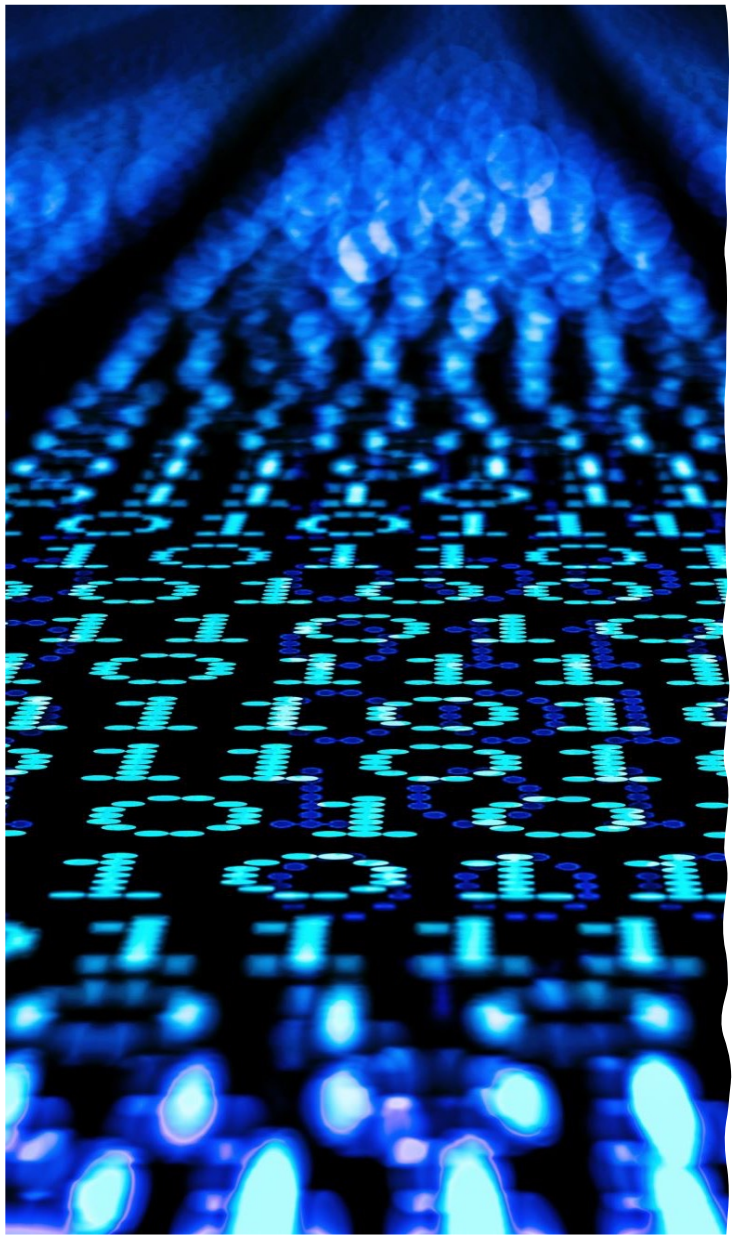
Now type correctness implies termination sensitive NI

- A choice will have drawbacks

… However

Drawback! I reject correct programs to avoid side channel attacks

while h > 0 **do** h= h+1; **is not well typed**!!!



Other programming abstractions

The notion of subtyte

Types denote sets of values

Subtyping is a relation between types induced by the subset relation between value sets

Informal intuition: If τ is a subtype of σ then any expression with type τ also has type σ

Acctual usage: If τ is a subtype of σ then any expression of type τ can be used in a context that expects a σ

Subtyping in Object-oriented languages

- Subtyping is a fundamental property of object-oriented languages
- If **S** is a subclass of **C** then an instance of **S** can be used where an instance of **C** is expected
- Slogan: "**subclassing ⇒ subtyping" philosophy**

Formal definizion

Rule of subsumption If $\tau<\sigma$ then an expression of type τ has type σ

expression e that has type τ — but τ is a subtype of σ

$$\frac{\Gamma \vdash e : \tau \qquad \tau < \sigma}{\Gamma \vdash e : \sigma}$$

then e has type σ.

Defining subtyping

The formal definition of subtyping is by derivation rules for the judgment $\tau < \sigma$

We start with subtyping on the base types (int<real or nat<int)

The rules are language dependent and are typically based directly on types-as-sets arguments

We then make subtyping a preorder (reflexive and transitive)

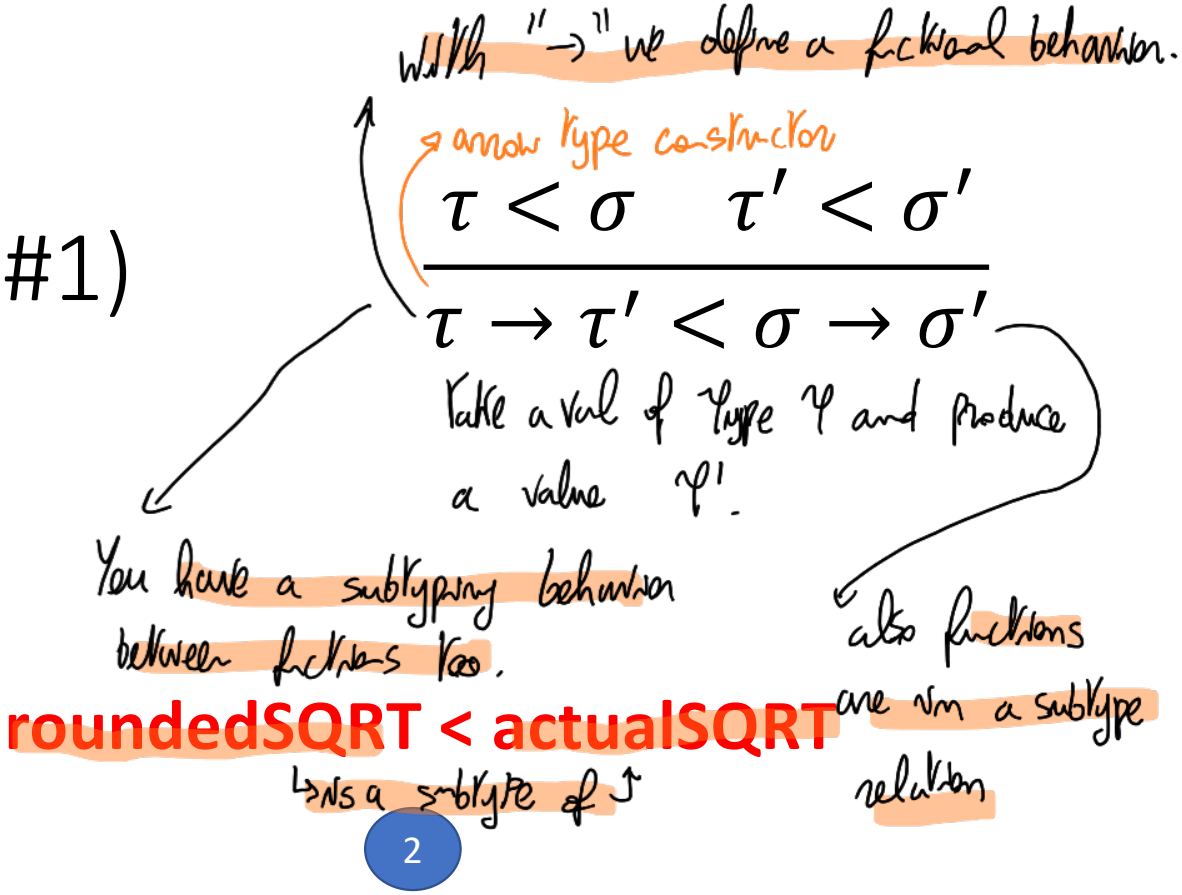
Then we build-up subtyping for “larger” types

Typing of functions (#1)

let's arrive to functions

roundedSQRT: R -> Z
actualSQRT: R -> R
Z < R

1



3

float result = roundedSQRT //2
float result = actualSQRT //2.23

Typing of functions (#1)

$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

Assume $\Gamma \vdash f : int \to bool$

What about $\Gamma \vdash f\ 0.5 : bool$? (real)

You now apply f to 0.5, a float. If you apply the rule of subtype result will be okay.

Typying of functions (#1)

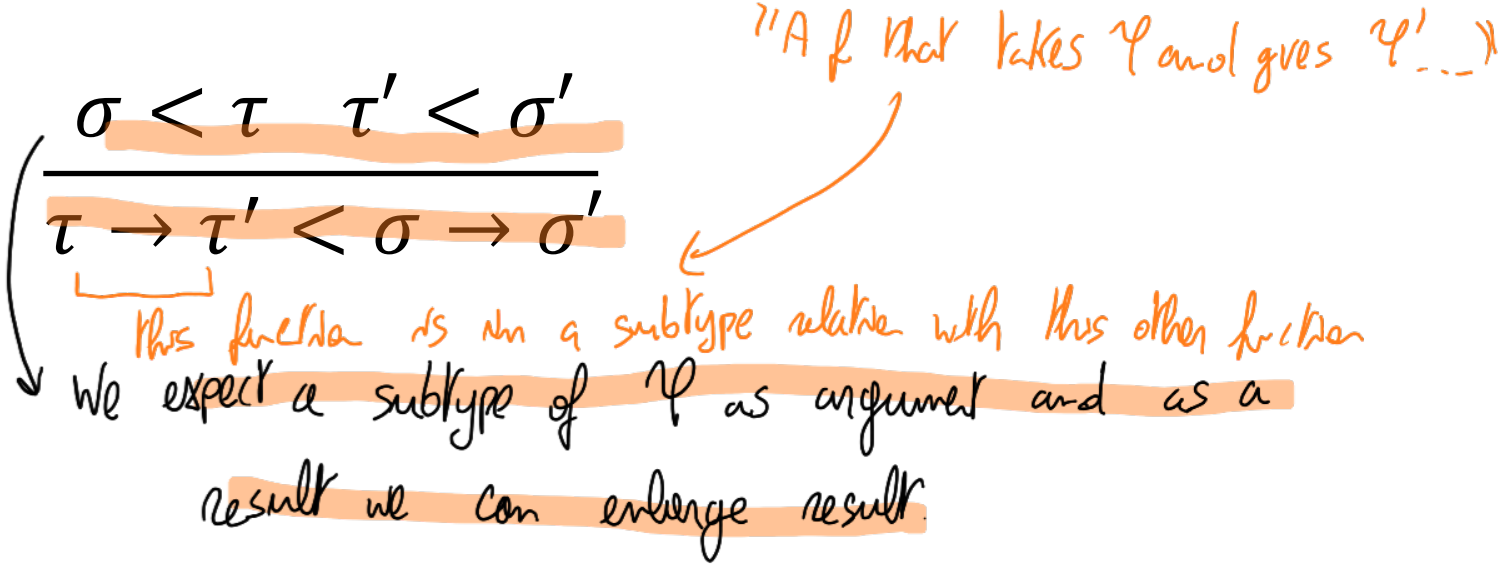
$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

Assume $\Gamma \vdash f : int \to bool$ What about $\Gamma \vdash f\ 0.5 : bool$??

$$\frac{\Gamma \vdash f : int \to bool \quad \dfrac{int < real \quad bool < bool}{int \to bool < real \to bool}}{\Gamma \vdash f : real \to bool} \quad \Gamma \vdash 0.5 : real$$
$$\Gamma \vdash f\ 0.5 : bool$$

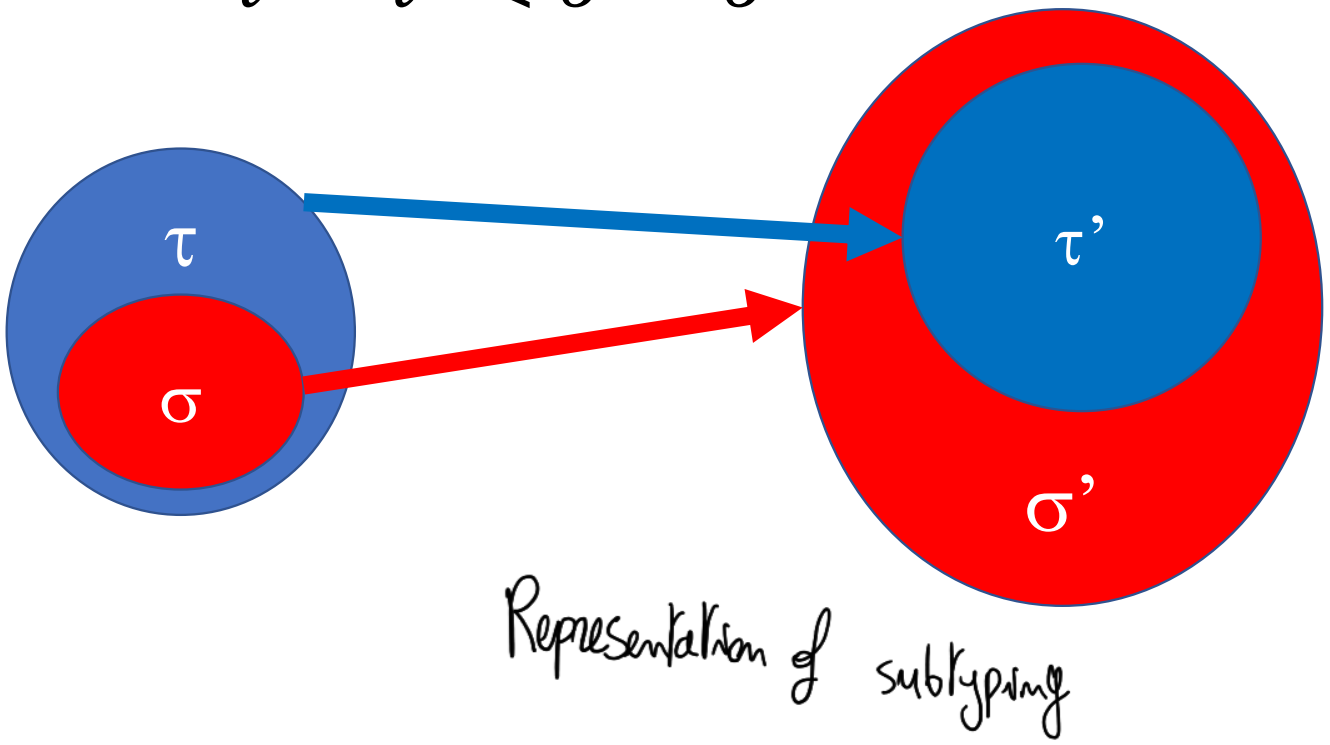
We know that 0.5 is not a valid argument for f.
The rule is unsound!!!

Correct Function Subtyping



Correct Function Subtyping

$$\frac{\sigma < \tau \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$



Correct Function Subtyping

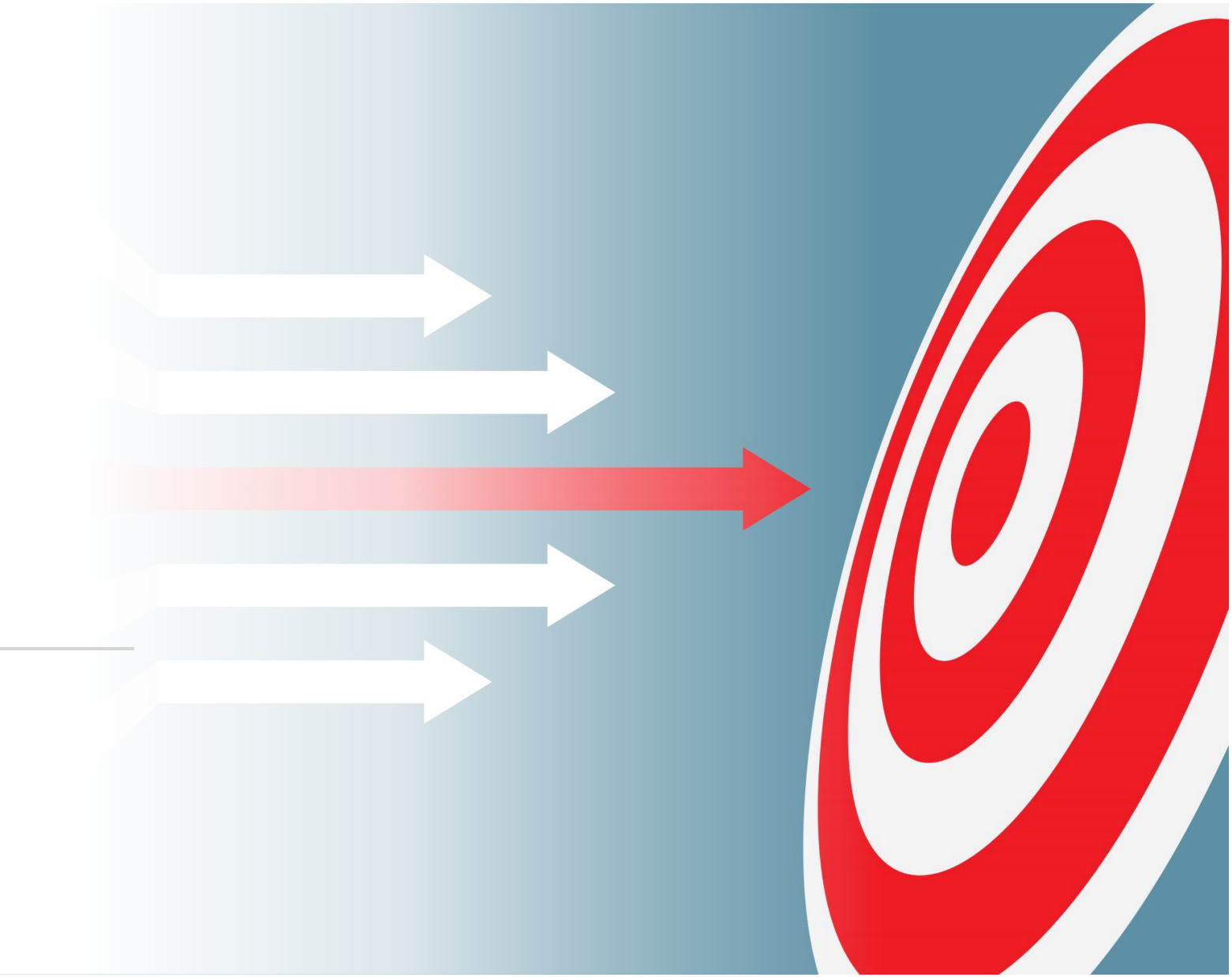
$$\frac{\sigma < \tau \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

Informal correctness (argument)

1. Pick a function **f: A -> A'**
2. Function **f** takes an argument **a** of **A**.
3. Function **f** also accepts an argument **b** of **B** since **B** < **A**
4. Function **f** returns a resullt **a'** of **A'**. The value a' can be also viewed as element of **B'** since **A'** < **B'**.

Moral: function f can be also typed as **B -> B'**

REMARK



Correct Function Subtyping

$$\frac{\sigma < \tau \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

Use the arrow type constructor

We say that the arrow type constructor is

↓ it is a constructor in the type theory

1. **Covariant** in the result type
2. **Controvariant** in the argument type

More on subtyping

In OOP you have generics!

- Suppose A and B are types, and T<U> denotes application of a type constructor T with type argument U.
- Example in Java
  List<T> list = new ArrayList<T>();
  list – object of *List interface*.
  T – The generic type parameter passed during list declaration
  ArrayList – the type constructor

→ providing a type you can create this list

If A is subtype of B, list of A is a subtype of list of B. But in Java, apart from Arrays, this is not the case.

When you look at prog. language you have to look at all the features you provide

More on subtyping

- Suppose A and B are types, and T<U> denotes application of a type constructor T with type argument U.
- Within the type system of a programming language, a typing rule for a type constructor T is:
  - covariant if it preserves the ordering of types (<), which orders types from more specific to more generic: If A < B, then T<A> ≤ T<B>;
  - contravariant if it reverses this ordering: If A < B, then T<B> ≤ T<A>;
  - invariant if not covariant and not controvariant

We move to functional prog. languages.

Higher order functions

Starting with a standard type system, we add levels on types:

$$\begin{array}{rll} \tau ::= \texttt{int}^{\ell} \mid \texttt{bool}^{\ell} & & \text{base types} \\ \mid (\sigma \to \tau)^{\ell} & & \text{functions} \\ \mid \texttt{list}(\tau)^{\ell} & & \text{lists} \end{array}$$

basic types that can be low or high

we also have functions

taking types can be low or high

The $\sqsubseteq$ order over levels induces a subtyping relation:

$$\frac{\ell \sqsubseteq \ell'}{\texttt{int}^{\ell} <: \texttt{int}^{\ell'}} \qquad \frac{\sigma' <: \sigma \quad \tau <: \tau' \quad \ell \sqsubseteq \ell'}{(\sigma \to \tau)^{\ell} <: (\sigma' \to \tau')^{\ell'}}$$

A subtyping relationship can be defined looking at security levels

↑ subtype of an integer at high level: generalization for security level

→ functions are values

Typing rules for a pure functional language

$$\Gamma \vdash n : \mathtt{int}^{\ell}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.e : (\sigma \to \tau)^{\ell}}$$

$$\frac{\Gamma \vdash e_1 : (\sigma \to \tau)^{\ell} \qquad \Gamma \vdash e_2 : \sigma \qquad \ell \sqsubseteq \mathit{Label}(\tau)}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool}^{\ell} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau \qquad \ell \sqsubseteq \mathit{Label}(\tau)}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau}$$

Where $\mathit{Label}(\mathtt{int}^{\ell}) = \ell$, $\mathit{Label}((\sigma \to \tau)^{\ell}) = \ell$, etc.

① Premise: body is of type τ in environment enriched with $x:\sigma$ mapping

Still type system is roughly the same

e_1 is an arrow type

argument to be complete must be of type σ, func expects that

FUN DECL: new parameter

PROGRAMMER DECIDES FUN APPLICATION

COMPOSITIONALITY

We need consistency with the label

* But we also need to check that sec. level of func is compatible with sec. level associated with result. We are interested in value produced, so output compatible with result type

Funct pl with mutable entities (assignments)

Mutable states

① Type associated to mutable entities/references

(F. Pottier, V. Simonet, *Information flow inference for ML*, 2002, 2003.)

Types: $\tau ::= \texttt{int}^\ell \mid \texttt{bool}^\ell$ base types

$\mid (\sigma \xrightarrow{pc} \tau)^\ell$ functions

$\mid \texttt{list}(\tau)^\ell$ lists

$\mid \texttt{ref}(\tau)^\ell$ mutable references

Type associated to fun takes into account implicit flow: pc over arrow: security level of execution ①

what is ℓ?

MUTABLE REFERENCE BECAUSE OF THIS ①

↳ I'm modifying a variable associated to mutable variable

We now need to track implicit flows by adding a level *pc* both to the typing judgment

$$pc, \Gamma \vdash e : \tau$$

and to function types as a latent effect

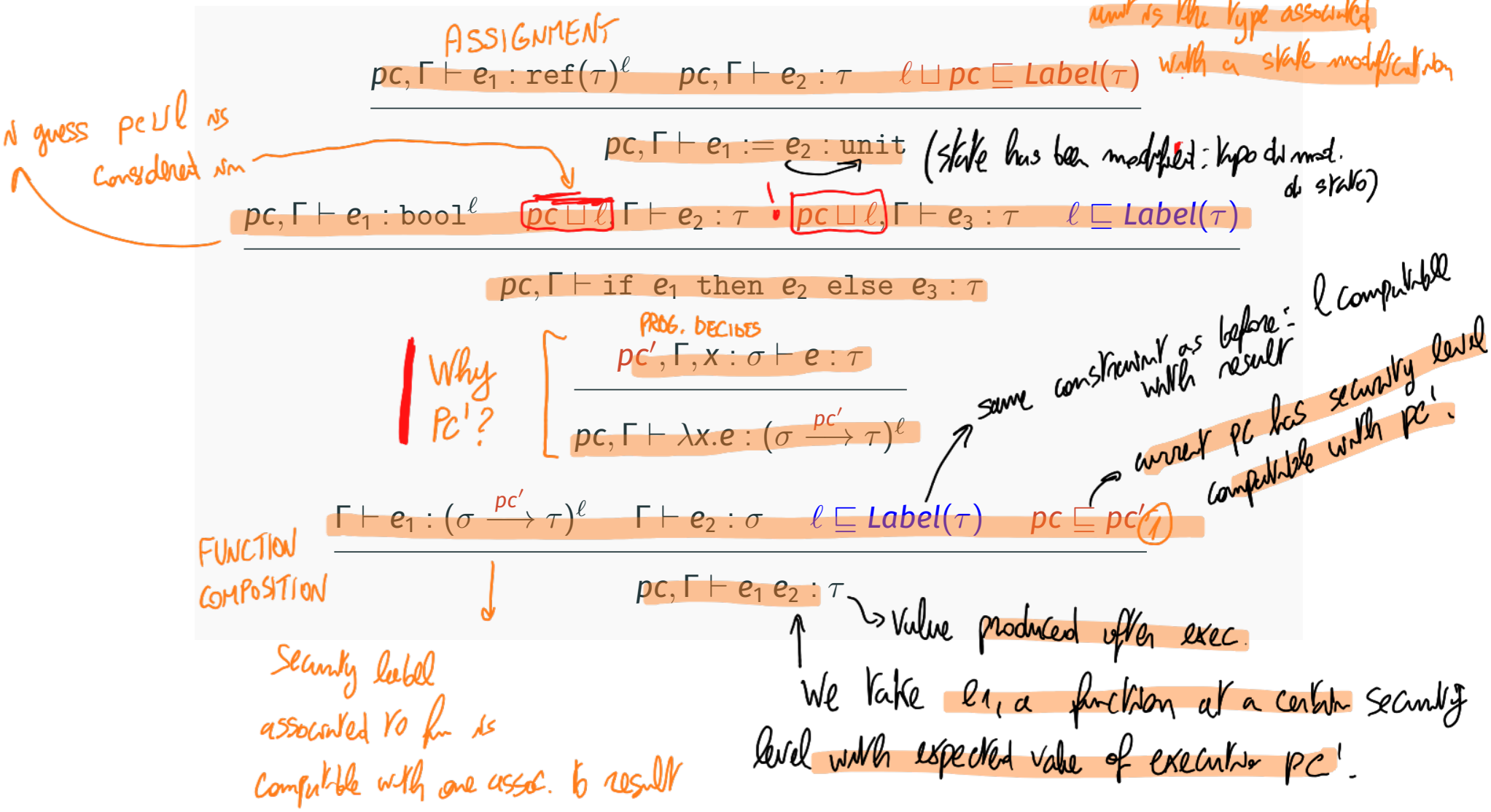
$$(\sigma \xrightarrow{pc} \tau)^\ell$$

security level of implicit flow that I want for function

① We take a value in σ, produce a value in τ and during execution you have to keep track indirect flow due to execution of functions.

- Important point: type associated to function: it now takes into account indirect flow: type is "I take a value in σ, I produce a result in τ. This value has a certain security level ℓ, but we also have implicit flow pc: security level of execution.
  "We are taking a value in σ, producing a value in τ, we have a certain security level and during the execution context context go over pc (IMPLICIT LVL) value associated.
  We want to measure indirect flow due to funct. execution.

Functions + Mutable states



① Current context PC has a security level compatible with pc! During evaluation of flexible I might enlarge the level of context but I cannot exceed.

Object Oriented Languages

Classes may be parameterized to make them generic with respect to some security labels (also called principals).

Class and interface declarations are extended to include an optional set of explicitly declared parameters for managing principals

You can define info flow also for OOPL.

Java Information Flow

- **Java Information Flow (JIF)** is a programming language (an extension of Java) designed to support **secure information flow**.
- **Types for information flow** of JIF centers on how **security labels** are integrated into the type system to enforce **confidentiality** and **integrity policies** statically.

Types for Information Flow in JIF

Each variable, expression, and method in JIF can be annotated with a **security.label** that describes:

- **Who owns the data** (the *owner*).
- **Who can read it** (*confidentiality*).
- **Who can modify it** (*integrity*).

int alice to Bob secret

int{Alice->Bob} secret is a security label

- **Alice owns** the data.
- **Only Alice and Bob** can read the data.
- By default, only Alice can write it unless integrity is separately specified.

Bob can only read

Security Policies

Confidentiality Policies

- Written as A -> B, meaning *principal A allows principal B to read*.
- Enforced to prevent high (secret) to low (public) data leaks.

Security Policies

Confidentiality Policies

- Written as **A -> B**, meaning *principal A allows principal B to read*.
- Enforced to prevent high (secret) to low (public) data leaks.

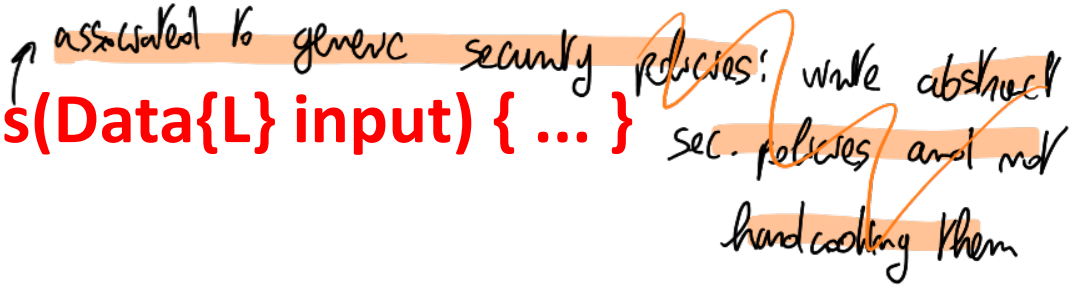
Integrity Policies

- Written as **A <- B**, meaning *principal A allows principal B to influence (aka write)*.
- Enforced to prevent low (untrusted) data from influencing high (trusted) outputs.

Polymorphism

JIF supports **polymorphic labels**, allowing methods or classes to be defined generically over labels:

public <L: label> void process(Data{L} input) { ... }



This enables reusable and secure abstractions without hardcoding specific labels.

Method Labels

Methods can specify the label under which they execute (the *program counter* or pc-label):

void m() where pc <= {Alice->*} { ... }

This constrains what the method can do depending on the security level of the current control flow.

• There is a lattice

Type Enforcement

JIF enforces a **type system with subtyping based on label lattices**:

- Labels form a **security lattice**, where label L1 can flow to L2 if L1 ≤ L2.
- The type system checks that all information flows respect this relation.

Summary

Type System Feature	Purpose	Example
Security label	Annotate data with security policies	int{Alice->Bob}
Confidentiality	Who can read	A -> B
Integrity	Who can write	A <- B
Label polymorphism	Generic code with labels	<L: label> void f(Data{L})
pc-label	Control-flow sensitivity	where pc <= {Alice->*}

Typing issues

The addition of security levels as parameters to classes makes parameterized classes into simple *dependent types* since types contain values.

Be calm … we do not consider dependent type in our course!!!

Downgrading

Requiring all information flow to obey the lattice order $\sqsubseteq$ is too restrictive. Applications need to be able to release secret information to effectively progress.

We have two dual operations:
downgrading confidentiality: **declassify()**
downgrading integrity: **endorse()**.

The justification for downgrading depends on the application

Declassification

- **Declassification** allows a program to **lower the confidentiality level** of data, making it accessible to more principals.
- Declassification **breaks noninterference intentionally**, but under strict conditions, to prevent unauthorized leaks.
- For example, changing data labeled **{Alice->Alice}** to **{Alice->Bob}** lets Bob read data originally meant for Alice only.

Declassification

- **Declassification** allows a program to **lower the confidentiality level** of data, making it accessible to more principals.
- Declassification **breaks noninterference intentionally**, but under strict conditions, to prevent unauthorized leaks.
- For example, changing data labeled **{Alice->Alice}** to **{Alice->Bob}** lets Bob read data originally meant for Alice only.

```
String{Alice->Bob} msg =
  declassify(secret, {Alice->Alice} to {Alice->Bob})
  by authority(Alice);
```

Authority

- To avoid accidental or malicious leaks, JIF requires **explicit authority** of the **owner** of the data:
- Only **Alice** can downgrade confidentiality of {Alice->...} data.
- Prevents untrusted code from bypassing policies.
- Authority must be **declared** at the class or method level:

```
public class SecretMessage {

   // Authority declaration: this class can act for Alice
   authority(Alice)
   public static void main(String[] args) where authority(Alice) {
      try {// Step 1: Create a principal for Alice and Bob
         Principal alice = new Principal("Alice");
         Principal bob = new Principal("Bob");
         // Step 2: Define the label: Alice allows Bob to read
         Label aliceToBob = new Label(alice, alice, bob); // Alice -> Bob
         // Step 3: Declare a secret message labeled {Alice->Alice}
         String{Alice->Alice} secret = "The launch code is 1234";
         // Step 4: Declassify to {Alice->Bob}, using Alice's authority
         String{Alice->Bob} readable =
            declassify(secret, {Alice->Alice} to {Alice->Bob}) by authority(Alice);
         // Step 5: Print (Bob can read this now)
         System.out.println(readable);
      } catch (SecurityException e) {
         System.out.println("Security violation: " + e.getMessage());
      }
   }
}
```

Principle stating security policy (only she can downgrade confident. of Alice → .. declas

↳ make something high to low

```
public class SecretMessage {

    // Authority declaration: this class can act for Alice
    authority(Alice)
    public static void main(String[] args) where authority(Alice) {
        try {// Step 1: Create a principal for Alice and Bob
            Principal alice = new Principal("Alice");
            Principal bob = new Principal("Bob");
            // Step 2: Define the label: Alice allows Bob to read
            Label aliceToBob = new Label(alice, alice, bob); // Alice -> Bob
            // Step 3: Declare a secret message labeled {Alice->Alice}
            String{Alice->Alice} secret = "The launch code is 1234";
            // Step 4: Declassify to {Alice->Bob}, using Alice's authority
            String{Alice->Bob} readable =
                declassify(secret, {Alice->Alice} to {Alice->Bob}) by authority(Alice);
            // Step 5: Print (Bob can read this now)
            System.out.println(readable);
        } catch (SecurityException e) {
            System.out.println("Security violation: " + e.getMessage());
        }
    }
}
```

If one tryies to assign secret directly to a variable with a broader label ({Alice->Bob}) **without declassification**, the compiler will **reject the program**.
This enforces **noninterference**: sensitive data won't leak without explicit permission.

DECLASSIFICATION

EXAMPLES

Password checking

A password check will tell the user whether the password is correct, so the adversary learns something about the secret password:

```
if  (declassify(guess == password),H to L)) login = true;
```

Average of confidential data

We can compute the average of a set of secret salaries and add random noise to make sure that we do not provide too much information about any individual salary.

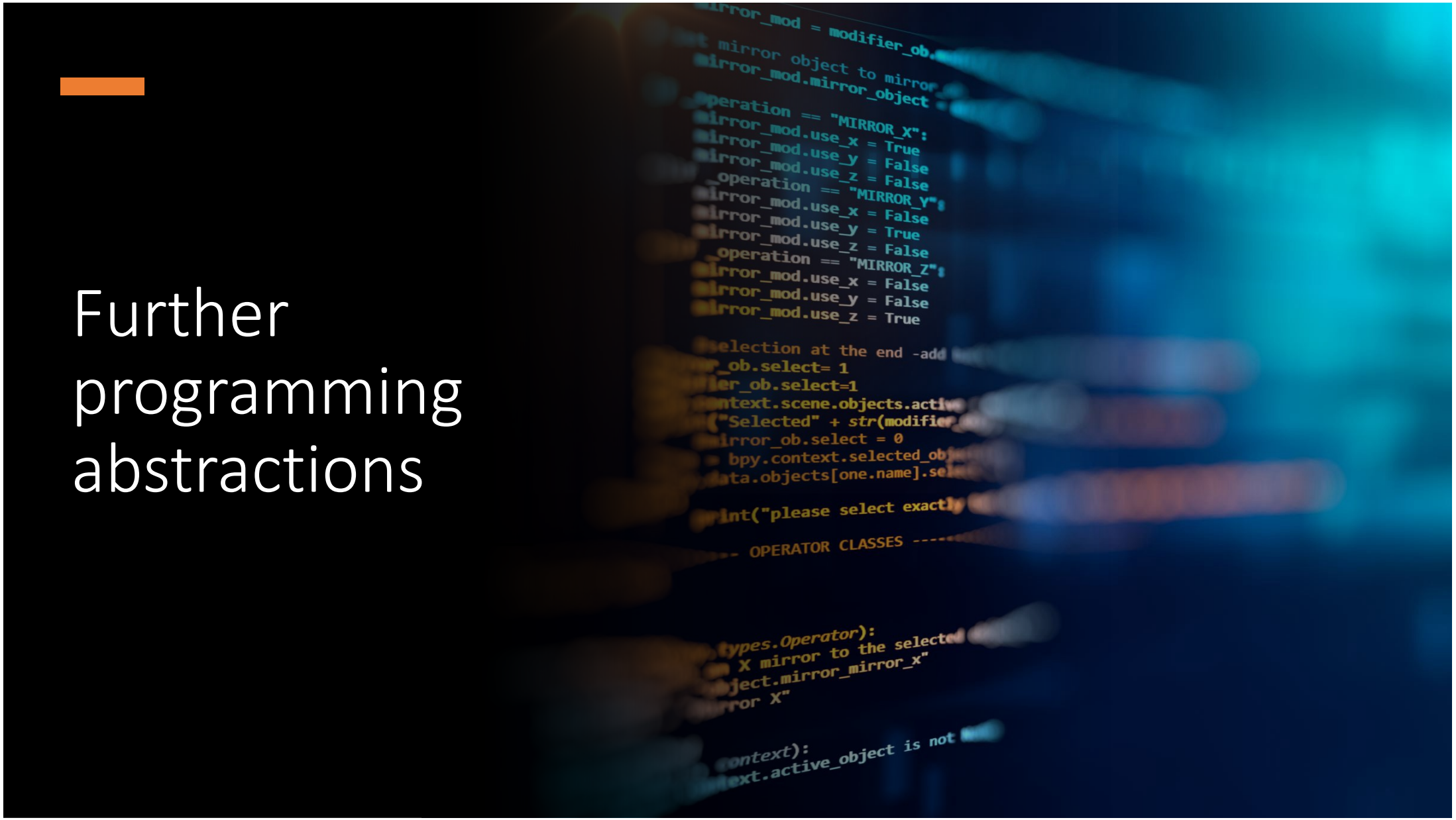
```
float arg = declassify(mean(salaries) + noise, H^-> to L)
```

Robust Declassification

$$\frac{l_2 \sqsubseteq \Gamma(x) \quad \Gamma(y) \sqsubseteq l_1. \quad l_1 \sqsubseteq l_2 \sqcup \Delta(l_1 \sqcup pc)}{\Gamma, pc \vdash declassify(y, l_1 to\ l_2)}$$

The view operator Δ defines the confidentiality level of the declassification: the integrity of pc, l_1 must be both trusted to declassify.

Further programming abstractions



There is a much more problematic definition of non interference with non determinism and threads.

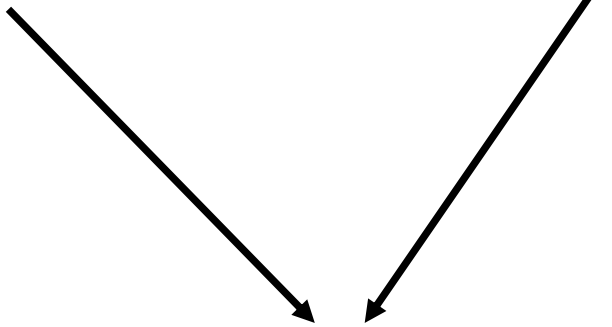
Non Determinism

A nondeterministic programming language is a language which can specify, at certain points in the program (called "choice points"), various alternatives for program flow.

The method of choice between these alternatives is not directly specified by the programmer the executor engine must decide at run time between the alternatives, via some general method applied to all choice points.

Non Determinism: Example

If h > 0 then { l = 0 □ l = 3} else {l = 2 □ l = 4}



Non Determistic Choice

Non Determinism: Example

P = If h > 0 then { l = 0 □ l = 3} else {l = 2 □ l = 4}

$$P \Downarrow s \; and \; s(l) = 3 \; implies \; h > 0$$

The notion of Non Interference we presented is not adequate to handle this class of programs

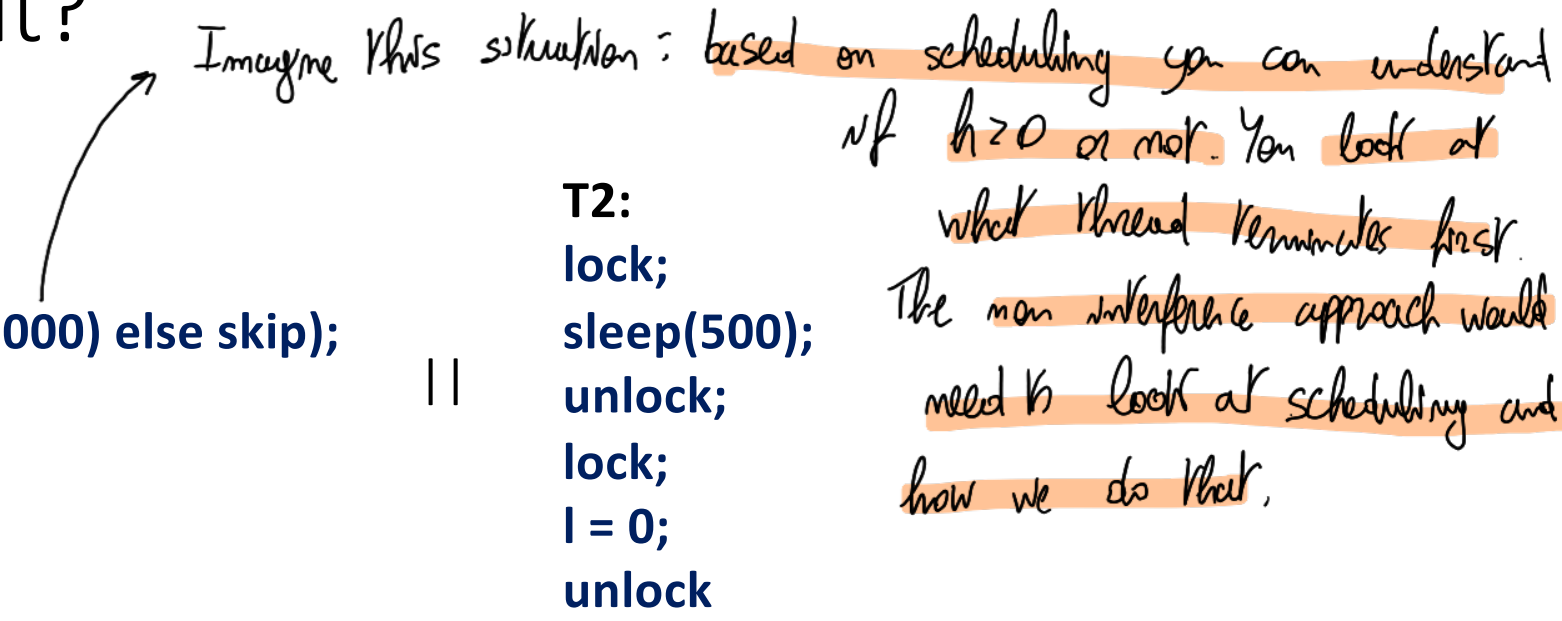
Multi Threading

Multithreading refers to a programming paradigm where multiple threads run concurrently.

A thread is the smallest unit of processing that can be executed by a progream

Multithreading allows a program to perform multiple threads concurrently, rather than sequentially. This means that different threads of the program can execute independently of each other.

What about?



T1:
lock;
(if h>0 then sleep(1000) else skip);
unlock;
lock;
l = 1;
unlock

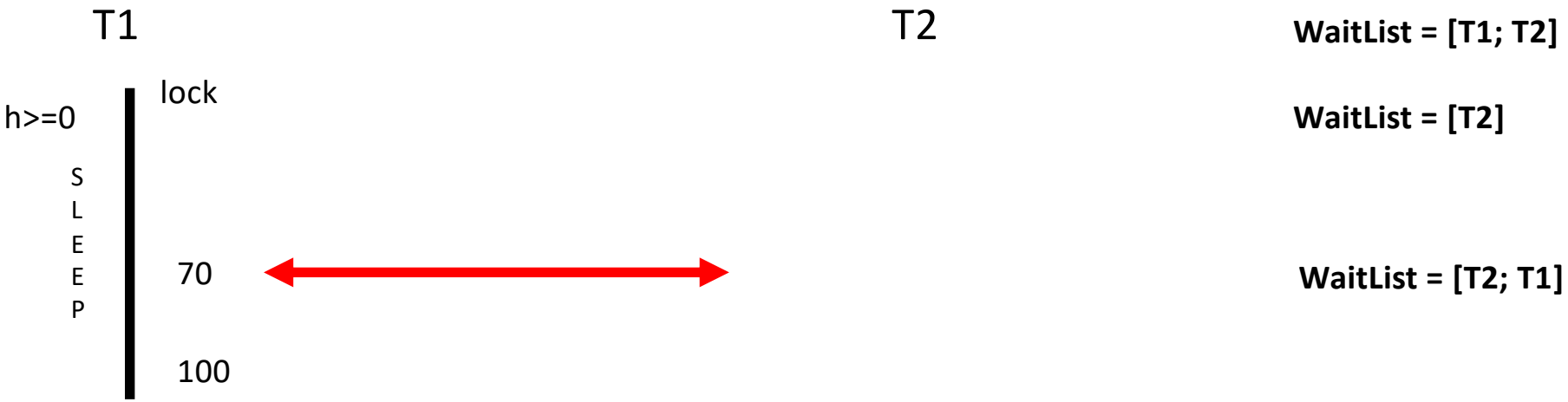
||

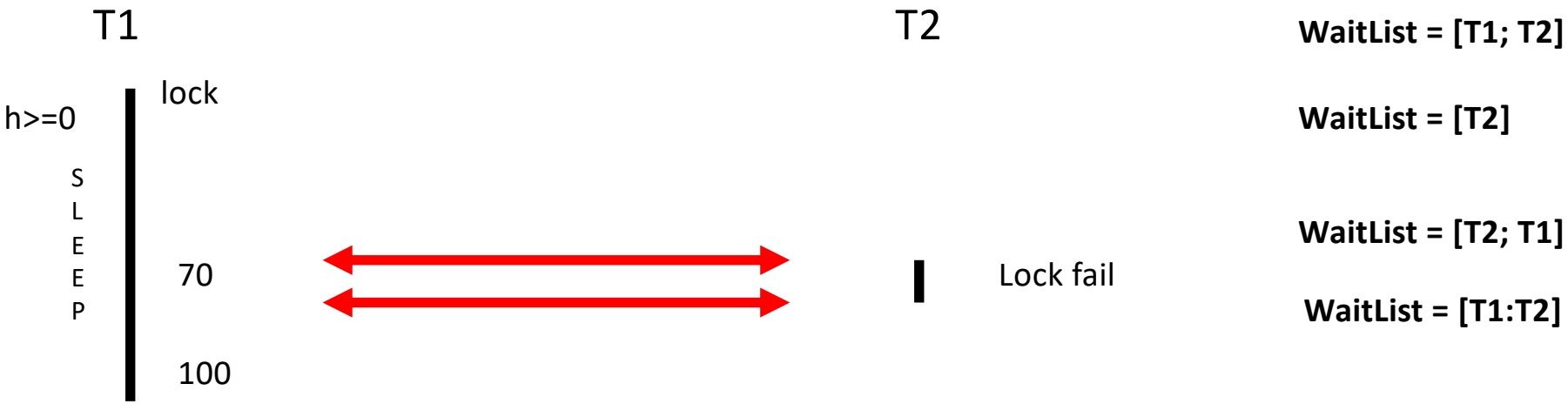
T2:
lock;
sleep(500);
unlock;
lock;
l = 0;
unlock

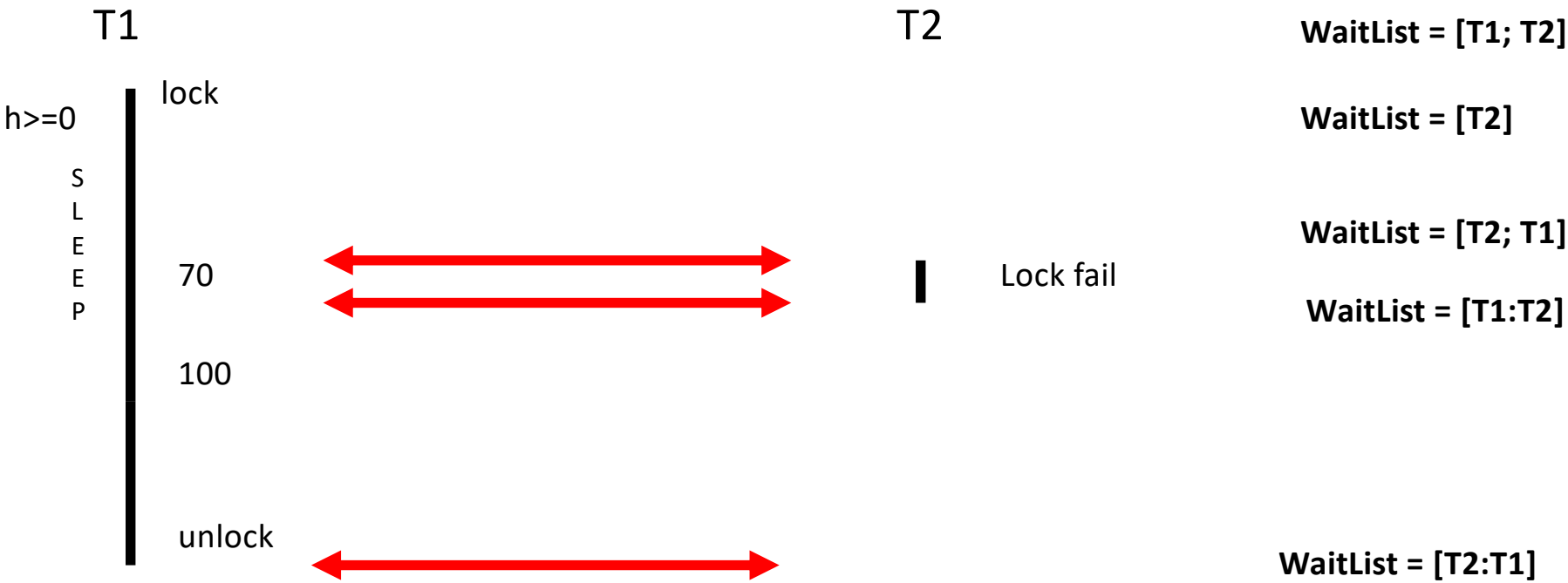
We analyze possible interleavings

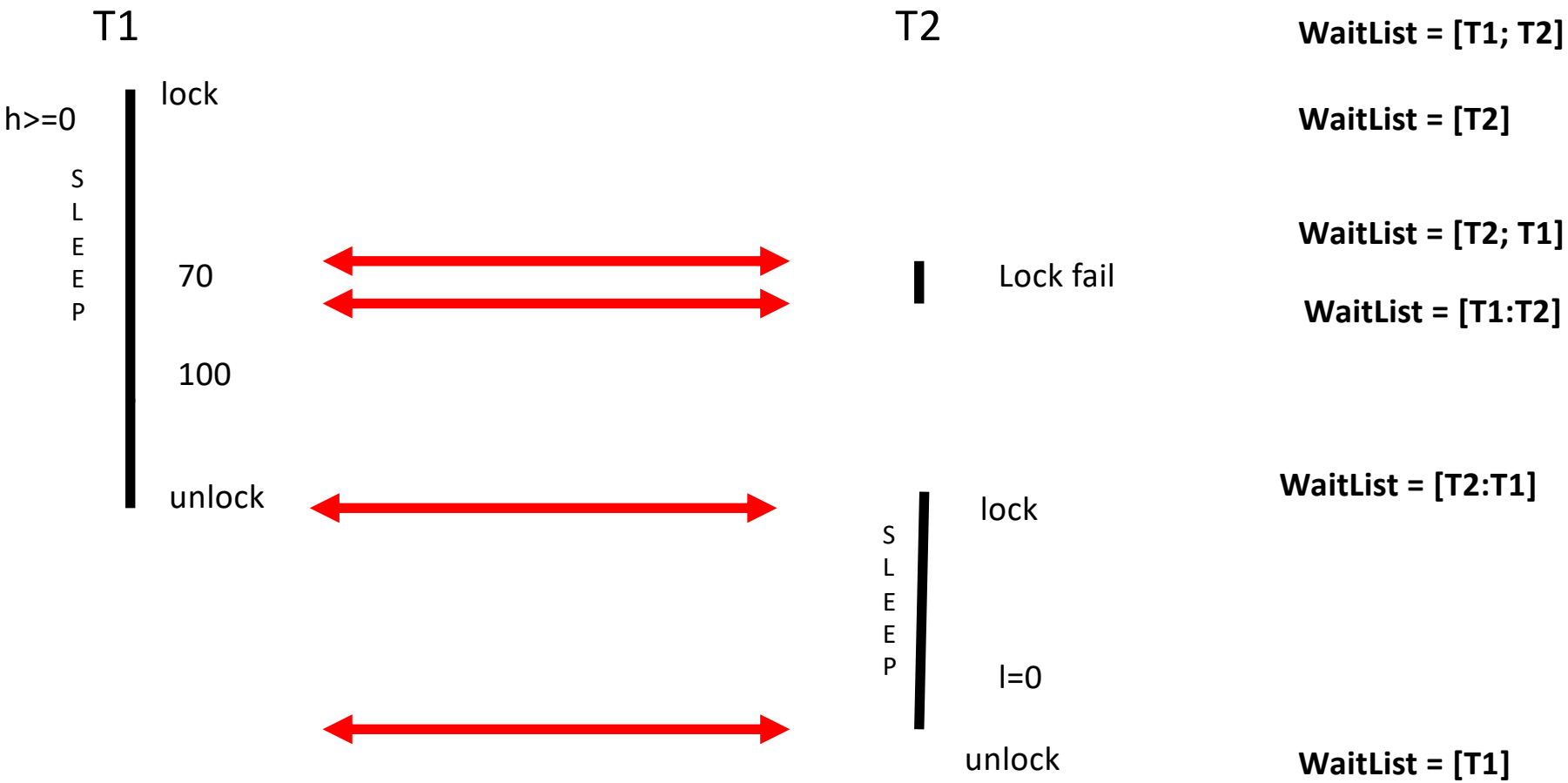
Our assumption (to simplify reasoning) Sleep(n) is equal to *n* execution steps which can be simulated by n*skip

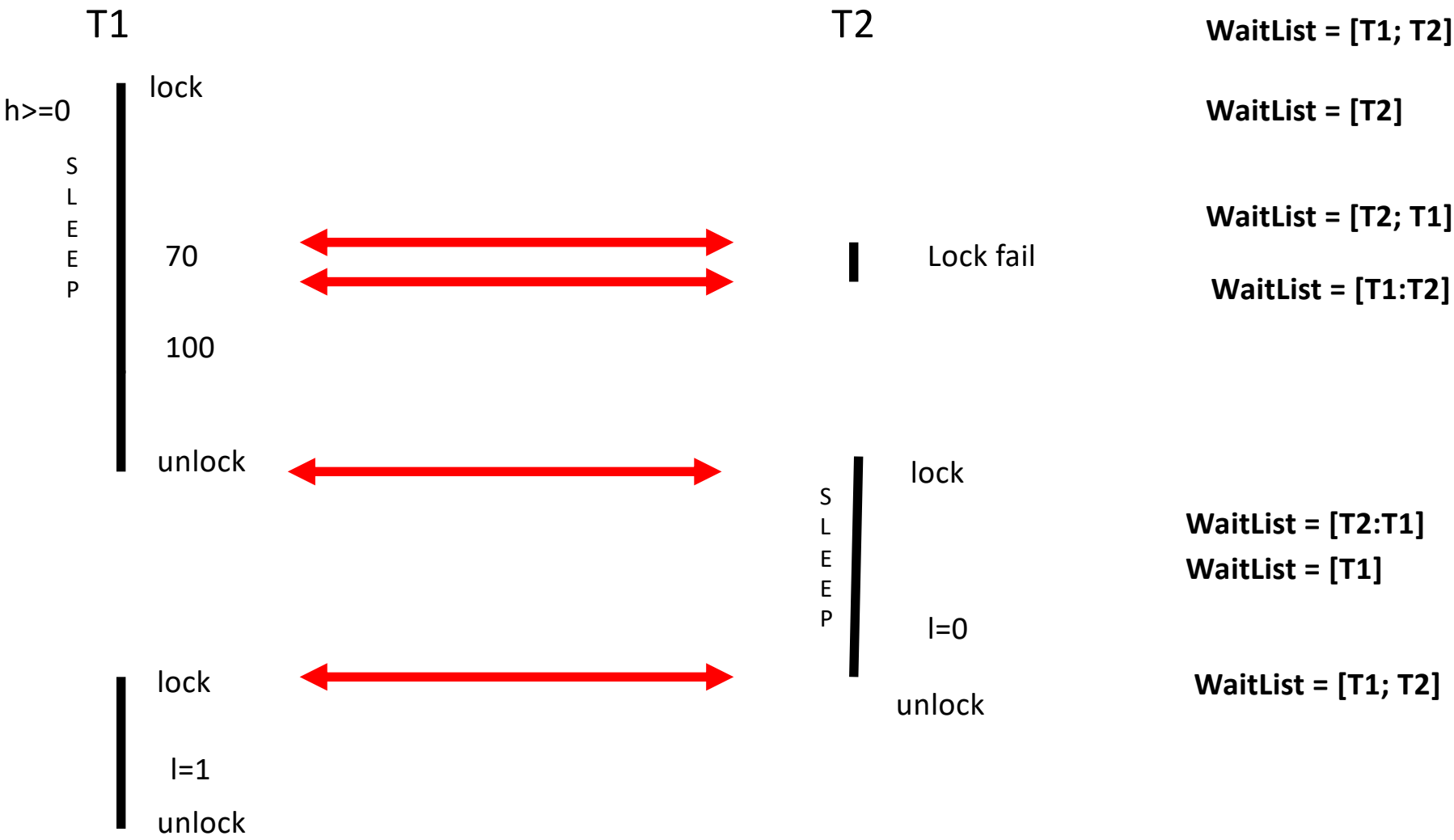
This corresponds to assume time slicing scheduling.The aim of time slicing scheduling is to give all threads an equal opportunity to use CPU

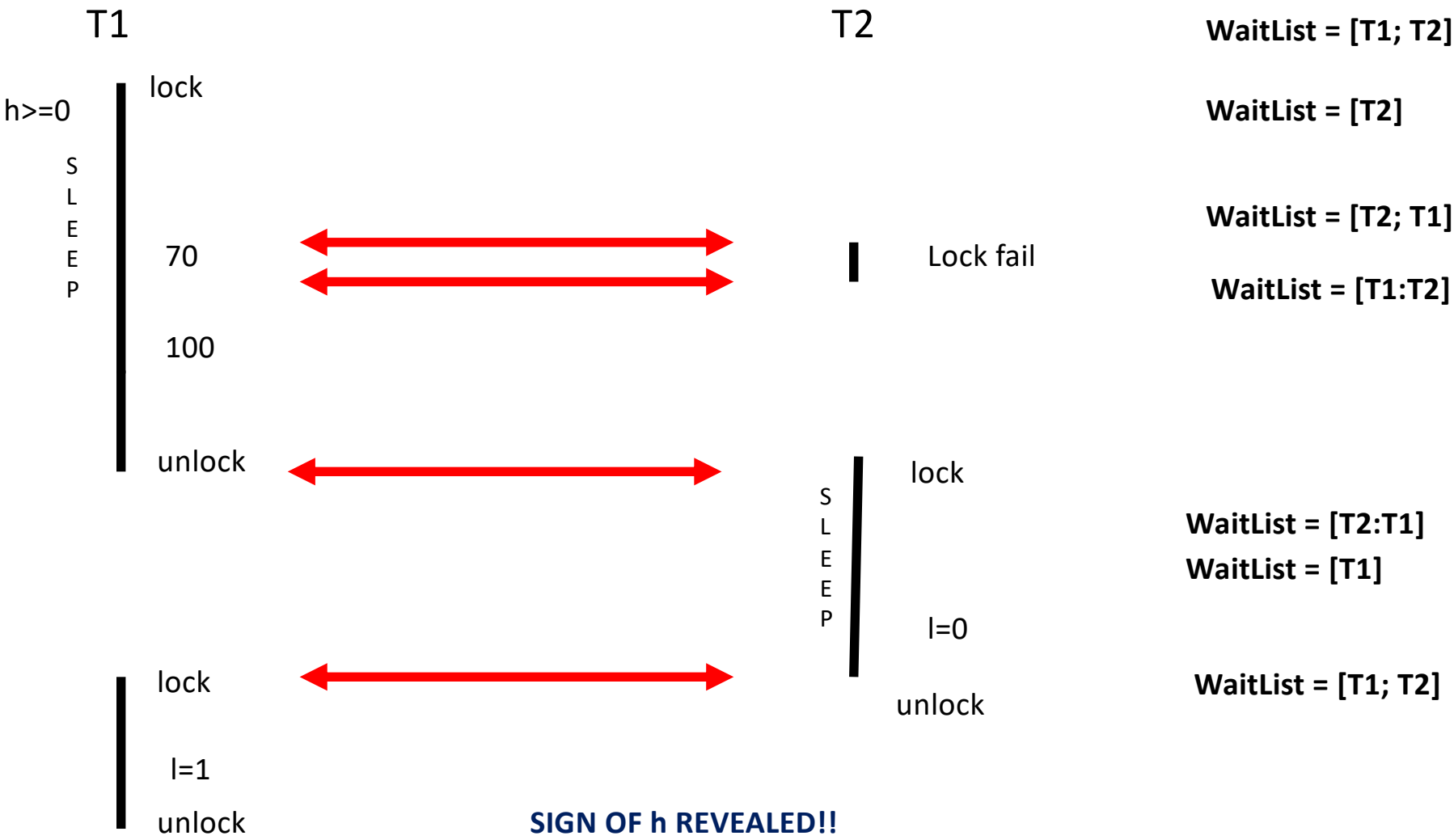

T1
T2
WaitList = [T1; T2]
h>=0
lock
WaitList = [T2]
S
L
E
E
P
70
WaitList = [T2; T1]
100


T1
T2
WaitList = [T1; T2]
h>=0
lock
WaitList = [T2]
S
L
E
E
P
WaitList = [T2; T1]
70
Lock fail
WaitList = [T1:T2]
100


T1
T2
WaitList = [T1; T2]
h>=0
lock
WaitList = [T2]
S
L
E
E
P
WaitList = [T2; T1]
70
Lock fail
WaitList = [T1:T2]
100
unlock
WaitList = [T2:T1]


T1
T2
WaitList = [T1; T2]
h>=0
lock
WaitList = [T2]
SLEEP
WaitList = [T2; T1]
70
Lock fail
WaitList = [T1:T2]
100
WaitList = [T2:T1]
unlock
lock
SLEEP
l=0
unlock
WaitList = [T1]


T1
T2
WaitList = [T1; T2]
h>=0
lock
WaitList = [T2]
SLEEP
WaitList = [T2; T1]
70
Lock fail
WaitList = [T1:T2]
100
unlock
lock
WaitList = [T2:T1]
SLEEP
WaitList = [T1]
l=0
lock
WaitList = [T1; T2]
unlock
l=1
unlock


T1
T2
WaitList = [T1; T2]
h>=0
lock
WaitList = [T2]
SLEEP
WaitList = [T2; T1]
70
Lock fail
WaitList = [T1:T2]
100
unlock
lock
WaitList = [T2:T1]
SLEEP
WaitList = [T1]
l=0
lock
WaitList = [T1; T2]
unlock
l=1
unlock
SIGN OF h REVEALED!!

Dynamic mechanisms: decrease false positives over static mechanisms through the use of run-time information.

What are the constraints that a, b have to respect to ensure well typedness?

Example

```
if (a==b) then {while (a>0){a = a-1;}
          else {a = a -1;}
```

QUIZ: Identify the information flow security policies over the lattice $\mathbf{L} \sqsubseteq \boldsymbol{H}$, namely the typing context $\boldsymbol{\Gamma}$, under which the program above is well typed