Substructural Information Flow via Polymorphism

Hemant Sai Gouni

10/21/2024

Explaining Information Flow

↓

Information Flow via Polymorphism

↓

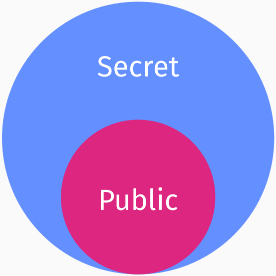
Substructural Information Flow

An Opinionated Guide to Information Flow 🧩

An Opinionated Guide to Information Flow 🧩

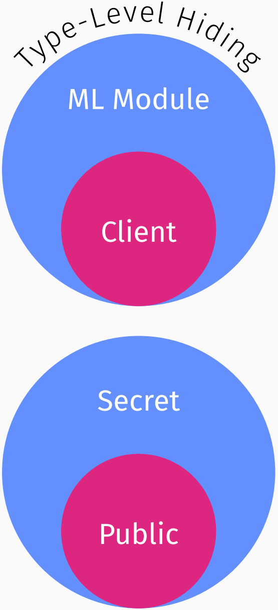
source	source & destination	destination

Information Flow is About Separation



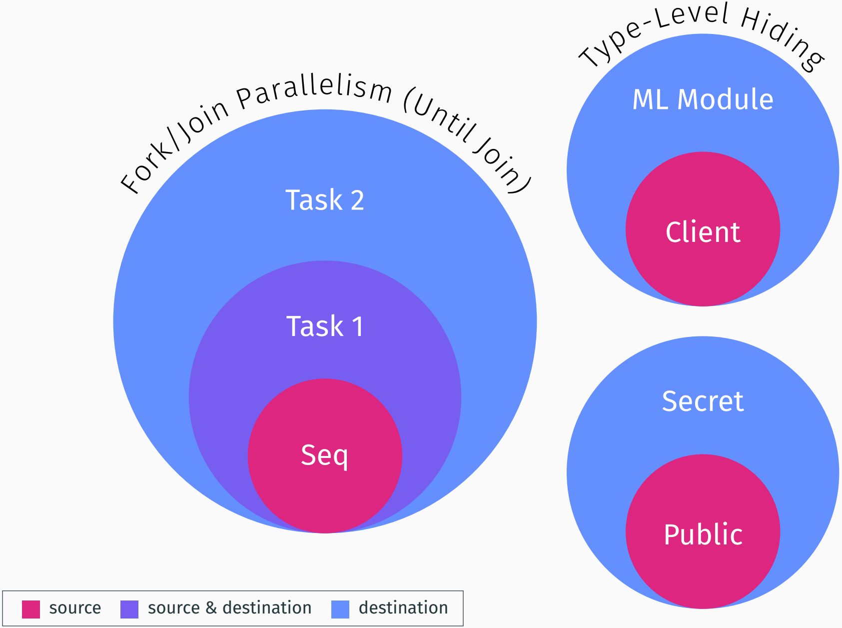
source | source & destination | destination

Information Flow is About Separation

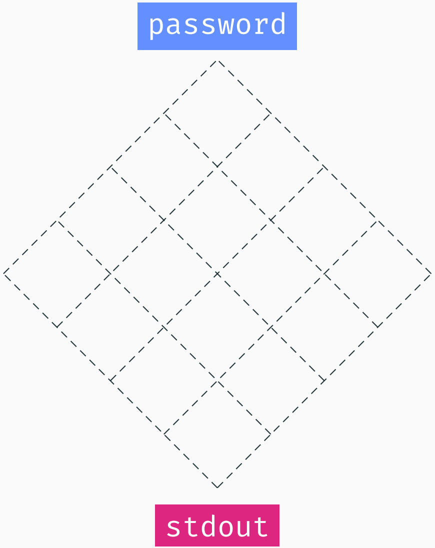


source · source & destination · destination

Information Flow is About Separation

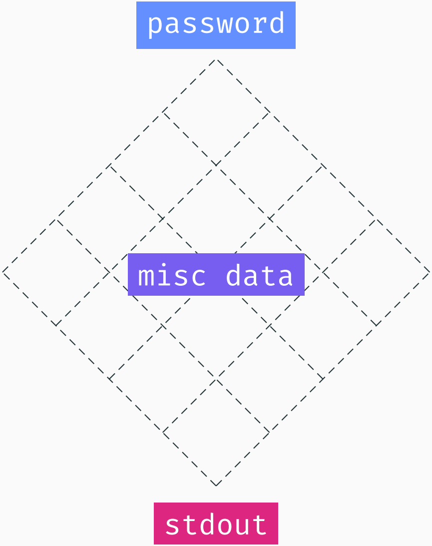


Information Flow, Classically



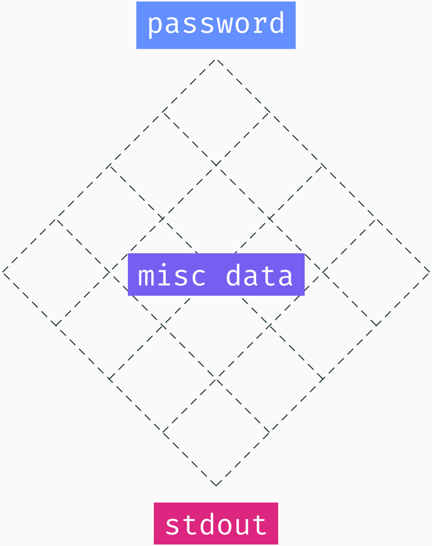
source   source & destination   destination

Information Flow, Classically



source   source & destination   destination

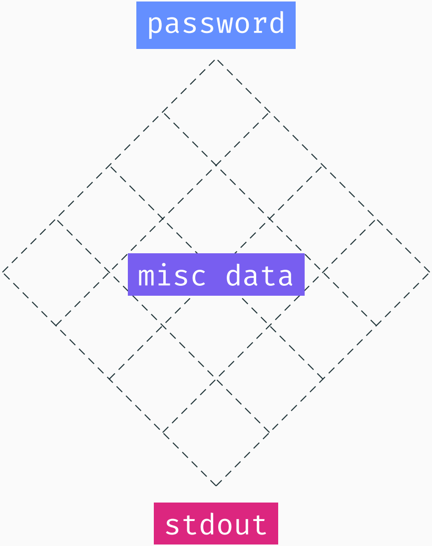
Information Flow, Classically



`"a" ++ "b" = "ab"`

■ source ■ source & destination ■ destination

Information Flow, Classically

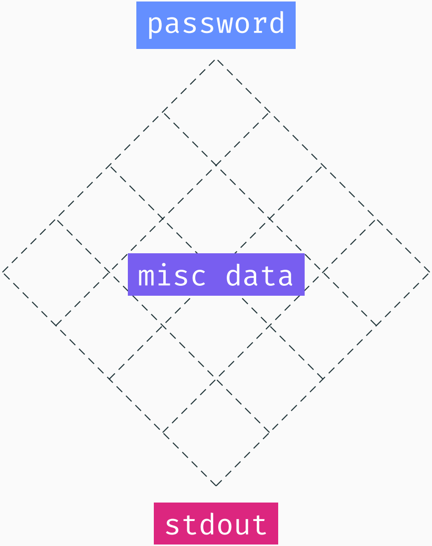


`"a" ++ "b" = "ab"`

`"a" ++ "pw" = "apw"`

source   source & destination   destination

Information Flow, Classically



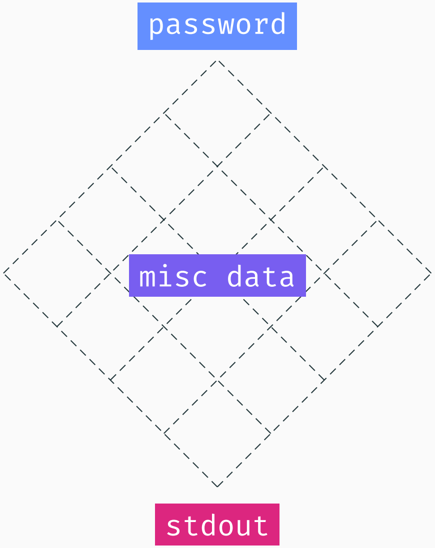
"a" ++ "b" = "ab"

"a" ++ "pw" = "apw"

stdout can flow to password

source | source & destination | destination

Information Flow, Classically



`"a" ++ "b" = "ab"`

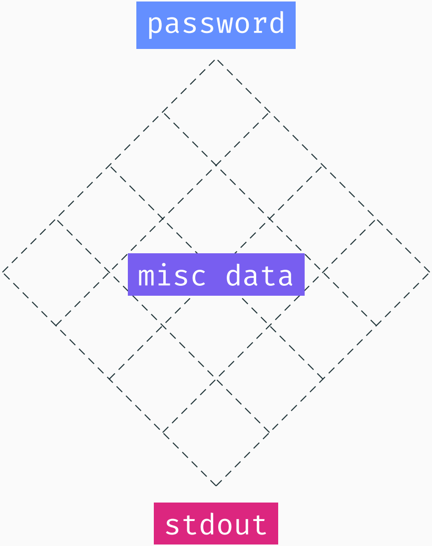
`"a" ++ "pw" = "apw"`

`stdout` can flow to `password`

`password` cannot flow to `stdout`

source · source & destination · destination

Information Flow, Classically



$\texttt{"a"} \sqcup \texttt{"b"} = \texttt{"ab"}$

$\texttt{"a"} \sqcup \texttt{"pw"} = \texttt{"apw"}$

$\texttt{stdout} \sqsubseteq \texttt{password}$

$\texttt{password} \not\sqsubseteq \texttt{stdout}$

source | source & destination | destination

Quick demonstration! 🧪

Information Flow via ✨ Polymorphism ✨

Information Flow via ✨ Polymorphism ✨

expressions   dependencies   types

Information Flow is Implemented by Tracking Dependencies

`e : [ a b ] int`

expressions dependencies types

Information Flow is Implemented by Tracking Dependencies

```
e : [ a b ] int
```

expression

expressions · dependencies · types

Information Flow is Implemented by Tracking Dependencies

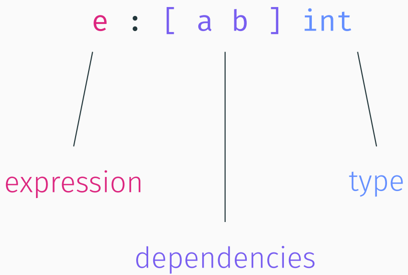
```
e : [ a b ] int
```

expression

dependencies

expressions dependencies types

Information Flow is Implemented by Tracking Dependencies



expressions dependencies types

Information Flow is Implemented by Tracking Dependencies

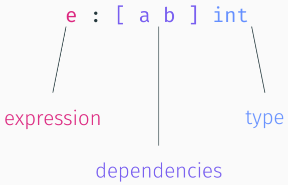
```
e : [ a b ] int
```

expression — dependencies — type

- Read `e : [ a b ] int` as "expression `e` is dependent on data from sources `a`, `b` with type `int`."

expressions dependencies types

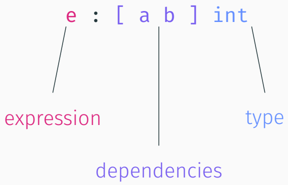
Information Flow is Implemented by Tracking Dependencies



- Read `e : [ a b ] int` as ”expression `e` is dependent on data from sources `a`, `b` with type `int`.”
    - `[ a b ]` tells you how something was computed

expressions dependencies types

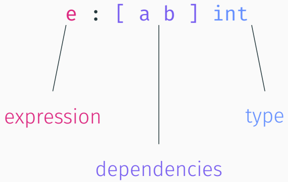
Information Flow is Implemented by Tracking Dependencies



- Read `e : [ a b ] int` as "expression `e` is dependent on data from sources `a`, `b` with type `int`."
  - `[ a b ]` tells you how something was computed
  - `int` tells you what that thing is

expressions dependencies types

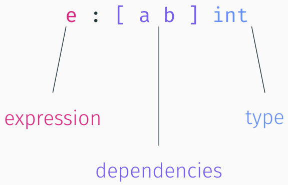
Information Flow is Implemented by Tracking Dependencies



- Read `e : [ a b ] int` as "expression `e` is dependent on data from sources `a`, `b` with type `int`."
    - `[ a b ]` tells you how something was computed
    - `int` tells you what that thing is
- Track information dependencies in types

expressions dependencies types

Information Flow is Implemented by Tracking Dependencies



- Read `e : [ a b ] int` as "expression `e` is dependent on data from sources `a`, `b` with type `int`."
    - `[ a b ]` tells you how something was computed
    - `int` tells you what that thing is
- Track information dependencies in types
    - Flows induce dependencies

expressions dependencies types

Information Flow is Implemented by Tracking Dependencies

```
e : [ a b ] int
```

expression

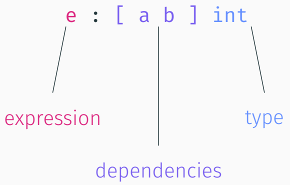
dependencies

type

```
let fst : [ a ] int -> [ b ] int -> [ a ] int
let fst x y = x
```

expressions dependencies types

Information Flow is Implemented by Tracking Dependencies

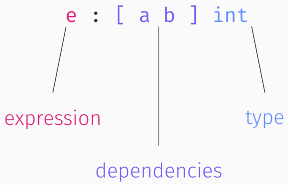



```
let fst : [ a ] int -> [ b ] int -> [ a ] int
let fst x y = x

let both : [ a ] int -> [ b ] int -> [ a b ] int
let both x y = x + y
```

expressions dependencies types

Information Flow is Implemented by Tracking Dependencies




```
let fst : [ a ] int -> [ b ] int -> [ a ] int
let fst x y = x

let both : [ a ] int -> [ b ] int -> [ a b ] int
let both x y = x + y

let br : [ a ] bool -> [ b ] int -> [ a b ] int
let br cond x = if cond then x else 0
```

expressions dependencies types

Why's this simpler?

```
auth : a string -> (b int -> c bool) -> d bool
where tok ⊑ b
      pwd ⊔ a ⊔ c ⊑ d
```

expressions | dependencies | types

Why's this simpler?

$\texttt{tok} \sqsubseteq \texttt{b}$

$\texttt{pwd} \sqcup \texttt{a} \sqcup \texttt{c} \sqsubseteq \texttt{d}$

expressions dependencies types

Why's this simpler?

`tok ⊑ b`

`pwd ⊔ a ⊔ c ⊑ d`

■ expressions ■ dependencies ■ types

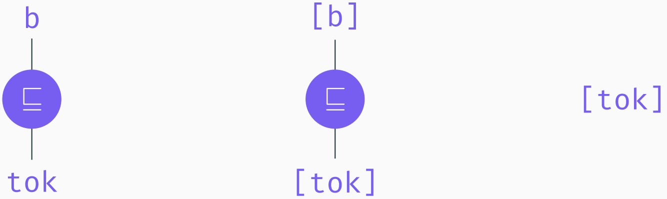
Why's this simpler?

$\mathtt{tok} \sqsubseteq \mathtt{b}$

$\mathtt{pwd} \sqcup \mathtt{a} \sqcup \mathtt{c} \sqsubseteq \mathtt{d}$

expressions dependencies types

Why's this simpler?



expressions dependencies types

Why's this simpler?

$\texttt{tok} \sqsubseteq \texttt{b}$

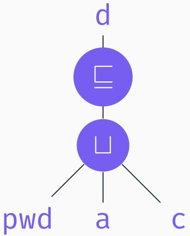
$\texttt{pwd} \sqcup \texttt{a} \sqcup \texttt{c} \sqsubseteq \texttt{d}$

expressions dependencies types

Why's this simpler?

$\texttt{tok} \sqsubseteq \texttt{b}$

$\texttt{pwd} \sqcup \texttt{a} \sqcup \texttt{c} \sqsubseteq \texttt{d}$

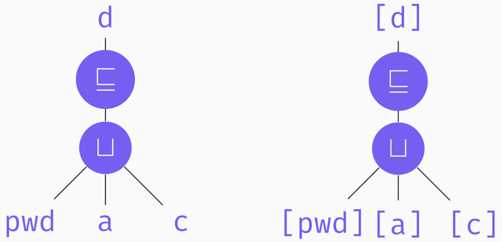


expressions dependencies types

Why's this simpler?

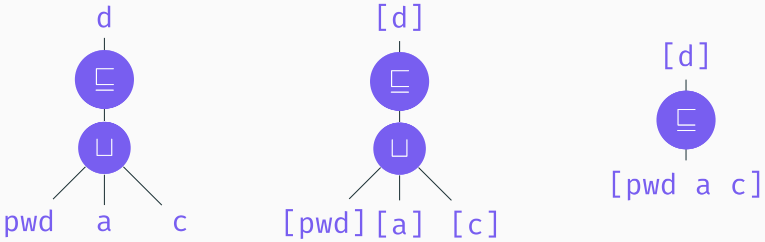
$\texttt{tok} \sqsubseteq \texttt{b}$

$\texttt{pwd} \sqcup \texttt{a} \sqcup \texttt{c} \sqsubseteq \texttt{d}$



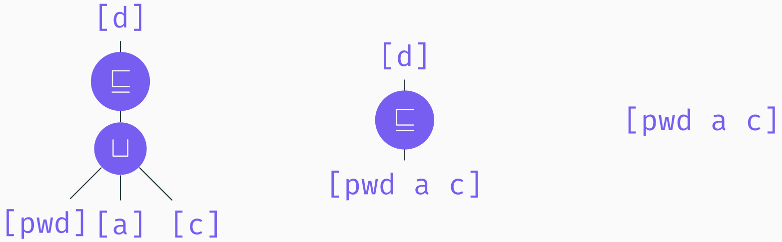
expressions dependencies types

Why's this simpler?



expressions dependencies types

Why's this simpler?



expressions dependencies types

Why's this simpler?

Lattices

```
a string -> (b int -> c bool) -> d bool
where tok ⊑ b
      pwd ⊔ a ⊔ c ⊑ d
```

Polymorphism

```
[a] string -> ([tok] int -> [c] bool) -> [pwd a c] bool
```

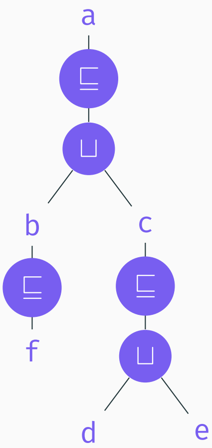
expressions · dependencies · types

Why's this simpler?

Lattices

A fragment of a more complicated Flow Caml type:

```
a string where b ⊔ c ⊑ a
               f ⊑ b
               d ⊔ e ⊑ c
```



Polymorphism

```
[ d e f ] string
```

✨ Substructural ✨ Information Flow

✨ Substructural ✨ Information Flow

■ exchange ■ weakening ■ contraction

Is [x y] the same as [y x]?

Is [x y] the same as [x y z]?

Is [x x] the same as [x]?

exchange | weakening | contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
  - With ✅

```
let id : [ 🐟 ] int -> [ 🐟 🐠 ] int
let id x = x
```

- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
  - With ✅

    ```
    let id : [ 🐟 ] int -> [ 🐟 🐠 ] int
    let id x = x
    ```

  - Without 🚫

    ```
    let id : [ 🐟 ] int -> [ 🐟 ] int
    let id x = x
    ```

- Contraction
- Exchange

exchange | weakening | contraction

- What does it mean to get rid of these rules?
- Weakening
  - With ✅

    ```
    let id : [ 🐟 ] int -> [ 🐟 🐠 ] int
    let id x = x
    ```

  - Without 🚫

    ```
    let id : [ 🐟 ] int -> [ 🐟 ] int
    let id x = x
    ```

  - Error 💥

    ```
    let br : [ 🐟 ] bool ->
         [ 🐠 ] int -> [ 🐟 🐠 ] int
    let br b x = if b then x else 0
    ```

    - No type for this term...?
- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction

- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
  - With ✅

```
let x2 : [ 🐟 ] int -> [ 🐟 ] int
let x2 x = x + x
```

- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
  - With ✅

    ```
    let x2 : [ 🐟 ] int -> [ 🐟 ] int
    let x2 x = x + x
    ```

  - Without 🚫

    ```
    let x2 : [ 🐟 ] int -> [ 🐟 🐟 ] int
    let x2 x = x + x
    ```

- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange
  - With ✅

```
let xy : [ 🐟 ] -> [ 🐠 ] -> [ 🐟 🐠 ]
let xy x y = y + x
```

exchange weakening contraction

- What does it mean to get rid of these rules?
- Weakening
- Contraction
- Exchange
  - With ✅

    ```
    let xy : [ 🐟 ] -> [ 🐠 ] -> [ 🐟 🐠 ]
    let xy x y = y + x
    ```

  - Without 🚫

    ```
    let xy : [ 🐟 ] -> [ 🐠 ] -> [ 🐠 🐟 ]
    let xy x y = y + x
    ```

exchange weakening contraction

I can't have things for free anymore... except capabilities!

Capability reasoning for free from dropping weakening!

```
module type Authorize : sig
     label 💎
     let auth : [ ] password ->
          [ 💎 ] unit + [ ] unit
end
```

....

```
let sensitive_op = [ 💎 ] arg_type -> ...
```

exchange weakening contraction

I can't squish stuff together anymore...

...which lets us prevent resource exhaustion issues!

```
module type Bank : sig
    type 💰
    label 🪙
    val empty : 💰
    val get_coin : [ a ] 💰 -> [ a 🪙 ] 💰
    val transact : password ->
        (unit -> [ 🪙 🪙 🪙 ] 💰) -> unit
end
```

```
transact (Password "katya")
    (fun _ -> (get_coin (get_coin empty)))
```

exchange weakening contraction

Substructural Non-Interference 🦖🔥

Turbocharging Non-Interference

Definition of Non-Interference

You must not be able to turn something you ***cannot*** *observe into something you* ***can*** *observe.*

exchange weakening contraction

Turbocharging Non-Interference

Definition of Non-Interference

You must not be able to turn something you ***cannot*** *observe into something you* ***can*** *observe.*

Central Idea

The structural rules ***define*** *your powers of* ***observation****.*

exchange | weakening | contraction

Turbocharging Non-Interference 💨🏎️

Definition of Non-Interference

*You must not be able to turn something you **cannot** observe into something you **can** observe.*

Central Idea

*The structural rules **define** your powers of **observation**.*

- Can't get a 💎 for free because that would be a violation!

exchange weakening contraction

I can't have things for free anymore... except capabilities!

Capability reasoning for free from dropping weakening!

```
module type Authorize : sig
    label 💎
    let auth : [ ] password ->
        [ 💎 ] unit + [ ] unit
end
```

....

```
let sensitive_op = [ 💎 ] arg_type -> ...
```

exchange weakening contraction

Turbocharging Non-Interference 💨🏎️

[🐟 🐠]
[x y] 🔮 [🪙 🪙]
[💎] [a]

Definition of Non-Interference

*You must not be able to turn something you **cannot** observe into something you **can** observe.*

Central Idea

*The structural rules **define** your powers of **observation**.*

- Can't get a 💎 for free because that would be a violation!
- Can't lie about the number of 🪙 we've got in our bag!

exchange | weakening | contraction

I can’t squish stuff together anymore…

…which lets us prevent resource exhaustion issues!

```
module type Bank : sig
    type 💰
    label 🪙
    val empty : 💰
    val get_coin : [ a ] 💰 -> [ a 🪙 ] 💰
    val transact : password ->
        (unit -> [ 🪙 🪙 🪙 ] 💰) -> unit
end
```

```
transact (Password "katya")
    (fun _ -> (get_coin (get_coin empty)))
```

exchange weakening contraction

Other Cool Work 🧭

Granule: General framework for graded type theories. Our system could be embedded in theirs by extending their compiler with the appropriate SMT encoding.

- Substructural non-interference offers strong guarantees for this kind of reasoning anywhere, whether in a standalone implementation or for embeddings into a more general setting.

Session Types, Choreographies, Typestate: Focus on the *future* rather than the past: they only tell you what *can be done* computationally, not what form a computation *already has*.

- Occasionally adopt slightly more alien computational models like process calculi.

Information flow can be captured using familiar machinery for parametric polymorphism.

Building on this, substructural information flow provides essential security and behavioral reasoning tools.

These tools have been proved to be sound via substructural non-interference, a powerful property that generalizes typical non-interference.

hsgouni@cs.cmu.edu / @hgouni@hci.social